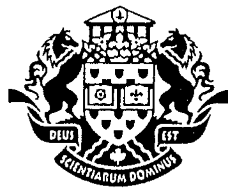


Université d'Ottawa · University of Ottawa

Université d'Ottawa · University of Ottawa

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTDOCTORALES

FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

Truong NGUYEN
AUTEUR DE LA THÈSE - AUTHOR OF THESIS

M. A. Sc. (Electrical Engineering)
GRADE - DEGREE

Department of Electrical Engineering
FACULTÉ, ÉCOLE, DÉPARTEMENT - FACULTY, SCHOOL, DEPARTMENT

TITRE DE LA THÈSE - TITLE OF THE THESIS

Efficient Determination of Consensus Secondary Structures in RNA

M. Turcotte
DIRECTEUR DE LA THÈSE - THESIS SUPERVISOR

CO-DIRECTEUR DE LA THÈSE - THESIS CO-SUPERVISOR

EXAMINATEURS DE LA THÈSE - THESIS EXAMINERS

D. Nussbaum

H. Viktor

J.-M. De Koninck, Ph.D

LE DOYEN DE LA FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTDOCTORALES

DEAN OF THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

Efficient Determination of Consensus Secondary Structures in RNA

Truong Nguyen

Thesis submitted to the
Faculty of Graduate and Postdoctoral Studies
in partial fulfillment of the requirements for the degree of

Master of Applied Science
in
Electrical Engineering

School of Information Technology and Engineering
Faculty of Engineering
University of Ottawa

April 2004

©Truong Nguyen, Ottawa, Canada, 2004

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 0-494-01562-4
Our file Notre référence
ISBN: 0-494-01562-4

NOTICE:
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

1 Introduction 10
1.1 Background and Rationale 10
1.2 Methodology 14
1.2.1 Hypothesis Space 15
1.2.2 Search Algorithm 16
1.2.3 Scoring Schemes 20
1.2.4 Scope of Work 21
1.2.5 Thesis Organization 21
2 Suffix Trees and Suffix Arrays 22
2.1 Suffix Trees 22
2.1.1 Suffix Tree Description 23
2.1.2 Suffix Tree Formalisms 24
2.1.3 Simplified Suffix Tree Construction 25
2.1.4 Classical Applications of Suffix Trees 26
2.1.5 Suffix Tree Construction 35
2.2 Suffix Arrays 39
2.2.1 Overview 40
2.2.2 Suffix Array Construction 43
2.2.3 Suffix Array Applications 45
3 Enumeration of Secondary Structure Motifs for One Sequence 56
3.1 Finding and Representing Stems 56
3.2 Assembling Stems into Motifs 61

3.3 Motif List Pruning 67

4 Secondary Structure Motif Inference 71

4.1 Secondary Structure Motifs: Ambiguity to Specificity 71

4.2 Structural Motif Search Methodology 79

4.3 Scoring Results 85

5 Cmyc Data 87

5.1 Data Set 87

5.2 Results 90

5.2.1 Experimental Results 91

5.2.2 Discussion of Results 97

6 Conclusion and Discussion 99

6.1 Conclusion 99

6.2 Discussion 100

7 Appendix 103

7.1 IUPAC Codes 103

7.2 Additional Simulation Results 104

7.3 Execution Statistics 104

List of Figures

Figure 1.1 - Secondary structure representation of the Histone 3 motif found in the 3' NTR mRNAs of Metazoan. 13

Figure 2.1 - Suffix tree for the string S = abcbcde.......... 24

Figure 2.2 - Lowest common ancestor path and node (bolded) of the leaf nodes for the suffixes bcbcde and bcde. 31

Figure 2.3 - Suffix tree for string S = abcbcde, with leaf numbers indicting the position of the suffix in S. 40

Figure 2.4 - Suffixes for string S = abcbcde. 41

Figure 2.5 - Suffix array for string S = abcbcde. 42

Figure 2.6 - Suffix tree for suffix S = abcbcde. 47

Figure 2.7 - LCP interval tree for the adjacent LCP array of the string S = abcbcde....... 49

Figure 2.8 - LCP interval tree overlaid with the corresponding suffix tree.......... 49

Figure 3.1 - Example of a stem-loop structure to be detected. 57

Figure 3.2 - Concatenation of forward and reverse complemented sequence. 57

Figure 3.3 - Stem loop structure with a two nucleotide bulge.......... 58

Figure 3.4 - Secondary structure comprised of recursively nested stem segments. 59

Figure 3.5 - Stem loop structure with a base pairing mismatch.......... 60

Figure 3.6 - Illustration of a stem segment nested within another stem segment.......... 62

Figure 3.7 - Illustration of a more complex secondary structure consisting of three separate stem segments. 63

Figure 3.8 - Secondary structure consisting of four stems.......... 65

Figure 3.9 - Binary tree representation of a motif comprised of four stem segments. 65

Figure 4.1 - Assembled stem segment combination from the solution space.................. 72

Figure 4.2 - Assembled stem segment combination reduced to generic structural form.. 74

Figure 4.3 - An example instantiation of the generic structural motif............................ 78

Figure 4.4 - Node chain for the structural motif of Figure 4.2. 81

Figure 5.1 - Sample of the Cmyc data sequences.. 88

Figure 5.2 - Cmyc sequence data corresponding to the human data. 89

Figure 5.3 - mfold secondary structure prediction of the human IRES sequence data..... 90

Figure 5.4 – Output result of Experiment 1.. 93

Figure 5.5 - Secondary structure representation of consensus motif for Experiment 1.... 94

Figure 5.6 - Output of result for Experiment 2.. 95

Figure 5.7 - Secondary structure representation of consensus motif for Experiment 2.... 96

List of Tables

Table 2.1- Relationship between suffixes and adjacent LCP values. 47

Table 5.1 - Criteria used for Experiment 1. 93

Table 5.2 - Criteria used for Experiment 2. 95

Table 5.3 - Attributes of positively correlated motif predictions..................................... 97

Table 7.1 - Various IUPAC Codes. 103

Table 7.2 - Results obtained by varying the gap size. 104

Table 7.3 - Results obtained by varying the confidence level. 104

List of Abbreviations

DNA – Deoxyribonucleic acid

IRES – Internal ribosome entry site

IUPAC – International Union of Pure and Applied Chemistry

LCA – Lowest common ancestor

LCP – Longest common prefix

NTR – Non-translated region

RNA – Ribonucleic acid

Acknowledgements

This work would not be possible without the assistance, inspiration, and knowledge of Dr. Marcel Turcotte. I am indebted to him.

Additional thanks are expressed to Martin Holcik and Stephen Baird from the Children's Hospital of Eastern Ontario, whose research on IRES motifs provided the foundation for this work.

Finally, heartfelt gratitude is extended to my parents, sisters, and to Phu, for their unwavering support.

Abstract

Transcription and translation are critical steps through which genetic expression occurs. Whereas there exists research for computationally determining the primary structure binding sites for transcription, research into the computational elucidation of secondary structure binding sites for translation has not been as thoroughly conducted.

The approach proposed involves first selecting a single sequence from a set of data sequences. From this sequence, all biological palindromes are determined. Using these palindromes, all possible candidate secondary structure motifs with minimum support are assembled, formulating the solution space.

The motifs in the solution space are reduced to structural form. These structures are searched against the remaining sequences. Negative matches are discarded, while positive matches converge to consensus secondary structure motifs by specification of constituent base pairs. Positively matched motifs are ranked according to the amount of information content.

The analysis techniques presented yielded promising results. Consensus secondary structures were successfully elucidated from the source sequences given appropriate constraints.

1 Introduction

Transcription and translation are two of the critical steps through which genetic expression occur at the cellular level. Transcription is the process whereby a segment of DNA (deoxyribonucleic acid) serves as a template to produce a complementary RNA (ribonucleic acid) sequence. The RNA strand is then used in the process of protein synthesis via translation. Whereas the transcription process involves the primary structure of DNA, the transcription process is hypothesized to additionally involve the binding of proteins to secondary structures, or motifs, in RNA. Determination of these secondary structure regulatory motifs is an inherently complex mathematical and algorithmic problem, and as such has lagged parallel developments in the field of determining primary regulatory sequences. This work details a multi-staged methodology that attempts to solve the problem of determining secondary structure regulatory motifs.

1.1 Background and Rationale

Regulation of a gene expression occurs during the transcription of a gene. Typically, this involves the binding of proteins near the beginning of the transcription phase and also affects the rate of the initiation of protein synthesis. Several pattern recognition algorithms have been developed to predict the location of the primary structures binding sites of the transcription factors. These programs primarily rely on empirically derived evidence that indicates that a particular set of genes are regulated by the same factors.

Using this information, the various programs then attempt to infer a sequence pattern that is common to all of the genes. The inference* of these patterns is done using both probabilistic and deterministic methodologies.

Although much less studied in academia, post-transcriptional regulation of gene expression, or translational regulation, is also common. Whereas transcriptional regulation is controlled by proteins binding to the primary structures of regulatory DNA sequences, RNA translational regulation is hypothesized to involve both binding with respect to a primary structure, as well as binding to a secondary structure. A set of similar secondary structure binding sites is referred to as a structural pattern, or motif.

The aim of the research, conducted and presented by this work, is to develop a new algorithm to computationally discover and determine structural motifs in nucleic acid sequences. To this end, specific empirically derived RNA regulatory elements are used as the source data set. Specifically, the Internal Ribosome Entry Site (IRES) [1] is used as the model RNA regulatory element. The IRES element is a postulated regulatory motif that is present in one end of non-translated region (NTR) of certain gene transcripts. The IRES element is thought to allow for the translation of the transcript while bypassing some of the common cellular regulatory mechanisms.

---

* Note that the usage of inference in this thesis is in the literal sense of the word, rather than implying the more complex notion used in logic theory.

Nucleic acid secondary structures involve the base pairing of nucleotides which can exist in a variety of manners and permutations in a sequence. A nucleic acid secondary structure occurs when a segment within a single nucleic acid strand base pairs with another segment on the same strand. Recall that a nucleic acid is a long polymer that is composed of a chain of individual nucleotides. A nucleic acid chain is terminated at one end by a free phosphate group, denoted by the 5' label. At the other end, the nucleic acid chain is terminated by a free hydroxyl group, denoted by the 3' label. Each nucleotide contains a nitrogenous base. For DNA, there are four possible bases – adenine (A), cytosine (C), guanine (G), and thymine (T). In the case of RNA, the bases present are identical except that instead of the thymine (T) base, a uracil (U) base is present. The bases can form hydrogen bonds with certain other bases; pairs of bases with hydrogen bonds are commonly referred to as base pairs. The base adenine base pairs with thymine or uracil, depending on whether the nucleic acid in question is DNA or RNA respectively[2]. The base cytosine base pairs with guanine. Bases that base pair with one another are considered to be complementary.

A secondary structure composed of a continuous segment of base pairs can be thought of as a biological palindrome[3]. Whereas a lexical palindrome consists of a string that reads the same both in the forward and reverse directions, a biological palindrome is complementary in the forward compared to the reverse direction. That is, a sequence consisting of ACCGU would have a biological palindrome of ACGGU and would appear

as a segment of ACCGUACGGU in an RNA sequence. Furthermore, as in lexical palindromes, biological palindromes can also have a separation between the palindrome segments. This separation is known as the gap in the palindrome. Modifying the above example into a biological palindrome with a gap yields ACCGUACACGGU, where the bases AC form the gap. Figure 1.1 depicts an example gapped biological palindrome, organized in a secondary structure representation. Note that the secondary structure builds from the 5' end to the 3' end. The secondary structure formation depicted in Figure 1.1 is known as a hairpin formation, or more generically, as a stem and loop formation. The dashes represent nucleotides that are base paired with one another. Note that the Y and R symbols are IUPAC representations for C or T and A or G respectively[4]. The complementary segments comprise the stem, and the gap comprises the loop. In this case, the loop consists of the nucleotide sequence HHUH. By recursively nesting hairpin structures, and combining sets of nested hairpin structures, more complex secondary structures can be described.

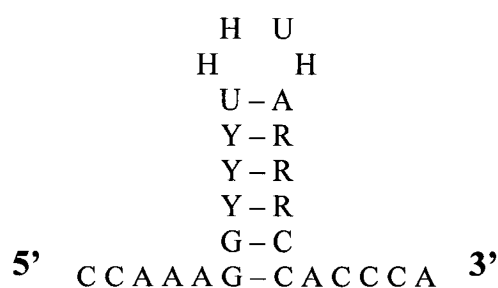


Figure 1.1 - Secondary structure representation of the Histone 3 motif found in the 3' NTR mRNAs of Metazoan.[5]

This work investigates the development of an efficient, computational methodology for inferring a generic secondary structure motif common throughout a set of unaligned RNA sequences. Such an efficient combinatorial algorithm would enable the discovery of functional motifs, as well as insights regarding the translation and regulation of genes that share these motifs.

1.2 Methodology

There are several known approaches to pattern discovery for a set of sequences. Some of these methods have a pre-condition that a multiple sequence alignment must already exist[6 7 8]. This work investigates methodologies which do not require previously aligned sequences as input. Furthermore, existing methodologies rely on determining a secondary structure motif from a single sequence and optimizing the result based on a variety of parameters and criteria[9 10 11]. The pattern discovery methodologies investigated in this work attempt to infer consensus secondary structures by instead utilizing a wider set of related source data sequences.

As proposed by Brazma et al.[12], a pattern discovery methodology is characterized by three essential criteria:

- The formalization of a hypothesis and solution space which characterizes and elucidates the motifs to be found.

- The development of an efficient search algorithm that is capable of annotating the patterns from the solution space. These patterns can also be enumerated in an order dictated by the scoring algorithm (below).
- The definition of a scoring/fitness function that evaluates the effectiveness of a derived pattern in reflecting the underlying source sequences. The scoring/fitness function yields a numerical score for the pattern, which is used to rank the derived pattern against other patterns derived by the search algorithm.

1.2.1 Hypothesis Space

In order to determine a consensus secondary structure within a target set of data, a set of hypothesized motifs must be generated. Currently, there exist software packages that can predict optimal and suboptimal secondary structures when given an RNA sequence. One of the more prominent programs is mfold[13] which utilizes a set of nearest neighbour thermodynamic criteria in its prediction. By using calculations with regards to free energy, mfold attempts to predict RNA secondary structures. However, mfold does not provide the level of granularity and breadth required for the determination of consensus sequences from a set of data sequences. Furthermore, mfold is designed to predict the optimal structure for RNA sequences where most nucleotides are thought to be involved in base pairs. Within the target set of data used by this work, it is thought that the motif is made of a small fraction of a relatively larger set of bases[1].

Suffix trees are a powerful and versatile data structure that enables the ability to efficiently match a pattern string against a database in linear time with respect to the size of the pattern. Suffix trees can be created in linear time and occupy a linear amount of space[14]. One application of suffix trees involves the ability to determine the existence of lexical palindromes within a string in linear time with respect to the length of the string. An extension of this application involves building a suffix tree using an RNA primary sequence as the source string. Utilizing similar techniques as in the determination of lexical palindromes, the suffix tree can be used to determine the presence of all maximal biological palindromes in the RNA primary sequence.

Thus, given an RNA sequence, suffix trees may be used to determine, in linear time, all maximal biological palindromes, and thus all possible contiguous stem-loop structures, contained within. By assembling a set of all of the combinations of the contiguous stem-loop structures that obey the ordering within a sequence, an exhaustive list of possible secondary structures motifs can be derived. This list formulates a solution space of hypothesized secondary structure motifs derived from the one sequence. Using a specialized search mechanism, the elements of the solution space can then be applied to a broader set of data sequences in order to ascertain a consensus secondary structure.

1.2.2 Search Algorithm

A search algorithm is required to efficiently enumerate the structural motifs that comprise the solution space against a larger set of data sequences. The search algorithm

possibilities can be reduced according to various criteria. The completeness of the search and whether or not the search is guaranteed to find the optimal solution are important elements for the search algorithm. Possible algorithms can be classified into pattern driven or data driven approaches.

Pattern driven search methodologies enumerate all of the patterns from the hypothesis space and then rank them according to some fitness criteria. There exist secondary structure search packages such as Palingol[15] that scan a set of sequences for a secondary structure motif that must be specified. The targeted secondary structure must first be specified using a formalized language as well as constraint matrices. Palingol, however, does not provide the level of performance, nor the flexibility required for the determination of consensus secondary structures.

In order to guarantee the completeness of the search, given computational and temporal realities, the solution space can be pruned prior to, and during the search, according to various criteria. Pre-processing of the solution space can be performed by utilizing various heuristics (detailed in Section 3.3), and the fact that motifs that are not likely to be biologically relevant (e.g. low complexity motifs) can be discarded. During the search, if a particular motif fails to generate a sufficient match, then that motif and all of its specializations can be discarded. Pattern driven search methodologies require the elucidation of a pattern prior to invocation; the ability of the pattern driven search to effectively yield an optimal solution relies on the quality and quantity of the underlying

source patterns[16]. Data driven methodologies, on the other hand, require no such source pattern as it is inferred from the underlying data.

Data driven search methodologies are based upon first deriving a set of building blocks from a single input example and/or a set of input examples. These building blocks are combined to create patterns to form the hypothesis space. Such an algorithm involves the random selection of a positive example from all of the input examples. Suppose for example that k stem-loop structures can be formed from a randomly chosen input sequence. From this set of possible structures, an initial set of j patterns can be constructed through various combinations of the k stem-loop structures. The number of patterns $j \geq k$, but more likely, $j >>> k$ due to combinatorial effects. These initial patterns only contain structural information, (e.g. a stem-loop structure would be represented as NNNN---N'N'N'N', where N' is the biological complement of N). The structural patterns are then searched against the remaining sequences in the data set; any generic structural pattern that fails to match a pre-determined threshold number of sequences in the data set is discarded. As generic patterns are successfully matched against the data set, the individual patterns are further specialized and searched against the underlying data set. Similarly, more specialized patterns that fail to sufficiently match sequences in the data set are discarded. In this manner, the search converges upon a possible solution.

A data driven methodology is more likely to converge upon a possible solution faster than a pattern driven methodology. However, this approach is not guaranteed to find the

optimal solution, as the actual solution that is produced depends upon the veracity of the underlying source sequence used. A hybrid solution between the two methodologies is postulated to provide a superior solution given the problem space. A data driven methodology is initially utilized to provide an initial seed set of generic structural search sequences. Various pattern driven methodology oriented heuristics are then applied to reduce the size of the set of generic structural search sequences. These heuristics are detailed further in this work. As the generic structural patterns are matched positively or negatively to the search space, they are made more specific or discarded, respectively. Negative matches also result in the recursive reduction of any structural patterns that were derived from the negative match.

The underlying mechanism to perform the search is based upon a simultaneous traversal of the search expression and a top-down traversal of a suffix tree. Conceptually, the methodology is somewhat similar to that developed by Baeza-Yates and Gonnet[17]. However, the results of Baeza-Yates and Gonnet applies only to regular expression motifs, whereas the methodology developed in this thesis applies to a restricted class of expressions that belong to context-free languages. Furthermore, since the search expression needs to contain structural information (i.e. base pairing references), the results of that paper were not directly applied.

1.2.3 Scoring Schemes

In order to complete any pattern discovery methodology, a scoring scheme is required in addition to techniques to generate and search the solution space. The scoring scheme is required to rank the results according to some criteria such that the most valuable results are ranked the highest. Assuming that a single pattern is sought to match all of the data sequences, a common fitness function consists of calculating the information content stored within the patterns. For regular motifs, information content has proven to be an effective measure for ranking patterns[18]. The pattern that has the highest information content is considered superior to the other patterns. The information of a particular pattern is given by the sum of all of the independent contributions. In this case, the information content measures the amount of information that is gained by knowing that a particular pattern has successfully matched the data sequences. This measure can be useful in situations where negative examples are difficult to obtain or define. This seems to be the case for the IRES motifs. At this point in time, it is not clear how many gene transcripts contain an IRES, and there is no available data set of gene transcripts that have been shown experimentally to not contain the motif. As such, one cannot simply use transcripts that have not been shown to contain an IRES as negative examples. Therefore, it is postulated that the information content of the search results can be effectively used as a ranking scheme for the purposes of this discussion.

1.2.4 Scope of Work

The amount of work involved in the development of an all-encompassing RNA secondary structure inference methodology is significant. As such, the scope of this thesis is constricted to include the determination of a motif assembly and search methodology. Furthermore, the solution space is constricted such that the exhaustive enumeration of all of the possible motifs is a computational feasible task. The determination of an effective scoring scheme as well as the elucidation of optimal parameters for the algorithm is beyond the scope of this thesis.

1.2.5 Thesis Organization

Section 3 of the thesis details the underlying suffix tree and suffix array data structures and associated algorithms used in the subsequent RNA secondary structure inference methodologies. Section 3 describes the methodologies used in determining and enumerating the secondary structure motifs from a single seed sequence. Section 4 details the mechanisms involved in using the enumerated secondary structure motifs from the seed sequence to infer consensus secondary structures from the remaining sequences. Section 5 describes the methodologies and data used to test the implemented system, and summarizes the results of these tests.

2 Suffix Trees and Suffix Arrays

The properties, construction, and usage of suffix trees, and by extension, suffix arrays, are discussed. The application of suffix trees and suffix arrays towards the creation of the solution space as well as searching the data sequences is elucidated in the subsequent sections.

2.1 Suffix Trees

A suffix tree is a conceptual data structure that provides efficient access to all of its constituent substrings. They can be constructed, represented, and manipulated in both linear time and in linear space. The first linear-time suffix tree construction algorithm was detailed by Weiner[19] in 1973. Shortly thereafter, McCreight produced a more space efficient algorithm[20] that is still one of the primary suffix tree construction algorithms in use today. Ukkonen developed a different linear-time algorithm that builds upon McCreight's solution, but detailed with greater clarity and efficacy[14].

A straightforward application of suffix trees that reveals their versatility is the substring problem where given a search string S of length n, one is required to determine whether or not S is contained with in a source text T of length m. Once the source text T is preprocessed into a suffix tree in time $O(m)$ and space $O(m)$, utilizing suffix tree search methodologies, the presence of the string S in T can be determined in time $O(n)$ and

independent of m[14]. The algorithm to perform this simple search will be detailed later in this section.

2.1.1 Suffix Tree Description

As briefly described above, a suffix tree is an efficiently stored data structure that allows for temporally efficient access to its substrings. More specifically, a suffix tree of an n-character string S is a rooted, directional tree with precisely n leaves. Each node in the tree has at least two child branches. Each branch in the tree has an associated label that consists of j characters, where j quantifies the length of the branch. The character depth at any node in the tree consists of the sum of the lengths of all of the branches preceding that node in a path from the root of the tree. Furthermore, the character depth at any point within a branch in the tree consists of the character depth at the parent node, plus the number of characters to the specified point within the branch. Each branch has a label consists of a concatenation of all of the characters represented by the branch. Each child branch emanating from a node has an edge label that consists of the first character in the branch label. No two child branches can ever have the same edge label. The path label for any node in the suffix tree consists of the concatenation of all of the branch labels on the path from the root to the node in question. A suffix tree for S with a length of n characters has n leaves, and no more than $n-1$ internal nodes, $2n-1$ total nodes, and $2n-2$ branches[21] and thus occupies $O(n)$ space.

Given a string S = abcbcde of length 7, note that there are exactly 7 suffixes: abcbcde, bcbcde, cbcde, bcde, cde, de, and e. An important property of suffix trees is that each leaf in the suffix tree uniquely identifies a suffix of the constituent string. Figure 2.1 illustrates a suffix tree for the string S = abcbcde.

Figure 2.1 - Suffix tree for the string S = abcbcde.

2.1.2 Suffix Tree Formalisms

Given a string S of length *n*, or, S[0:n-1], let the individual suffixes be labeled S_0, S_1, .. S_{n-1}. The suffix S_i begins at the ith position in S. That is to say, suffix S_i is given by the substring of S specified by S[i i+1 i+2 … n-1]. As mentioned previously, each leaf node of the suffix tree has a path label of a suffix of the string S. Let each of the leaf nodes for the suffixes S_i be represented by *i*. Implicit in the suffix tree of Figure 2.1 is a unique terminal symbol at the end of each leaf branch. Without a unique terminal symbol, it is possible for a suffix tree to contain a suffix that does not end at a leaf. That occurrence is

the result of a particular suffix matching a prefix of another suffix in S. Thus, prior to construction a suffix tree, a given string S is concatenated with a terminal symbol that does not occur in S. For the remainder of this document, this terminal symbol will be represented by $ unless otherwise specified.

2.1.3 Simplified Suffix Tree Construction

In order to clearly illustrate the conceptual simplicity of suffix tree usage, it is useful to detail here a simplified suffix tree construction algorithm, as adapted from Gusfield[14]. Given the string S[0..n-1], first insert this entire string into the empty tree. This constitutes the first branch in the tree. Note that this branch represents the suffix S_0 and has a leaf node label of 0. Next, the suffix S_1 is inserted into the tree under construction. S_1 is compared character by character against the only branch currently in the tree, S_0, until the location of the last match is found. At this position a node is then created and the remaining unmatched characters of S_1 are split off into their own branch. At the end of this branch is of course a leaf node with the label 1. The remaining suffixes S_2 .. S_{n-1} are subsequently inserted into the tree in a similar manner. This simple suffix tree construction algorithm takes $O(n^2)$ time and occupies $O(n)$ space[14], where n is the length of the source string.

2.1.4 Classical Applications of Suffix Trees

Suffix trees have a wide variety of applications and can be used to efficiently solve a number of problems. A few simple approaches are presented here in order to provide a framework for the eventual application of suffix trees to the previously outlined problem space.

2.1.4.1 String Matching

Given a set of text T of length *n* and a search string P of length *m*, a general suffix tree problem is to find all of the occurrences of P in the text T. The solution proceeds as follows. First, preprocess the text T into a suffix tree. Then, starting at the root of the suffix tree of T, attempt to traverse a path dictated by the characters in the pattern P. More specifically, starting at the first character in P, given by P_0, attempt to find an edge label that has the same initial character as P_0. Since no two child branches of a node have the same edge label, there will either be one unique path or no paths at all. If there are no paths, then the search is complete and there are no occurrences of P in T. Otherwise, continue to match the characters in P with branch labels and edge labels in the suffix tree T. If either a leaf node in T is reached or no further matches are possible and P has not yet been completely processed, there are no occurrences of P in T. Otherwise, P will match up to a certain location in the suffix tree. This location will either be a node or within an edge in the tree. All of the leaves that are below this point in the tree (i.e. leaf child nodes of the next node down the tree) represent matches of the pattern P in T.

Furthermore, the leaf node labels $i_1 \ldots i_k$ of these leaves indicate the *k* positions that P matches in T. The search takes *O(m)* time to match the pattern P as it is of length *m*. The sub-tree containing the matches contains *k* internal nodes and *k* leaves, and can be traversed in *O(k)* time, independent of the number of characters. Enumerating the leaf nodes that represent the match locations in T thus takes *O(k)* amount of time where *k* is specified to be the total number of matches of P in T. Thus, this simple exact string matching search requires *O(m+k)* time to process. Since the suffix tree requires only *O(n)* space[14], the search has a linear spatial complexity relative to the search space. Note that the linear spatial complexity is possible as the edges in the suffix tree are compressed and simply contain a reference to the respective start and end positions of the edge label.

String matching utilizing wild cards requires only a slight modification to the above algorithm. In this case, the wild card is simply treated like another character in the search pattern P. Every time a wild card is encountered in the pattern P during the traversal of the tree, it is considered a match and the search index in P and the path in the suffix tree T are both advanced. Note that if a wild card is encountered at a branch node in the suffix tree, all of the child paths have to be processed. Note that this incurs a significant performance penalty; there are superior methods for performing string matching searches with wild cards, however the reasoning behind this methodology will become clear further in this document.

Additionally, one can readily allow an arbitrary maximum k number of mismatches in the search. First, set the mismatch counter m to 0. Every time a mismatch is encountered during the traversal of the suffix tree and the search pattern, m is incremented. If m exceeds the maximum mismatch value of k, the search is declared to have failed. Otherwise, both the search index in P and the path in the suffix tree are advanced. Again, as above, if the mismatch occurs at a branch node, all of the child paths have to be processed. Similar to above, by itself, the k-mismatch search has a superior solution involving suffix trees; also similar to above, the reasoning behind this methodology will become clear further on.

2.1.4.2 Generalized Suffix Trees

In certain situations, particularly with regards to computational biology, a suffix tree needs to be constructed given a set of k separate strings. The aforementioned Weiner, Ukkonen, and McCreight suffix tree algorithms specifically deal with creating a suffix tree from a single source string (i.e. the text T). In order to modify the suffix tree to accept k distinct texts T_i while maintaining an ability to differentiate between substrings of each T_i, a slight modification has to be made. One method is to concatenate each sequence with a unique terminal symbol[22], concatenate the resulting sequences together, and then proceed with the suffix tree construction process. Recall that for an individual source text T to be processed into a suffix tree, it must be concatenated with a unique terminal symbol that does not occur anywhere in T. For a set of k texts T_i, each T_i must be concatenated with a unique terminal symbol $\$_i$, where for every i in (0…k-1), $\$_i \neq \$_j$ if

$i \neq j$. The suffix tree is then constructed as desired and has the same properties as suffix trees based on a single source text.

One drawback is that below the first terminal symbol encountered in a branch, there exists the concatenation of the remainder of the string. This is a byproduct of the construction process, since the source text is comprised of the concatenation of all of the sequences. By delineating the existence of the first terminal symbol as the leaf of the branch, problems arising from the presence of these synthetic suffixes are avoided. A suffix tree comprised of multiple, differentiated source texts is known as a generalized suffix tree. Note that the number of leaves in the generalized suffix tree is equivalent to the total number of suffixes in all of the source texts. Each leaf is uniquely identified with a leaf number identifying the position of the suffix in its source text, as well as the unique terminal symbol that identifies the originating text T_i.

Searching for a particular string S in a generalized suffix tree proceeds in a similar fashion as to the single source text suffix tree. For each character in S, a path in the generalized suffix tree is attempted. If no path exists, then S does not exist in any of the source texts T_i. Otherwise, S does exist in at least one of the texts T_i. Furthermore, all of the leaves in the sub-tree below the path determined by S yield all of the matches of S in the source texts T_i. More precisely, each leaf node uniquely identifies the position j in text T_i where S is found. Note that this search must terminate whenever one of the terminal symbols $\$_i$ is encountered. The terminal symbol denotes a leaf, even though

there may be text below the terminal symbol due to the concatenated strings. This remaining text is not relevant to the suffix tree or its properties. Searching for a particular string S of length n in a generalized suffix tree takes $O(n)$ time to determine the path of the suffix. An additional $O(j)$ amount of time is required for each of the j total matches across the k source texts, yielding a total search time of $O(n + j)$[14]. This is the same order of time required for searching the suffix tree built from a single text T.

Gusfield presented[14] a modification of the standard Ukkonen suffix tree construction methodology to facilitate generalized suffix tree generation without requiring the concatenation of the source texts T_i. Very simply, the suffix tree is first constructed for text T_0. For the next text T_1, start at the root of the tree previously constructed. Attempt to match T_1 to a path in the suffix tree until a mismatch occurs. At this point, the Ukkonen suffix tree construction algorithm is applied to the remainder of T_1. In this form of construction, each leaf must contain the identity and position information regarding the suffixes of the various texts T_i that it terminates.

2.1.4.3 Lowest Common Ancestor

The combination of suffix trees and the lowest common ancestor algorithm (LCA) is a useful application. The lowest common ancestor of any two nodes i and j is generally defined by the node that is furthest from the root of any rooted tree whose sub-trees contain both the nodes i and j. Figure 2.2 illustrates the lowest common ancestor and its path from the root for the leaf nodes that describe the suffixes bcbcde and bcde. Note

that the lowest common answer search yields the longest common prefix of any two suffixes in the source text. This is an important result and will be expanded upon in the following section.

Figure 2.2 - Lowest common ancestor path and node (bolded) of the leaf nodes for the suffixes bcbcde and bcde.

Gusfield and Lewis[23] detail a methodology to enable suffix trees to perform lowest common ancestor searches in constant time. The methodology involves the application of a linear pre-processing algorithm to the suffix tree. This pre-processing allows a conceptual binary tree to be mapped to the suffix tree. For any two specified nodes, the lowest common ancestor is derived through a series of bit-level computations which require constant time to execute[14]. The specific details of this algorithm can be found in Gusfield[14] and are beyond the scope of this thesis. In and of themselves, constant time lowest common ancestor computations do not seem to have useful, direct applications.

However, such functionality lends itself indirectly in the application of a plethora of useful string-based algorithms.

2.1.4.4 Longest Common Extension

The longest common extension application of suffix trees provides a foundation for the application of suffix trees to string-based algorithms. Given two strings S_0 and S_1, and two indices i and j that are relative to S_0 and S_1 respectively. The longest common extension of S_0 and S_1 and i and j, yields the length of the longest common prefix of the two strings starting at positions i and j respectively. In other words, the longest common extension determines the length of the longest possible prefix that is common to the substrings S_0 starting at i and S_1 starting at j.

Given any i and j, the longest common extension of two strings is computed in constant time. From the two strings, a generalized suffix tree is constructed and the necessary pre-processing is performed in order to enable lowest common ancestor queries. The lowest common ancestor query is performed on the two leaf nodes i and j; the character depth of the resultant lowest common ancestor node yields the longest common extension[14]. Referring to Figure 2.2, the longest common extension is 2, which is the depth of the lowest common ancestor node.

2.1.4.5 Palindrome Detection

Relevant to the problem space is the detection of biological palindromes within a given sequence. The problem of detecting lexical palindromes will be first examined. An example of a lexical palindrome is given by the string S = abccbade. The pivot point of this palindrome is considered to be the location between the two 'c' characters. This string has a substring abccba that is a palindrome as it reads the same both forwards and backwards. This substring is referred to as a maximal palindrome as it defines the longest possible palindrome. Sub-maximal palindromes would be given by the substrings bccb and cc. Note that the example given is that of a palindrome that has an even length. A string S' = abcdcbade has an odd length maximal palindrome of abcdcba. In this case, the character 'd' is the pivot point of the palindrome. The radius of the palindrome is considered to be the length between the pivot point of the palindrome and its start or end point. For the purposes of this discussion, maximal palindromes are examined.

Given a source text T of length $n - 1$, in order to detect the presence of all of the maximal palindromes, the text T must be first processed into a suffix tree along with the pre-processing required for longest common extension queries. However, prior to being processed into a suffix tree, the text T is concatenated with the reverse string of itself, with each portion terminated by an appropriate terminal symbol. That is, the concatenated string $T \$_0 T^R \$_1$ of length $2n$ is processed into a suffix tree, where T^R represents the reverse string of T. Henceforth, the terminal symbols will be considered implicit. Assuming that the string TT^R starts at position 0, the palindrome search begins

at position $i = 1$, where i represents the forward index (i.e. in the forward string), and $j = 2n - i - 1$. Then, for every i from 1 to n-1, the longest common extension query is performed for the pair i and j. Every non-zero result yields a palindrome centred at position i and all maximal length palindromes can be found in $O(n)$ time[14].

Note that the above is applicable for even length palindromes. To handle odd length palindromes as well, a slight modification is required. For every i, two longest common extension queries are required – one for j and another for $j + 1$. The search for all maximal palindromes, odd or even, thus takes $O(2n)$, which remains linear. The latter simply skips over the pivot point of the odd length palindrome. In order to handle gapped palindromes such as S = abcdfgcbade, where the palindrome abc…cba has a gap of dfg that does not satisfy the palindrome criteria, a slight modification has to be made to the odd-length palindrome search method. Rather than performing two longest common extension queries for each position i, g queries are invoked, from $j = 2n - i - 1$ to $j = 2n - i - 1 + g - 1$. The value g is the bounded size of the gap. The problem of finding all gapped palindromes with an unbounded gap length is actually a more complex problem of finding all inverted repeats[14], and is beyond the scope of this discussion.

A k-mismatch palindrome is a substring that reads the same forwards and backwards, if k non-matching characters in the palindrome are permitted. For example, the substring abcfeedcba is a 1-mismatch palindrome, where the 'f' character constitutes the mismatch. In order to accommodate k-mismatch palindromes, a minor modification has to be made

to the algorithm discussed above. For every *i* and *j* as bounded above, each value is offset by a value *m*, which is initially set at 0. Starting at an arbitrary p^{th} iteration, the longest common extension, *l*, is computed for *i* and *j*, with m set to 0. If $j + l + m$ is equal to *n*, then a palindrome with a mismatch *m* occurs starting at *i* with a gap *g*. This iteration is then concluded. Otherwise, if $m < k$, *m* is incremented, *j* is updated to $j + l$, and the next iteration is executed. The time required to find all maximal, *k*-mismatch, *g*-gapped palindromes in a text T with length *n* is thus *O(nkg)*, where *k* and *g* are fixed constants. Thus, the palindrome search maintains its linear temporal bound.

In order to adapt this technique to detect biological palindromes, instead of concatenating the reverse string T^R to T, one concatenates the complemented reverse string $C(T^R)$ to T. For example, if the target string is T = ACTGAACAGT, the complemented reverse string is given by $C(T^R)$ = ACTGTTCAGT. Then, the algorithm is performed as detailed above and detects all maximal, *k*-mismatch, *g*-gapped biological palindromes.

2.1.5 Suffix Tree Construction

There are two linear-time suffix tree algorithms that are widely known and used. They are the Ukkonen and McCreight algorithms briefly mentioned previously. Both algorithms require linear time to execute. The major difference between the two algorithms lies in the fact that the Ukkonen algorithm has an on-line property. This property is such that at each stage of its construction, the algorithm simply generates a valid suffix tree that contains one additional character of the prefix of the source text. On

the other hand, only upon completion does the McCreight algorithm generate a valid suffix tree. Although the McCreight algorithm differs from Ukkonen in that respect, the Ukkonen algorithm can be converted into the McCreight algorithm via a series of transformations, as detailed by Giegerich and Kurtz[24]. While the specification of these suffix tree construction algorithms is beyond the scope of this thesis, it is useful to elaborate upon their spatial characteristics.

Giegerich and Kurtz's treatise on suffix tree construction algorithms[24] determined that with the application of a slight optimization, the McCreight algorithm had a minor efficiency improvement over the Ukkonen algorithm. Furthermore, Kurtz developed a compact data structure to represent suffix tree implementations that took advantage of characteristics inherent to the McCreight algorithm. This data structure is the most compact suffix tree representation known to be available that does not sacrifice any of the suffix tree characteristics[25]. From this body of work, it was initially thought that this compact suffix tree representation would be suitable to store and access the large amount of sequence data to be searched.

Kurtz's representation takes advantage of redundancies inherent in the suffix tree structure. By exploiting the relationships between branching nodes in the suffix tree, Kurtz's data structure partitions the branching nodes into two sets – large nodes and small nodes. The small and large descriptor characterizes the amount of data required for each type of node. Within the suffix tree, Kurtz defines chains of small nodes that are

terminated by large nodes. By doing so, certain information that is infrequently accessed by the small nodes can be stored in the large nodes. When a small node access requires this information, it can traverse the chain of small nodes to reach the large node and retrieve the requisite data. In addition, Kurtz's design packs data as tightly as possible into memory words; bitwise masking and shifting are required to read and write the data from and to the various nodes. Kurtz's scheme costs 8 bytes per small node and 16 bytes per large node; it allows the addressing of up to 2^{27} suffix tree nodes. These calculations are based on maximizing the usage for a computer system with 32-bit memory words. In the worst case scenario, Kurtz's scheme requires $20n$ bytes of storage, where n is the size of the input text. This scheme bounds the size n of the input text to 2^{27}.

The computing system available for the purposes of solving the problem space utilizes 64-bit memory words with a significant amount of RAM. In order to maximize the usage of available system resources, for the suffix tree implementation Kurtz's data structure is modified accordingly. In order to allow for a bottom up traversal of the suffix tree, another modification is made to Kurtz's data structure such that additional information is added to child nodes in order to point to their direct parent node. The resulting packed data structures require 16 bytes per small node and 24 bytes per large node. In the worst case, the adapted storage scheme requires $28n$ bytes of storage, where n is the size of the input text. This scheme allows for an input text size n of up to 2^{32}.

Implementation of the original Ukkonen or McCreight algorithms is a non-trivial task. Kurtz's optimization of the McCreight algorithm coupled with the space optimizations outlined above further compounded the complexity of the algorithms. A less complex, less space consuming data structure is desired. A recent development in the study of suffix arrays provides the alternate desired solution.

2.2 Suffix Arrays

In 1989, a relatively simple data structure that seemed to provide many of the benefits of suffix trees was introduced by Manber and Myers[26]. Essentially, a suffix array consists of a lexically sorted list of all of the suffixes of a source text T with length n, consuming $O(n)$ space[14]. Note that the suffix array is simply an ordered list of the positions of the suffixes in the source text; the array does not store the suffixes themselves. As such, the suffix array occupies a linear amount of space. More practically, a suffix array implementation requires significantly less space than a suffix tree based on the same source text, as well as occupying a vastly simplified data structure. Furthermore, it has been shown that all bottom-up[27] and top-down[28] approaches to manipulating suffix trees can be replicated on suffix arrays with the same $O(n)$ time complexity, when augmented with additional information requiring $O(n)$ space and time. These reasons sufficiently motivate a desire to develop techniques to utilize suffix arrays for solving the problem at hand.

A suffix array can be readily created in $O(n)$ time by simply traversing the suffix tree in a depth-first, lexically ordered traversal and adding the leaf node indices to the suffix array as they are encountered[14]. This method, however, requires the suffix tree to be previously constructed. Direct construction of suffix arrays had previously required $O(n \log n)$ to be completed[14 26]. Such a construction time is certainly not desirable. Issues and advancements regarding suffix array construction will be revisited following a brief overview.

2.2.1 Overview

Given a string S = abcbcde, the resulting suffix tree with leaf numbers is illustrated below in Figure 2.3.

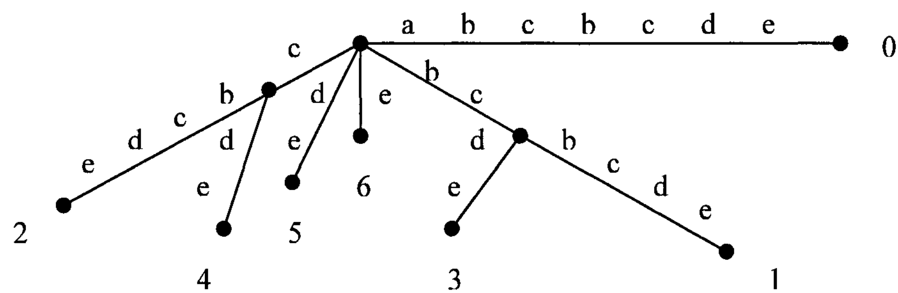


Figure 2.3 - Suffix tree for string S = abcbcde, with leaf numbers indicting the position of the suffix in S.

For illustrative purposes, the suffixes in the order of their positions in the original string are listed below in Figure 2.4.

S_0 = abcbcde

S_1 = bcbcde

S_2 = cbcde

S_3 = bcde

S_4 = cde

S_5 = de

S_6 = e

Figure 2.4 - Suffixes for string S = abcbcde.

The suffix array of S is simply an array of the integers 0-6, ordered in the lexical order of its constituent suffixes, as shown below in Figure 2.5. Referring to Figure 2.3, a depth first, lexically ordered traversal will first follow the path to leaf node 0. This is the first suffix in the suffix array. The next depth first, lexically ordered traversal will choose the path with the starting edge label of 'b' from the root. At the next node, it will choose, again, the path with the starting edge label of 'b'. This path yields the leaf node 1, the second suffix in the suffix array. The traversal continues to completion, yielding the suffix array as shown below in Figure 2.5.

Suffix array: [0,1,3,2,4,5,6]

Suffixes in lexical order:

S_0 = abcbcde

S_1 = bcbcde

S_3 = bcde

S_2 = cbcde

S_4 = cde

S_5 = de

S_6 = e

Figure 2.5 - Suffix array for string S = abcbcde.

As a comparison, recall that the adaptation of Kurtz's reduced size suffix tree to accommodate a sequence length 2^{32} resulted in the usage of *28n* bytes of storage. For a suffix array to store the same sequence, approximately *4n* bytes of storage are required. Further in this discussion, it is indicated that a few other data structures are needed. However, the total amount of space required is still significantly less than that required for Kurtz's reduced size suffix tree.

The construction of the suffix array using the depth first traversal of the suffix tree is conceptually elegant and simplistic. However, it is not a pragmatic solution as it still

requires the creation of a suffix tree, a task that is desired to be bypassed. A more efficient and less complex solution is required.

2.2.2 Suffix Array Construction

At approximately the same time in 2003, three papers were published that detailed a linear time construction of suffix arrays. The algorithm developed by Karkkainen and Sanders[29] is by far the simplest out of the three; a terse sample implementation is described in their paper and requires only 50 lines of C++ code. The Karkkainen and Sanders algorithm is based upon dividing the suffixes of the text into distinct sets, sorting them, and merging the results[29]. Ko and Aluru present an algorithm that is based upon a linear time sort of the suffixes of a string[30]. Finally, Kim et al utilize the same divide and conquer approach as the other two algorithms; their algorithm creates two disjoint suffix arrays each from a subset of the suffixes, and then these suffix arrays are merged[31] . All of these algorithms are based on the utilization of integer alphabets, which is completely coherent with the problem space. Given the existence of linear time suffix array construction algorithms along with the demonstrated application of top down and bottom up suffix tree traversals on suffix arrays, it is now becoming evident that suffix trees are more and more applicable only in a historical and conceptual context[32].

Out of the three existing linear time suffix array construction algorithms, the Karkkainen and Sanders algorithm is by far the simplest. The implementation of the Ko and Aluru algorithm presented a greater degree of complexity for an uncertain benefit. The

algorithm of Kim et al, while having a linear time complexity, utilizes a complicated merging algorithm. As such, the Karkkainen and Sanders algorithm, henceforth referred to as the skew algorithm[29], is selected as the desired suffix array construction algorithm.

2.2.2.1 Suffix Array Skew Algorithm

The skew algorithm by Karkkainen and Sanders [29] is straightforward and is presented briefly. The skew algorithm works by partitioning the text into two parts, generating two suffix arrays, and then merging them into the final result.

1. The suffixes of the source string S, given by S_i are first partitioned into two distinct sets: one set consisting of all of the suffixes i, such that $i \bmod 3 \neq 0$, and the remaining suffixes such that $i \bmod 3 = 0$. This step serves to partition the set of suffixes into one part comprising of 2/3 of the suffixes, and the other part comprising the remaining 1/3.
2. The suffixes that belong to the $i \bmod 3 \neq 0$ class are sorted using the radix sort for the respective triples (s[i], s[i+1], s[i+2]) of the original string. These suffixes are ranked accordingly. The result of this sort is then scanned to determine the number of unique triples. If the number of unique triples is the same as the number of suffixes in the $i \bmod 3 \neq 0$ class, the result is a suffix array for this class. Otherwise, the suffix array creation routine is recursively applied to this subset.
3. The suffixes that belong to the $i \bmod 3 = 0$ class are then sorted using the radix sort against the suffix array generated in the previous step. The radix sort is

performed for the pair (s[i], rank of S_{i+1}), where *i* mod 3 = 0, and the rank of S_{i+1} has been obtained in step 2. This yields a suffix array for the suffixes that belong to the *i* mod 3 = 0 class.

4. Finally, the two suffix arrays are merged by comparing the lexical ordering of the suffixes in the two suffix arrays.

The execution time of this algorithm is given by the recurrence relation T(n) = *O(n)* + T(⌈2n/3⌉) which yields *O(n)* run time[29 38].

2.2.3 Suffix Array Applications

Both the elucidation of the solution space as well as the execution of the search algorithm rely upon applications of the text searching capabilities of suffix trees. In order to successfully use suffix arrays in place of suffix trees to solve the problem space, the concepts described in section 2.1.4 need to be successfully applied to suffix arrays.

2.2.3.1 Additional Data Structures

Prior to detailing the methodology for traversing, searching, and processing suffix arrays to achieve equivalent results as suffix trees, additional data structures must be created. Suffix arrays do not store the topological information that is inherent to suffix trees. Unlike suffix trees, there are no physically present nodes in a suffix array. Therefore, in order to traverse and manipulate suffix arrays in a similar manner as to suffix trees, but

without the space complexity, a limited number of data structures are required to augment the available information.

2.2.3.1.1 Adjacent Longest Common Prefix

A commonly mentioned augmentative data structure for suffix arrays is the adjacent longest common prefix array[26]. This array stores the length of the longest common prefix (LCP) between any two adjacent suffixes in the suffix array. The adjacent LCP array has the same number of elements as the suffix array. Note that the value at the 0th element is undefined. The value of the adjacent LCP at some element x is the length longest common prefix between suffixes $x-1$ and x in the suffix array. Given any two suffixes, the length of the longest common prefix between the two suffixes will be given by the minimum of the adjacent LCP's for consecutive pairs of suffixes between them in the suffix array[33]. More specifically, given an adjacent LCP array adjLCP, a suffix array SA, and two suffixes in SA labeled i and j, we have the following definition for the length of the longest common prefix between any two suffixes i and j:

Equation 2.1 - $LCP(i, j) = \min_{i < x \leq j} \{ adjLCP[SA[x]] \}$

This is an important definition that will need to be revisited. The adjacent LCP array occupies n integer locations, and is created through a single pass (n iterations) of the suffix array. Table 2.1 details the relationship between the suffix array, the adjacent LCP array, and the various suffixes for the example sequence.

i	SA	adjLCP	S[i..n-1]
0	0	-	abcbcde
1	1	0	bcbcde
2	3	2	bcde
3	2	0	cbcde
4	4	1	cde
5	5	0	de
6	6	0	e

Table 2.1- Relationship between suffixes and adjacent LCP values.

2.2.3.1.2 LCP-Interval Tree

If the adjacent longest common prefix values in Table 2.1 are compared to the suffix tree depicted again in Figure 2.6, one may make some observations. For elements that have adjacent longest common prefix values of 0, the corresponding suffixes are rooted at the root of the suffix tree in Figure 2.6. Furthermore, the adjacent LCP value for two adjacent suffixes in the suffix array indicates the length of their longest common prefix.

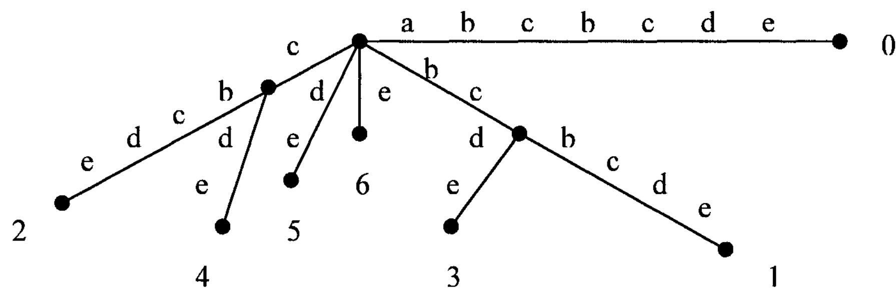


Figure 2.6 - Suffix tree for suffix S = abcbcde.

In fact, the entire suffix tree can be represented implicitly with the concept of LCP intervals, as introduced by Abouelhoda et al[28]. What follows is adapted from Abouelhoda et al[28]. The LCP interval is defined as follows:

An interval (i...j) , $0 \leq i < j \leq n - 1$, is an LCP interval of LCP-value l if the following hold:

adjLCP[i] < l

adjLCP[k] ≥ l for all k with $i + 1 \leq k \leq j$

adjLCP[k] = l for at least one k with $i + 1 \leq k \leq j$

adjLCP[j + 1] < l

Let an LCP interval (i…j) of value l be referred to as an l-interval. Another interval of value m (referred to as an m-interval) (a…b) is considered to be embedded within an l-interval (i…j) if $i \leq a < b \leq j$. Furthermore, if an m-interval is embedded within the l-interval, and there is no interval embedded within the l-interval that embeds the m-interval, the m-interval is considered the child of the l-interval. The parent-child interval relationship established by the existence of the nested intervals is similar to the parent-child relationship of suffix trees. In fact, the set of LCP intervals of the adjacent LCP array constitutes a conceptual LCP interval tree. Furthermore, this conceptual tree maps onto a suffix tree, where the leaves are comprised of LCP intervals where the start and end indices of the intervals are identical (i.e. interval (i…j) is a leaf interval if $i = j$)[28].

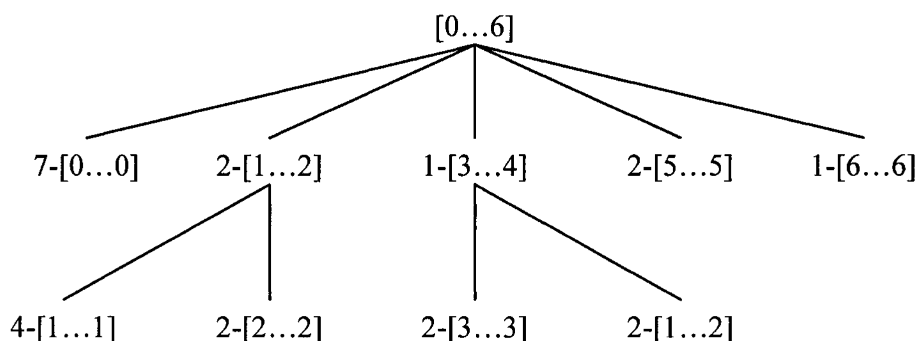


Figure 2.7 - LCP interval tree for the adjacent LCP array of the string S = abcbcde.

The LCP interval tree for the adjacent LCP array of Table 2.1 is depicted in Figure 2.7. The number preceding the interval indicates the interval value.

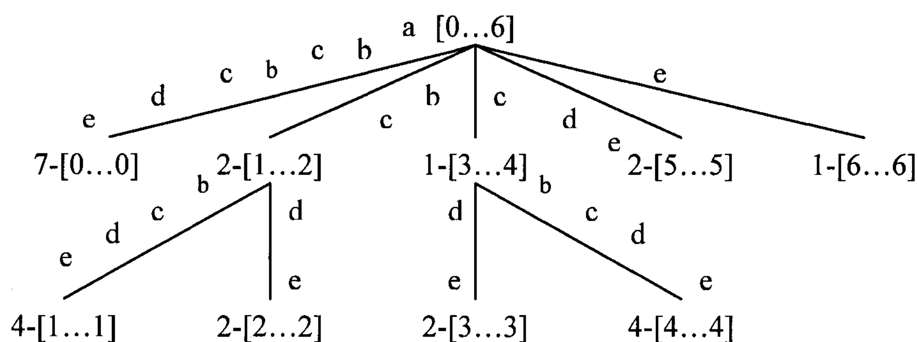


Figure 2.8 - LCP interval tree overlaid with the corresponding suffix tree.

Figure 2.8 shows the LCP interval tree for the adjacent LCP interval array of the string

S = abcbabcde. Overlaid on this figure is the suffix tree of Figure 2.6 for the same string. Note that the two correspond identically. Thus the suffix tree can be traversed by simply processing the virtual LCP interval tree[28], which, is simply an array of integers. Abouelhoda et al developed an additional data structure, the child table[28], that enumerates the parent-child relationships in the adjacent LCP interval array. The child table occupies *n* integer memory locations and is created through a single pass (*n* iterations) of the adjacent LCP array.

Given any LCP interval in the LCP interval tree, the child table enables the determination of all of child intervals in a linear amount of time with respect to the number of child intervals. With this capability, any top down suffix tree traversal can be simulated using a suffix array. By traversing the adjacent LCP array with a stack, the simulation of a bottom up traversal of a suffix tree is also possible[27]. For the purposes of this discussion, what remains to be shown is the computation of the lowest common ancestor, and consequently, the longest common extension, using suffix arrays rather than suffix trees.

2.2.3.1.3 Longest Common Extension Computation

In order to complete the foundation for the application of suffix arrays towards solving the problem space, Equation 2.1 must be revisited:

Equation 2.1 - $LCP(i, j) = \min_{i < x \leq j} \{ adjLCP[SA[x]] \}$

The result of this equation is that for any two suffixes in the suffix array, the length of the longest common prefix is the minimum of the adjacent LCP values between the two

suffixes in the suffix array. Furthermore, by knowing the length of the LCP(i,j), say *k*, the longest common prefix can be extracted from the original string via S[SA[i] SA[i]+1 ... SA[i]+k-1], where S represents the original string and SA represents the suffix array. Note that this prefix is identical to S[SA[j] SA[j]+1 ... SA[j]+k-1] since *k* represents the length of the longest common prefix between suffixes *i* and *j*. Thus, given the ability to compute LCP(i,j) in constant time, both the lowest common ancestor and the longest common prefix can be ascertained in a equivalent time and space complexity manner as suffix trees.

The problem of computing the minimum value between a range of values in an unsorted array is a well understood problem known as a range minimum query (RMQ). Bender and Farach-Colton detailed a constant time LCA solution for a tree[34]. The solution involves a linear amount of time in pre-processing time to create a set of tables. The LCA solution simply requires a fixed number of lookups to these tables, yielding a constant time complexity. In order to extend the LCA solution for use with general range minimum queries, an additional data structure has to be implemented.

The Bender and Farach-Colton LCA solution relies on pre-computing a set of lookup tables from a source array of integers where each adjacent integer differs from the other by one. In order to complete the algorithm, the adjacent LCP array needs to be processed into another array to yield the property that the array elements differ by 1. Bender and Farach-Colton detail using a Cartesian tree[34] for this pre-processing step. In order to

generate the required data, an Euler tour of the constructed tree is required. An Euler tour is simply a depth first traversal of the Cartesian tree that yields the sequence of nodes visited in succession. At each step during the Euler tour, the depth of the node from the root is recorded in a separate array, yielding the required data. Note that at each step, the depth of the node differs from the previous depth by one. The Cartesian tree, as described by Vuillemin[35], is described as a labeled, ordered, binary tree. However, a variant of the Cartesian tree, the treap, is implemented over the adjacent LCP array as a modification of Nilsson's application[36]. A treap utilizes the same data structure as a Cartesian tree, but incorporates some modifications to improve the likelihood that the tree is balanced. The correlation between the treap and the Cartesian tree is suggested by Seidel and Aragon[37] and the implementation of the treap by Nilsson is the least complex of the two. The implementation requires *O(n)* time and space complexity[34].

The implementation of the treap over the adjacent LCP array is quite short and efficient. Upon completion of the pre-processing of the resulting Euler tour of the treap to yield the lookup tables for the RMQ, the length of the longest common prefix between two suffixes in the suffix array is immediately available and can be completed in constant time. The total amount of processing involved utilizes an *O(n)* time and space complexity[34]. This enables constant time longest common extension queries, as required by the palindrome detection algorithm. Note that following the completion of the pre-processing for the RMQ tables, the treap can be discarded. It should also be noted that the Euler tour can also be performed through a conceptual top down traversal of the

interval tree. However, this result is not as elegant as in order to compute the longest common extension, the unique labeling of each conceptual interval in the interval tree is required, as well as additional post-processing.

2.2.3.2 Palindrome Detection

Now that the constant-time computation of the length of the longest common prefix of two suffixes (i,j) has been established, biological palindrome detection can be performed via a single pass of the suffix array. Similar to the suffix tree palindrome detection, a suffix array is created from the concatenation of the forward sequence as well as the reverse, complemented form of the sequence. Palindrome detection, including the k-mismatch and g-gap variants, is then performed in the same manner as previously discussed.

2.2.3.3 Generalized Suffix Arrays

In order to create a suffix array consisting of all of the data sequences that comprise the search space, a generalized suffix array, much like a generalized suffix tree, needs to be created. The construction of the generalized suffix array follows the same method used by the more simplistic generalized suffix tree creation in section 2.1.4.2. Each sequence required in the search space is first appended with a terminal unique to itself. The set of all sequences is then concatenated together, and the suffix array is created as previously described. If each sequence s_i requires sl_i characters, and if there is a total of n sequences, then the generalized suffix array requires $(\sum_{0 \leq i \leq n-1} sl_i) + n$ integers. In order to differentiate suffixes between each sequence, an n-length array is constructed to store the start and end positions of each sequence in the concatenated sequence. The penalty

for the generalized suffix array constructed here is the additional *n-1* terminal symbols over the single string case, plus an *n*-length endpoint array.

An alternative method, described by Aluru[38], of representing a generalized suffix array involves representing each suffix as an integer pair (i,j) that denotes suffix *i* from sequence s_j. If two suffixes from different strings are identical, they occupy consecutive positions in the suffix array. This method requires 2 * sl_i integers to store the suffix indices for a single sequence, since one integer is required to identify the sequence, and the other integer is required for the suffix. Also, there is only one terminal symbol required for this method. Assuming that each sequence s_i requires sl_i characters, the total space required for this representation is $2 * (\sum_{0 \leq i \leq n-1} sl_i) + 1$, which, in general, is inferior to the above result.

2.2.3.4 String Matching

By simulating the traversal of the suffix tree by traversing instead the interval tree as detailed in 2.2.3.1.2, the string matching techniques detailed in 2.1.4.1 can immediately be applied with the same computational complexity. Note that the generalized suffix array conceptually maintains linkages to the synthetic suffixes - every suffix is appended with the remaining sequence data, as the source text is comprised of all sequences concatenated together. However, by enforcing a rule that the searches terminate upon detection of a terminal node, problematic circumstances where strings are matched to a suffix of one sequence and a prefix of the subsequent sequence are avoided.

3 Enumeration of Secondary Structure Motifs for One Sequence

The problem of enumerating secondary structure motifs for one sequence can be partitioned into three separate sub-tasks: 1) All of the contiguous stems in the sequence, subject to certain criteria, are discovered. 2) These stems are treated as the atomic element of a motif and are assembled into all possible valid secondary structures, subject to certain criteria. The result of this second step is the enumeration of all possible secondary structure motifs for a single sequence. 3) Since the solution space resulting from the second step can be extremely large, it must be reduced. This reduction is performed via the selective application of certain criteria to eliminate undesired secondary structure motifs.

3.1 Finding and Representing Stems

The process of detecting all of the stems involves the detection of all biological palindromes within a given set of data. Recall the stem loop structure elucidated earlier. The stem of this structure is essentially a palindrome with a gap of width g, where g is undetermined. In Figure 3.1, a stem loop structure is shown with a gap of size 8 and a stem length of 6. The nucleotides belonging to the loop are italicized, and the nucleotides belonging to the stem are bolded. The remaining nucleotides are unmatched and are not considered part of the stem.

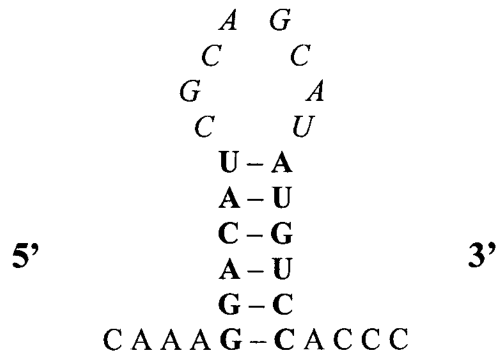


Figure 3.1 - Example of a stem-loop structure to be detected.

In order to detect all of the stems of a given source sequence, the entire sequence is concatenated with its reverse complement, and then processed into a suffix array. The adjacent LCP array is then computed over the suffix array. Following that, the necessary pre-processing is performed to allow the constant time range minimum query calculation, and as a result, the constant-time longest common extension query. The stem loop structure depicted in Figure 3.1, following concatenation with its reverse complement, would appear as a sequence as shown in Figure 3.2.

CAAAGGACAUCGCAGCAUAUGUCCACCC*GGGUGGACAUAUGCUGCGAUGUCCUUU*G

Figure 3.2 - Concatenation of forward and reverse complemented sequence.

In actual sequences, stem loop constructs are not necessarily as simple as depicted above. For instance, stem loop structures may have one or more unpaired nucleotides on either

side of the stem. Such a configuration is referred to as a bulge, and is shown below in Figure 3.3. The bulge in this case is a two nucleotide bulge consisting of C and U.

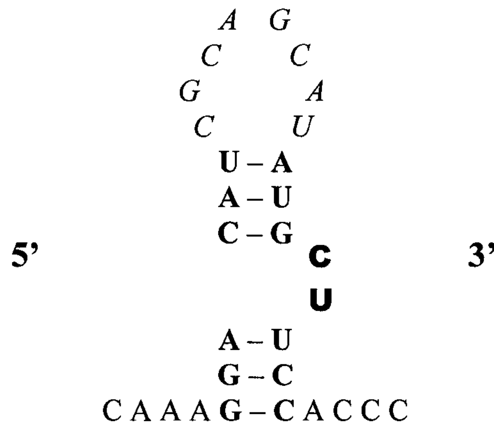


Figure 3.3 - Stem loop structure with a two nucleotide bulge.

When no mismatches are permitted, the stem detection algorithm treats this as two separate, contiguous stems. The first consists of the three G-C, G-C, and A-U base pairs, and the second consists of the C-G, A-U, and U-A base pairs. By increasing the granularity of the stem detection algorithm in this manner, the stem detection algorithm is exceptionally flexible. Note that stem segments in secondary structures do not necessarily occur as simple stem loop structures. In fact, it is quite possible to have secondary structures recursively composed of stems within stems[3]. Such a secondary structure is depicted in Figure 3.4.

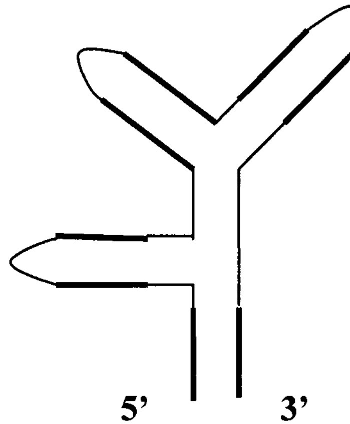


Figure 3.4 - Secondary structure comprised of recursively nested stem segments.

As this secondary structure does not constitute a stem loop structure in its entirety, the stem detection algorithm detects the individual contiguous stem segments independently. The bold lines indicate the contiguous stem segments in this secondary structure.

There are three important pieces of criteria required to process the stem detection: maximum gap size, minimum stem length, and the maximum number of mismatches allowed per stem. The maximum gap size entails the same restriction placed upon the palindrome detection algorithm specified in the previous section. This criteria limits the distance between the 5' side of the stem and the 3' side of the stem that will be detected. Note that in Figure 3.4, the "root" stem in the recursive construction requires a reasonably large gap size in order for it to be successfully detected. The minimum stem

length restricts the minimum radius for which a biological palindrome will be detected. Conversely, the minimum stem length refers to the minimum number of base pairs in the stem. In order for the stem in Figure 3.1 to be detected, a maximum gap size of at least 8 and a minimum stem length of at least 6 is required. The maximum number of mismatches permitted per stem refers to the maximum number of base pair mismatches in a stem and is directly related to the *k*-mismatch stem detection method previously discussed. The maximum number of mismatches is the value ascribed to *k*. In Figure 3.5, the stem depicted has one base pair mismatch between the C and U nucleotides. In order for this stem to be detected, the maximum number of mismatches has to be set to at least one.

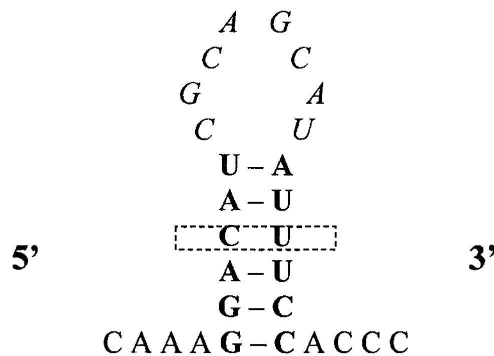


Figure 3.5 - Stem loop structure with a base pairing mismatch.

Now that the stem detection methodology has been established, what remains is to present a method for storing detected stems. Bouthinon and Soldano developed a formalism with which stem constructs can be represented[39]. However, this formalism

lacks the numerical information required for the constrained assembly of the motifs as will be discussed in the following section. The stem information is very simply stored in a structure consisting of the start and end indices of the 5' side and the 3' side of the stem in the source sequence. More lucidly, the stem information is stored in a 4-tuple (ls, le, rs, re), where ls is the start index of the 5' side, le is the end index of the 5' side, rs is the start index of the 3' side, and re is the end index of the 3' side. From this 4-tuple, the stem length and gap size of the stem structure can be readily computed. For any given 4-tuple of a stem segment i, given by (ls_i, le_i, rs_i, re_i), the following is always true:

$$ls_i < le_i < rs_i < re_i$$

This structure allows the motif to be constructed in a manner that reflects the search methodology to be performed against the data sequences.

3.2 Assembling Stems into Motifs

In the previous section, from the source sequence, all of the possible existing stem segments meeting specified criteria were discovered and enumerated. These stem segments form the atomic unit of a secondary structure motif. The stems must be assembled into motifs in a manner that can be used to search against the data sequences. By exhaustively generating all valid combinations of these atomic units, subject to certain criteria, a thorough list of potential motifs can be formed.

As depicted in Figure 3.4, through recursive nesting, secondary structures can be formed in a number of combinations limited only by the number of stem segments involved and

their respective topologies. More precisely, given two stem segments st_i and st_j identified by their respective 4-tuples, (ls_i, le_i, rs_i, re_i) and (ls_j, le_j, rs_j, re_j), there are three valid relationships between the two:

- st_i is adjacent to the left of st_j if $re_i < ls_i$
- st_i contains st_j if $le_i < lsj$ and $re_j < rs_i$ (in other words, st_j is nested within st_i)
- st_i is adjacent to the right of st_j if $ls_i > re_j$
- otherwise, there are no valid relationships between st_i and st_j

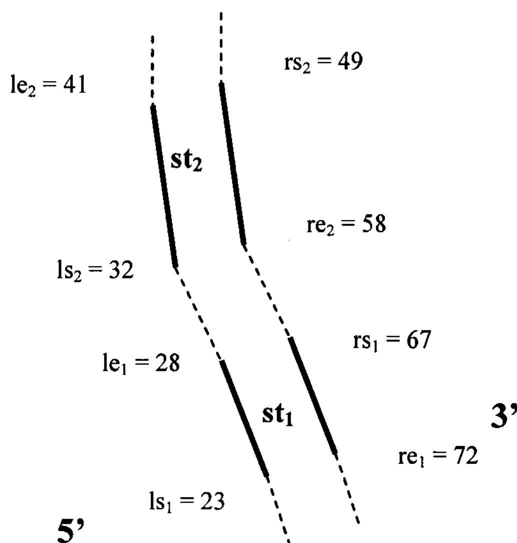


Figure 3.6 - Illustration of a stem segment nested within another stem segment.

Figure 3.6 depicts a pair of stem segments, with st_2 nested within st_1. Note that stem segments st_1 and st_2 do not have equal distance between their respective endpoints. On

the 5' side, the distance between the end of the 5' side of st_1 and the start of the 5' side of st_2 is 4 nucleotides. On the 3' side, the distance between the end of the 3' side of st_2 and the start of the 3' side of st_1 is 9 nucleotides. Also note that st_1 has a stem length of 5 nucleotides, and st_2 has a stem length of 9 nucleotides.

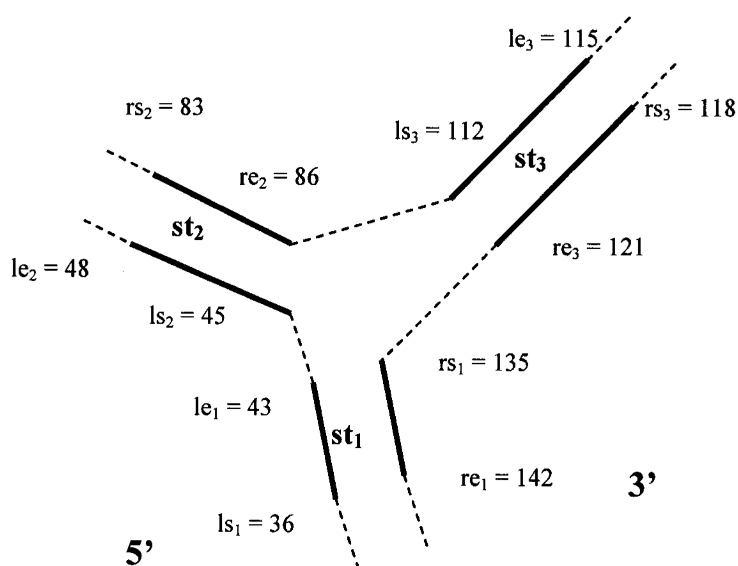


Figure 3.7 - Illustration of a more complex secondary structure consisting of three separate stem segments.

The secondary structure depicted in Figure 3.7 is more complex, and consists of three separate stem segments. Note that st_1 contains both st_2 and st_3, and that st_2 and st_3 are adjacent to one another. Also note that in terms of motif composition, the detection of stems st_1, st_2, and st_3 as described in section 3.1, would result in seven unique, valid motif configurations – st_2 nested within st_1, st_3 nested within st_1, st_2 adjacent to st_3, st_2 adjacent

to st_3 with both nested in st_1 as shown in Figure 3.7, and then configurations consisting of the individual stem segments.

To generate the complete set of motifs, the list of 4-tuples provided by the stem detection methodology is first sorted against the ls field in each. This list is sorted from least to greatest. In doing so, the motif assembly process can make a single pass through the stem 4-tuple list. The motif configurations are built by constructing what are essentially binary trees, were each node represents a stem segment. Starting at the root node, the root stem segment is represented. A left branch indicates the presence of a stem nested within the root stem. A right branch indicates the presence of a stem adjacent to the root stem. The remainder of the tree is built recursively. For each stem configuration, there exists a binary tree structure to represent it.

In order to illustrate the usage of the binary tree representation of the motifs, it is useful to introduce a slightly more complex secondary structure, as illustrated in Figure 3.8. This secondary structure has four individual stem segments. Given the combinatorial nature of the motif construction from individual stem segments, there are a total of 15 plausible, valid motif configurations. Recall that the secondary structure in Figure 3.7 is similar, but missing the stem segment st_4. Adding this one segment doubles the amount of possible configurations, plus itself, yielding 15 possible configurations.

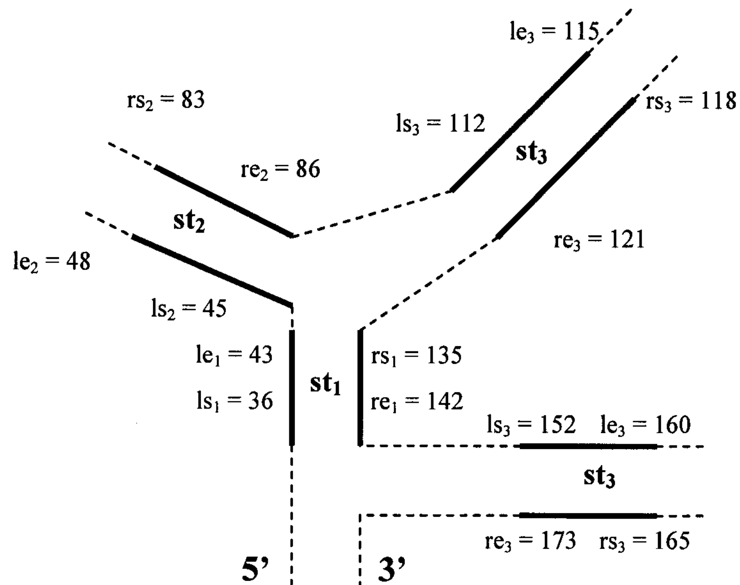


Figure 3.8 - Secondary structure consisting of four stems.

The binary tree representation of the motif construction consisting of all four stem segments in Figure 3.8 is shown below.

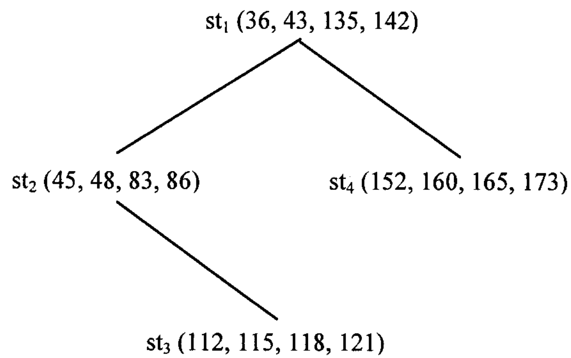


Figure 3.9 - Binary tree representation of a motif comprised of four stem segments.

The motif assembly process at some stage i is as follows. The ith stem is first instantiated into a motif and added to the motif list. Then, all possible combinations of the ith stem with all other motifs currently in the motif list are generated. The newly created motifs are then appended to the motif list. The process continues until the list of stems represented by 4-tuples is exhausted. The pseudocode describing this algorithm is below.

```
// General algorithm to assemble motifs from list of stems
// let i be an index in the list of sorted stems
// let j be an index in the list of motifs
initially set i = j = 0
for each stem i in list of stems

   add stem i to the end of the motif list
   for each motif j in the motif list (other than the last entry)

      if stem i can be nested in motif j
         combine stem i with motif j and add the new motif to the list

      else if stem i can be adjacent to motif j
         combine stem i with motif j and add the new motif to the list

      end if

   end for

end for
```

Note that all new motifs generated in each stage are searched against the target data set. Any motifs that fail to generate a match are discarded. The search process utilized in the iterative motif assembly is detailed in section 4.

3.3 Motif List Pruning

As mentioned briefly above, the process of generating all possible, valid configurations of stems results in a combinatorial number of solutions. The following derivation assumes the degenerate case scenario, in that all of the stems detected can be combined with each other into motifs. Note that the space of all possible motifs is bounded by the seed sequence; only motifs that satisfy the physical constraints of the seed sequence can be considered valid potential motifs. Potential motifs are entirely inferred from the seed sequence. This methodology is an analogue to that of Progol, as explored by Muggleton[40]. Assume that there are a total of n stems in the list of stem segments.

$$S = \{s_0, s_1, \ldots, s_{n-1}\}$$

The first stem, s_o, is instantiated into its own motif in the motif list:

$$M = \{m_0\}$$

The second stem, s_1, is also instantiated into its own motif and added to the motif list:

$$M = \{m_0, m_1\}$$

The second stem m_1 can also be combined with the first, m_0, to create another motif, which is added to the motif list:

$$M = \{m_0, m_1, \{m_0, m_1\}\}$$

Similarly, for the third stem segment, m_2, the list is extended as follows:

$$M = \{m_0, m_1, \{m_0, m_1\}, m_2, \{m_0, m_2\}, \{m_1, m_2\}, \{m_0, m_1, m_2\}\}$$

Observe that the addition of a single stem segment results in the motif list doubling in size, plus an additional motif for the new stem segment. During the addition of the ith stem segment, assume the motif list has a total number of motif constructs denoted by

$|M_{i-1}|$. For the construction of the motif list at stage *i*, stem s_i is first added to the list as its own motif. Then, stem s_i is combined with each element in the motif list M_{i-1} to create a new motif. These motifs are subsequently added to the motif list. As a result, the length of the motif list upon completion of stage *i* is given by:

$$|M_i| = 2 \cdot |M_{i-1}| + 1.$$

The length of the motif list upon completion of processing of a stem list of length *n* is given by the recursive relation:

$$|M_n| = 2 \cdot |M_{n-1}| + 1$$

or

$$|M_n| \approx O(2^n)$$

The $O(2^n)$ defines the spatial complexity of the motif list in the degenerate case scenario. Given that the number of stem segments *n* can be quite large depending on the length of the source sequence, the motif list requires selective pruning. The iterative motif assembly method described previously serves to reduce the number of motifs to be processed. Additional motif list pruning criteria include the following:

- Minimum stem length. The stem detection phase can be constrained to limit detection of stems to only those of length *l* or longer. In practice, *l* cannot be made too large as stem lengths in RNA secondary structures, given an alphabet size of 4 (A, C, G, U), average around 4 pairs in length[41]. Furthermore, the larger the stem length *l* is set to, the less likely that secondary structure motif candidates with stem segments containing bulges will be assembled. Conversely, stem

lengths cannot be set too short as many random and biologically irrelevant stems will be detected.

- Maximum number of mismatches. The greater flexibility permitted in the number of mismatches, the larger the number of stems will be detected. If the minimum stem length is set to a lower number, the maximum number of mismatches should be accordingly set to a lower value. Otherwise, the significance and accuracy of the detected stems will be adversely affected.
- Maximum gap size. The stem detection process can be configured to limit the maximum size of the gaps in the stems. Although this serves to reduce the number of motif configurations, it also reduces the likelihood that the motif construction process will generate a complex secondary structure. At the root of complex recursive secondary structures often lies a stem segment that encloses the other stem segments. As such, this stem segment necessarily has a large gap size. Overly constraining this variable will prevent such potentially biologically relevant structures from being generated as part of the solution space.
- Distance between stem segments. The motif construction can be limited in terms of allowable distances between its constituent stem segments. For instance the distance between the end of the 5' side of a stem segment and the start of the 5' side of a nested stem segment can be constrained to limit the number of configurations generated. In the IRES predicted secondary structure, the distance between disparate stem segments is typically four or fewer nucleotides[1].

- Total number of base pairs. The motif list can be pared down by providing the motif construction process with a delineation of a range of desired numbers of base pairs. This can significantly focus a particular motif search for a certain type of secondary structure, if so desired.
- Inheritance. As the potential motif structures are processed by the search (described in the following section), negative motif structure matches can be discarded. By extension, motif structures that inherited the negative motif structure will thus eventually be negatively matched. As such, any potential motif structures that inherit structure from a negatively matched motif can be discarded without consequence.

Note that there are no existing software systems or algorithms that automatically generate secondary structure motifs in the manner described above. As such, there are no competing algorithms, nor are there any established conventions to accomplish these tasks. Biologists must therefore devise experiments blindly. Given that the processing of large amounts of data can be time consuming (see Section 5), users of the proposed methodology must be provided with sufficient means to constrain the search space.

The list of potential motifs constructed comprises the solution space. In order to correctly infer possible secondary structure motifs, the solution space must be searched against the remaining data sequences.

4 Secondary Structure Motif Inference

Given the solution space of artificially assembled potential motifs, a determination must be made as to which of these potential motifs is most likely to be correct. This determination is made by inferring the secondary structure motifs by first creating generic versions of the structure motifs and searching them against a broader set of sequences. Positive results are further specialized in an exhaustive manner in order to ultimately converge to a set of inferred, consensus secondary structure motifs.

4.1 Secondary Structure Motifs: Ambiguity to Specificity

The composition of the solution space results in a large set of stem fragments assembled in various combinations. In order to allow the data to drive the composition of the consensus secondary structure motifs, the combinations of stem fragments are reduced from sets of contiguous, base paired nucleotides, into generic structures that retain only the structural (i.e. base pairing) and topological (distances between base pairs and stem segments) of the assembled stem combinations. The specific nucleotide information pertaining to each base pair is discarded; only the notion that nucleotides in the respective positions formed a base pair is retained. These generic structures are then searched against the broad data set using specialized search techniques. For each search of a generic structure, negative matches are immediately discarded. Positive matches are subsequently incrementally increased in specificity in an exhaustive manner. This process continues until either no more positive matches are yielded, or the structure

cannot be increased in specificity (i.e. it is exactly specified) and has converged upon a maximal result. The positive results are recorded, and the next generic structure is processed accordingly. It is useful to elucidate this process with an illustration.

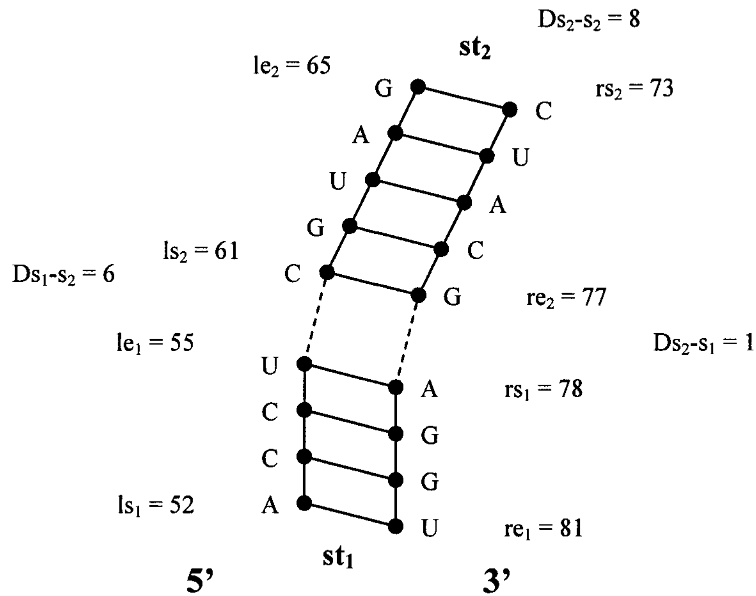


Figure 4.1 - Assembled stem segment combination from the solution space.

Figure 4.1 illustrates an example of an assembled combination of two stem segments as provided from the solution space. The depicted base pairs are those from the original source sequence as discovered by the stem detection methodology. The values Ds_1-s_2 and Ds_2-s_1 represent the distance between the nucleotides of the two stems on the 5' and 3' sides respectively. The value Ds_2-s_2 represents the distance between the 5' and 3' side of the stem st_2.

Now, the stem segment combination is reduced to a structure configuration without specific base pairing information. The individual nucleotide labeling (A, C, G, or U) is removed and replaced with a general base-base complement symbolism, N-N'. Note that N is the IUPAC code[4] for a generic nucleotide; for the purposes here, N' will represent its base pair complement. In addition, the specific stem segment location information has been removed and replaced with relative nucleotide distance information between the two stems. Since the nucleotide numbering information shown in Figure 4.1 is relative to the source sequence, such numbering bears no relevance to the frame of reference of the data search space. Only the relative distance between the nucleotides of constituent stems in the generic motif structure are relevant. The completed reduction is illustrated below in Figure 4.2.

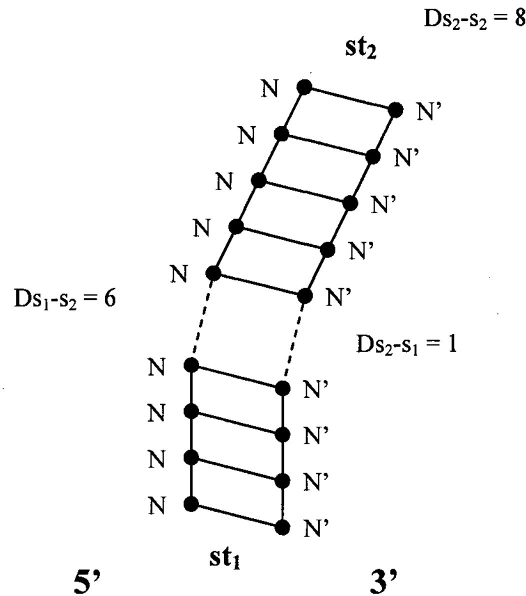


Figure 4.2 - Assembled stem segment combination reduced to generic structural form.

Note that the distance values are labeled from st_1 to st_2 and vice-versa, and for the loop of st_2, for clarity. In practice, the information required is simply a relative distance to the next stem; the identity of the next stem is not stored.

Now that the generic motif structure has been defined, it is then searched against the data space for structural matches. The actual search mechanism is detailed in the following section. It is necessary to note here, however, the rationale behind the generic structural search. Rather than searching for explicit base pairing at this point, the search simply looks for combinations of stem segments that match the specified configuration. Referring to Figure 4.2, a generic structural search based on that configuration would look for a pair of contiguous stems, one consisting of 4 base pairs and the other of 5 base

pairs. On the 5' side, the stems are separated by 6 nucleotides and on the 3' side, the stems are separated by 1 nucleotide. Note that any base pairs that satisfy the biological complementary relationship previously described are accepted as positive matches. Furthermore, note that the number of sequences that a structural motif must match before being considered a positive match is a level of support that is configurable prior to the initiation of the search. The goal of this methodology is to allow the data to drive the selection of the individual nucleotide base pair values. In this manner, the structural motif that is eventually converged upon will maximally reflect the underlying data sequences.

If the generic structural motif negatively matches the search space, it is discarded and the next generic structural motif is processed accordingly. However, if it positively matches the search space, it is systematically increased in specificity in a series of stages in order to elucidate a consensus secondary structure. In this process, the generic structural motif is considered as a set of base pairs. In the first stage, each structural motif is instantiated into n separate copies, where n is equal to the number of base pairs in the motif. For each instantiation i, the generic base pair at position i is set to all possible configurations of the A, C, G, and U nucleotides and tested against the search space. More precisely, the base pair at position i is tested with four possible configurations:

A-U, U-A, C-G, G-C

Where the first nucleotide denotes the 5' side of the base pair and the second nucleotide denotes the 3' side of the base pair. Note that the other base pairs retain their generic N-N' relationship. All negative matches are discarded and the positive matches are saved. In the next stage, each instantiation i of the previous stage is instantiated into $n - i$ separate copies. For each of these instantiations j, the generic base pair at position $i + j$ is set to all possible configurations of the A, C, G, and U nucleotides and tested against the search space. The process continues with positive matches retained and instantiated further, while negative matches are discarded. In this manner, all possible, matching configurations of the generic structural motif are considered.

To further illustrate this procedure, one can consider a structural motif as a collection of base pairs, enumerated from the 5' stem to the 3' stem. Further more, each base pair can be assigned a bit position in a binary sequence of length n, where n is the number of base pairs in the structural motif. The bit reflects whether or not that base pair is "set" and requires testing against all possible nucleotide base pairs. If the bit is clear (value of 0), then the base pair is tested only for its complementary relationship. If the bit is set, (value of 0), then the base pair needs to match specific base pairs. In the first stage, the structural motif is set to:

00000000...0

The above bit pattern indicates that each base pair requires testing against all possible nucleotide base pairs. Assuming this structural motif positively matches against the search space, the next stage has n separate instantiations:

100000000...0

010000000...0

001000000...0

etc.

Assume that the instantiation represented by 100000000...0 positively matches. At the next stage, this instantiation expands to:

110000000...0

101000000...0

100100000...0

etc.

Also assume that the instantiation represented by 010000000...0 positively matches. At the next stage, this instantiation expands to:

011000000...0

010100000...0

010010000...0

etc.

In this manner, all possible positively matching configurations are tested. Note that upon being set, each position corresponds to 4 separate searches – one for each valid nucleotide base pair. This results in a dramatic increase in the total number of searches required. Consider the base case where the generic structural motif is positively matched. In the immediately following stage, there are a total of n instantiations generated, one for

each position. At each of these instantiations, 4 sets of base pairs must be searched, resulting in 4^n possible search results. This 4^n factor is compounded as the search expands as positive matches are realized.

To further illustrate this processing, the generic structural motif of Figure 4.2 is processed through an example. First, the generic structural motif itself is examined. Assuming it positively matches the search space, it is copied into 9 separate instantiations, with the necessary base pairs processed as above. The first instantiation for this generic structural motif is depicted below in Figure 4.3

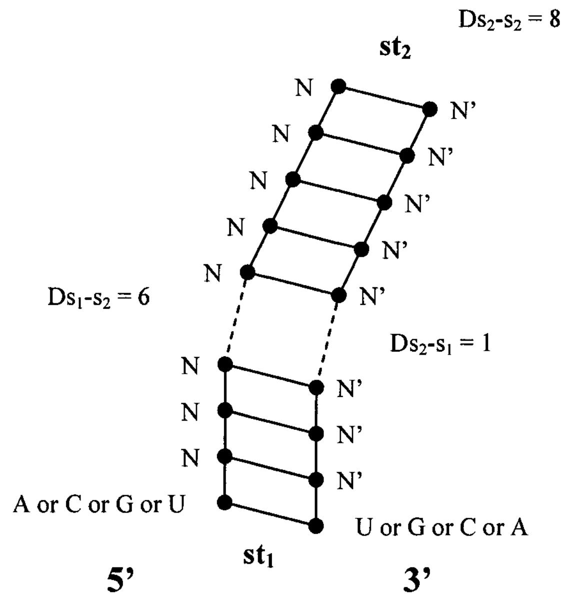


Figure 4.3 - An example instantiation of the generic structural motif.

In this example, all of the base pairs remain generic save the first base pair. There are four separate searches generated by this instantiation; one each for the base pairs A-U, C-G, G-C, and U-A.

By starting at the most generic instantiation of a secondary structural motif possible, and then systematically setting individual base pairs to specific nucleotides and discarding negative results, the secondary structure motif with maximal specificity to the data space can be converged. In order to be complete however, this technique requires an efficient system to determine whether or not a structural motif instantiation matches the data search space.

4.2 Structural Motif Search Methodology

The pattern that is to be searched against the data sequences consists of a structural motif that may or may not have specific base pairs. In order to utilize the suffix array based search techniques alluded to in section 2.2.3, the structural motif needs to be converted into a manageable form.

Although the structural motif is a two-dimensional secondary structure, the actual search must be performed against a one-dimensional primary structure – the data sequences. As such, the secondary structure must be converted into a form that can be readily matched against a set of sequences. This is performed by 'unraveling' the secondary structure motif into a primary structure representation, from the 5' end to the 3' end, into a chain of

nodes. For each nucleotide in the secondary structure motif, a node is created. Each node has several properties:

- Nucleotide type. A node is either representative of a 5' nucleotide or a 3' nucleotide.
- Stem start flag. A node is either a start of a stem (on the 5' side or 3' side), or it is not.
- Distance to next stem. If a node is the last node before the start of a new stem (on the 5' side or 3' side), this value contains the number of nucleotides between this node and the next node in the original source sequence.
- Nucleotide value. A node can be set to nucleotide values of A, C, G, U, or N.

For the structural motif instantiation illustrated in Figure 4.2, the node chain depicted in Figure 4.4 is generated. The properties relevant to each node are stored accordingly.

5' node	5' node	5' node	5' node	5' node	5' node	5' node	5' node	5' node
Start	N	N	N	Start	N	N	N	N
N			Dist=6	N				Dist=8

3' node	3' node	3' node	3' node	3' node	3' node	3' node	3' node	3' node
Start	N'	N'	N'	N'	N'	N'	N'	N'
N'					Dist=1			

Figure 4.4 - Node chain for the structural motif of Figure 4.2.

The motivation for this representation is an adaptation of the regular expression search methodology expressed by Baeza-Yates and Gonnet[17]. In that methodology, a regular expression to be searched is converted into a minimized deterministic finite automaton[42] (DFA). The text to be searched is pre-processed into a variant of a suffix tree. To process the search, the traversal of the minimized deterministic finite automaton (DFA) is then simulated on the suffix tree. When a final state is reached in the DFA, the search is halted and any sub-trees in the suffix tree represent matches. Since the nature of the structural motif requires information not available in a regular expression, nor is it clear how to represent such information in the form of a minimized DFA, the node chain described above is used to delineate the search path.

The data sequences to be searched are pre-processed into a suffix array. Each structural motif is converted into a node chain. Although the underlying data sequences are stored in a suffix array, for clarity, the description uses a suffix tree instead; the result is the same. The node chain is then traversed node by node; simultaneously, the suffix tree is traversed character by character. Since the traversal of both the suffix tree and the node chain is performed recursively, it suffices to describe a single recursive instance. Because the search path followed is delineated by the node chain, the recursive search algorithm is performed on a node-by-node basis. The search starts at the root of the node chain and the root of the tree.

If the node is a 5' node that is not set (that is, the node is described by the N generic nucleotide), the node automatically matches the current character in the tree. This character is then pushed onto a stack so that its complement can later be verified. If the node is a 5' node that is set (that is, the node is described by a specific nucleotide, A, C, G, or U), the character in the tree must be identical to the node nucleotide value. If it is identical, the node matches the character. Otherwise, the match fails.

If the node is a 3' node that is set (that is, the node is described by a specific nucleotide A, C, G, or U), the character in the tree must be identical to the node nucleotide value. If it is identical, the node matches the character. Otherwise the match fails. If the node is a 3' node that is not set (that is, the node is described by the N' generic nucleotide), the current character in the tree must be the complement of the top value of the stack. This

can be shown as follows. Every 5' node has a corresponding 3' node and is subject to the following possibilities:

- If the 5' node is set, nothing is pushed onto the stack. Since if the 5' node is set, the corresponding 3' node is also set and thus this scenario is of no consequence with regards to the stack.
- If the 5' node is not set, the current character in the tree at the time is pushed onto the stack.
    - If the next node is a 5' node and if it is set, nothing will be pushed onto the stack and the top of the stack has the character representing the current 5' node. If the node is not set, the next character in the tree is pushed and becomes the top element in the stack and must eventually match a complement of a 3' node.
    - If the next node is a 3' node, the top of the stack contains the character just pushed for the corresponding 5' node. In order for there to be a match, the next character in the tree must be the complement of the top of the stack.

Thus, if the current character of the tree is a complement of the top value of the stack, a match is made. The stack is then popped. Otherwise, the match fails.

Now that the matching rules for each node type have been established, the general algorithm for processing each node can be described. The rules for processing each node are as follows in order of operation:

- If the current character in the tree is a terminal symbol, this search path fails and yields no match.
- If the next node is the start of a new stem, and if the distance to the next node is non-zero, decrement the distance by one, advance the path in the branch by one character, and process recursively. By allowing multiple variants of distances to the start points of new stems, a certain, controllable degree of tolerance can be permitted between the distances of stems.
- If the tree is at a branch point, process each child branch recursively.
- If the node is the last node in the chain, then if the stack is empty, this search path matches. The leaves below this point in the suffix tree represent the matching sequences for this search path. If the stack is non-empty, this search path fails as there remain unmatched, non-set 5' nodes.
- If the node matches the character, advance the path in the branch by one character and process recursively. Otherwise, if mismatches are permitted, and the node is not set, the mismatch count is incremented. If the mismatch count is below a pre-defined limit, the path in the branch is advanced, and the search is continued recursively. Otherwise, there is no match and this search path fails.

The search is initiated at the root of the tree, and every child branch is processed. After all of the child branches are processed, the number of sequences matched in the process is tabulated. If this number of sequences meets or exceeds the pre-defined confidence level,

this instantiation of the secondary structure motif is considered a positive match. Otherwise, the instantiation is discarded. Note that a given structural motif may simultaneously exist in multiple paths in the tree; the search algorithm will discover all of these instances if they exist.

Allowances for variance in the distance between stems as well as numbers of mismatches are defined externally. Increasing these variables can possibly adversely affect the quality and reliability of the consensus secondary structure motif that is eventually inferred.

4.3 Scoring Results

Upon enumeration of a set of secondary structure motifs, it is necessary to score them in order to evaluate their value as candidates for consensus secondary structure elements. The manner in which this is done involves simply grading the information content within each structure. Out of the matching secondary structure motifs, the results with the least information are those which have no set base pairing relationships. That is to say, the results that are purely structural have the least amount of information content as they are the most generic. At a base level, these generic structural motifs are given a score of 1 for each of its constituent base pairs. As more and more base pairs are set, the information content of the structural motif increases as more commonality across the data set is elucidated. Each set base pair in a structural motif is ascribed a score of 3. The resultant score of a structural motif is the sum of the values described. This simple

scoring scheme allows the effective ranking of the structural motifs inferred. Moreover, the scoring values ascribed to generic, as well as set and even specific base pairs can be easily customized to suit a particular search criteria.

5 Cmyc Data

The source data used to validate the algorithms and methodologies previous expressed is the experimentally determined C-MYC IRES data derived by Quesne et al[43]. This data has been subject to biological experimental verification and is hypothesized to contain consensus secondary structures.

5.1 Data Set

The sequences used are eight experimentally derived sequences hypothesized to contain an IRES structure. Each sequence is approximately 400 nucleotides long. Four of the sequences are shown below in Figure 5.1.

```
>gibbon
AATTCCAGCGAGAGGCAGAGGGAGCGAGCGGGCGGGTCGGCTAGGGTGGAAGAGC
CGGGCGAGCAGAGCTGCGCTCCGGGCGTCCTGGGAAGGGAGATCCGGAGCGAA
TAGGGGGCTTCGCCTCCGGCCCAGCCCTTCCGCTGACCCCCCAGCCAGCGGTCC
GCAACCCTTGCCGCATCCACGAAACTTTGCCCATAGCAGCGGGCGGGCACTTTGCA
CTGGAACTTACAACACCCGAGCAAGGACGCGACTCTCCCGACGCGGGGAGGCTTAT
TCTGCCCATTTGGAGACACTTCCCCGCCGCTGCCAGTACCCGCTTCTCTGAAAGGCTC
TCCTTGCAGCTGTTTAGACGCTGGATTTTTTTAGGGTAGTGGAAAACCAGCAGCCTCC
CGCGACGATG
>marmoset
AATTTCAGCGAGAGGCAGAGGGAGCGAGCGGGCGGGCAGGCGAGGGTGGAAGAGC
CGGGCGAGCCGAGCTGCGCTCCGGGCGTCCTGGGAAGGGAGATCCGAAGTGAAAAG
GGGGCTTCGCCTCCGGTCCAGCCCTCCCGCTGACCCCCGCAGCCAGCAGCCCACAA
CCCTCGCCGCATCCACGAAACTTTGCCCTTAGCAGCGGGCGGGCACTTTGCACTGGAA
CTTACAACACCCTAGCAAGGACGCGACTCTCCCGACGCGGGGAGGCTATTCTCCCCA
TTTGAGGACACCTCCCCGCCGCTGTCGGGACCCGCTCCTCTGAAAGGTCTCCTTGCCG
CAGTTTGGACGCTGGATTTTTTTCGGGCAGTGGAAAACGAGGCTCCCGCGACGATG
>pig
AATTCCAGCGAGAGGCAGAGGGAGCGAGCGGGCGGGCCCTCCAGGGTGGAAGAGCAGAGC
CGGCCGAGCAATCTGAGTCGCGCTCTGGGCGCCCGGGGGAAGGGAGATCCGGAGTGAA
AGAGGGTCTTCGCCTCCGTCCCGGCCGCCCCCACCCCACCCTGCCCGCCGACCCCTGCCA
GCGGTCCGCCACCCGCGCCGCATCCACGAAACTTTGCCCACTGCAGCGGGCGGGTACTTT
CCACTGGAACTTACAACACCCGAGCGACAACGCGACTCTCCGGACGCGGAGAGGCTATTC
TGCCTATTTGGGGAGACACTTTTCCCTGTCGCTGCCCACGACTCGCTCCTCTGAAAGGCGCT
CCTCGCCGCTTTTTGGACGCTGGATTTCCTTCGGATAGTGGAAAACCCGGCTGCCGCGATG
>cat
AATCCTAGCAAGAGGCAGAGGGAGCGACGGGCGCGCCCGCCTGGGTAGAAGAGCAGAGC
GAGGCGAGCGAGCCGAGTCCGGCTCCGGGCGCCCGGGGAAGGGAGATCCGGAGTGAA
AGGGTGCTTCGCCTCCGGCCCACCCCACCCTGCCCGCCGCCCCCTGCCAGCGGTGCGC
AACTCCCCGCCGCACCCGCGAAACTTTGCCGGTTGCGGCGGGCGGACACTGTTCCCTGG
AACTTACAACACCCGAGCAAGAACGCGACTCTCCGGTCCGCCTATTTGGGGAAACACTTCT
CCCCTACGCTGCCCGGGACCCGCTCCTCTGAAAGGGCGCTCCTCGCCGCTTTTCGGACGC
TGGATTTCCTTCGGATAGGGGAAAACCCGCGAATG
```

Figure 5.1 - Sample of the Cmyc data sequences.

As previously discussed, the usage of experimentally derived data where a consensus secondary structure is postulated to exist is the more pertinent method of validation available. This data is most likely to produce a consensus secondary structure if the methodologies used in the search are correct. Since there exist no sequences which definitively do not contain consensus secondary structure motifs, it is not possible to perform negative match validation.

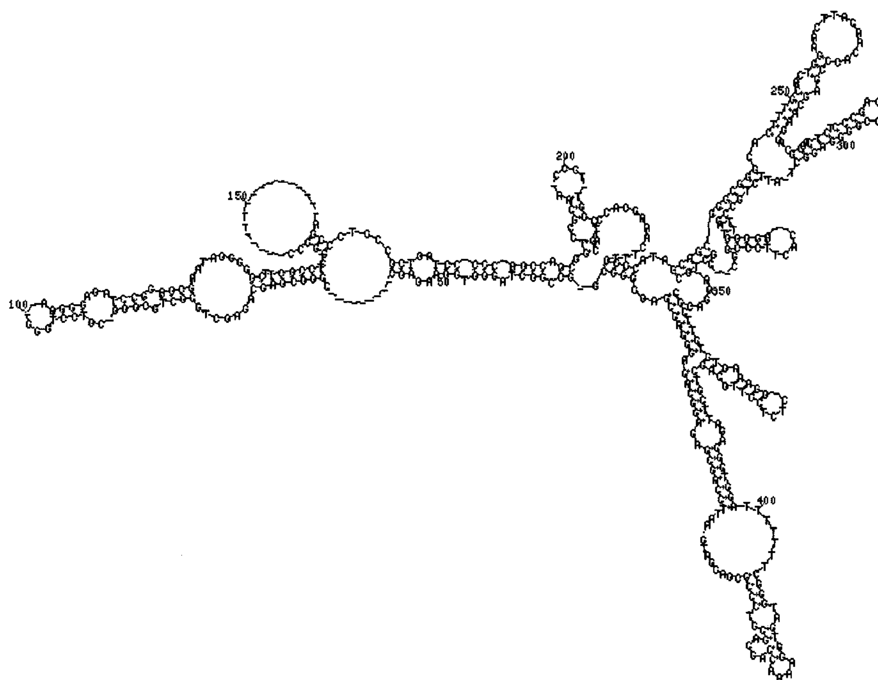
A single sequence out of the eight available sequences is chosen to serve as the source sequence from which the solution space is constructed. Arbitrarily, the sequence corresponding to the human is selected for this purpose. Its primary structure is shown below in Figure 5.2.

```
>human
AATTCCAGCGAGAGGCAGAGGGAGCGAGCGGGCGGCCGGCTAGGGTGGAAGAGC
CGGGCGAGCAGAGCTGCGCTGCGGGCGTCCTGGGAAGGGAGATCCGGAGCGAA
TAGGGGGCTTCGCCTCTGGCCCAGCCCTCCCGCTGATCCCCCAGCCAGCGGTCC
GCAACCCTTGCCGCATCCACGAAACTTTGCCCATAGCAGCGGGCGGGCACTTTGCA
CTGGAACTTACAACACCCGAGCAAGGACGCGACTCTCCCGACGCGGGGAGGCTATT
CTGCCCATTTGGGGACACTTCCCCGCCGCTGCCAGGACCCGCTTCTCTGAAAGGCT
CTCCTTGCAGCTGCTTAGACGCTGGATTTTTTTCGGGTAGTGGAAAACCAGCAGCCT
CCCGCGACGATG
```

Figure 5.2 - Cmyc sequence data corresponding to the human data.

The mfold secondary structure prediction with the lowest free energy of the human Cmyc sequence data[44] is depicted in Figure 5.3. The human Cmyc sequence data has a total of 398 nucleotides.

plt22gif by D. Stewart and M. Zuker
©2004 Washington University



dG = 99858.2 [initially -156.1] human

Figure 5.3 - mfold secondary structure prediction of the human IRES sequence data[44].

5.2 Results

Application of the software developed according to the methodologies and algorithms specified previously yielded interesting results. Upon utilizing the software application with the target human sequence data consisting of 398 nucleotides, the enormity of the number of structural motif possibilities was realized. Setting a criteria of a minimum stem length of 4 and maximum gap size of 100, the application detected 214 independent, contiguous stem segments. Processing this solution space proved to be an intractable task

given the number of potential motif combinations and the current level of efficiency of the application. In order to achieve computationally realizable results, various criteria are applied to reduce the solution space. Primarily, it proved to be necessary to restrict the stem detection search within the source Cmyc human data sequence to a region 100 nucleotides long. In doing so, a more thorough, exhaustive search could be applied against the data sequences from the solution space.

5.2.1 Experimental Results

Each experiment is conducted with specific search criteria. The parameters used are described below:

- Confidence level. The percentage of sequences in which positive matches are required in order for the search to be accepted. This value is chosen by the user to restrict the results as desired.
- Maximum Gap Size. The maximum gap size allowable by the stem detection algorithm.
- Maximum Bulge Offset. The maximum amount of variance permitted for the distance between 5' and 5', 3' and 5', and 3' and 3' stem segments during the search.
- Maximum Loop Offset. The maximum amount of variance permitted for the distance between the 5' and 3' stem segments during the search.
- Minimum Stem Length. The minimum length of stem detected.

- Maximum Stem Detection Mismatch. The maximum number of mismatches permitted by the stem detection algorithm.
- Maximum Motif Mismatch Per Stem. The maximum number of mismatches permitted for each stem segment in a motif during the search.
- Maximum Total Motif Mismatch. The maximum number of total mismatches permitted during the search.

For experimental purposes, these parameters are selected based on input from biologists and bioinformatics specialists in accordance with the target data. In a real-world application, these parameters are selected based on the desired search boundaries of the user. The effect of varying certain parameters is listed in the Appendix in Table 7.2 and Table 7.3. Varying these parameters seems to have no significant effect upon the summary results described in Section 5.2.2.

A selection of results illustrating a positive and negative match is provided.

5.2.1.1 Experiment 1

The search criteria for this experiment is detailed in Table 5.1.

Search criteria	Value
Confidence level	70.00%
Maximum Gap Size	100
Maximum Bulge Offset	1
Maximum Loop Offset	2
Minimum Stem Length	4
Maximum Stem Detection Mismatch	0

Maximum Motif Mismatch Per Stem	1
Maximum Total Motif Mismatch	3

Table 5.1 - Criteria used for Experiment 1.

The application of this search yielded a variety of results. The output result with the largest number of base pairs is listed below in Figure 5.4.

```
Motif 3089 matched 5 sequences for 71.43 coverage.
Motif had 18 pairs, 2 fixed, giving a score of 22.

Motif chain info:
 5:NNGNNN D: 10  3:N'N'N'CN'N' D: 13  5:NNNN D: 27  3:N'N'N'N' D: 6  5:NNNN D: 10
3:N'N'N'N' D: 9  5:NGNN D: 6  3:N'N'CN'
Matched these sequences:
cat (With offset 7 and 3 mismatches)
rat (With offset 5 and 3 mismatches)
gibbon (With offset 5 and 3 mismatches)
marmoset (With offset 6 and 3 mismatches)
pig (With offset 9 and 2 mismatches)
```

Figure 5.4 – Output result of Experiment 1.

The search of the solution space converged upon the above motif, with 2 base pairs ultimately being fixed out of a structure consisting of 18 base pairs. The offset value indicated how much variance was required in the distances between the individual stem segments comprising the structural motif in order for the match to be ascertained. A pictorial representation of the candidate consensus structure is illustrated below in Figure 5.5.

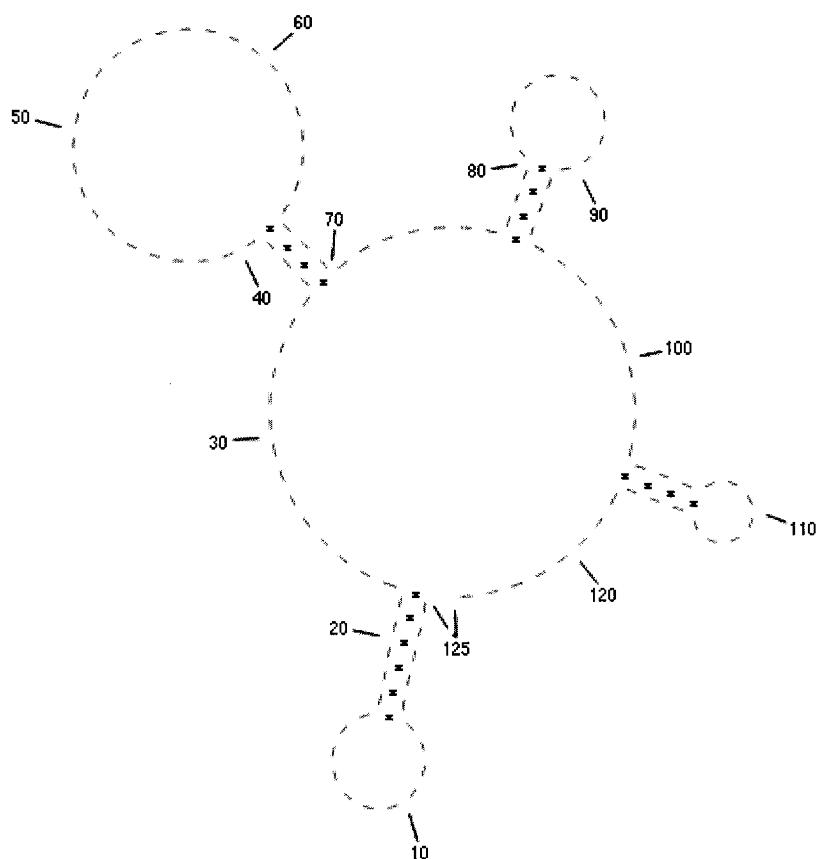


Figure 5.5 - Secondary structure representation of consensus motif for Experiment 1[45].

Comparison of the consensus secondary structure motif generated in Experiment 1 with experimentally derived analysis of the cmyc data yielded a significant amount of correlation. Out of the 18 base pairs detected, 8 of these correlated precisely with the 57 base pairs in the experimentally derived structure, resulting in 14.0% coverage. Since 8 out of the 18 base pairs correlated, the predicted structure is said to have a positive predictive value of 44.4% (given by 8/18).

5.2.1.2 Experiment 2

The search criteria for this experiment is detailed in Table 5.2.

Search criteria	Value
Confidence level	70.00%
Maximum Gap Size	100
Maximum Bulge Offset	1
Maximum Loop Offset	2
Minimum Stem Length	4
Maximum Stem Detection Mismatch	0
Maximum Motif Mismatch Per Stem	1
Maximum Total Motif Mismatch	3

Table 5.2 - Criteria used for Experiment 2.

The output result with the highest score is listed below in Figure 5.6.

```
Motif 507 matched 5 sequences for 71.43 coverage.
Motif had 12 pairs, 8 fixed, giving a score of 28.

Motif chain info:
 5:CNGA D: 7  3:TCN'G D: 7  5:NNGG D: 0  5:NAAG D: 23  3:CTTN' D: 9  3:CCN'N'
Matched these sequences:
woodchuck (With offset 4 and 2 mismatches)
gibbon (With offset 1 and 1 mismatches)
marmoset (With offset 1 and 1 mismatches)
cat (With offset 3 and 2 mismatches)
pig (With offset 3 and 1 mismatches)
```

Figure 5.6 - Output of result for Experiment 2.

The search of the solution space converged upon the above motif, with 8 base pairs ultimately being fixed. A pictorial representation of the candidate consensus structure is illustrated below in Figure 5.7.

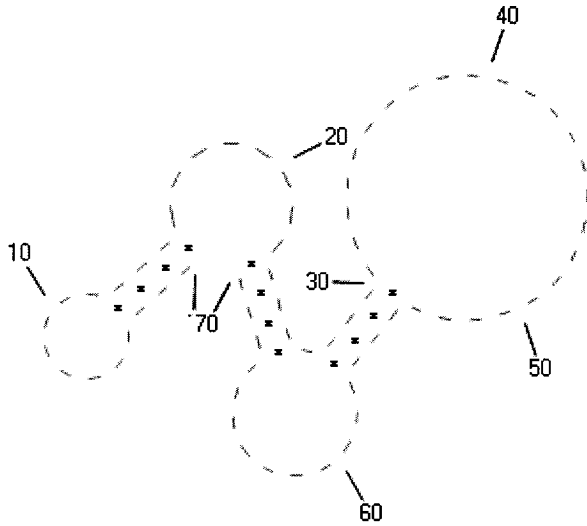


Figure 5.7 - Secondary structure representation of consensus motif for Experiment 2[45].

Comparison of the consensus secondary structure motif generated in Experiment 2 with experimentally derived analysis of the cmyc data yields no correlation. It is possible that although the application determined a consensus secondary structure motif in this region with reasonable accuracy, no biologically relevant structures have yet been discovered there. Additionally, it is likely that the scoring scheme selected does not necessarily select biologically relevant motifs.

5.2.2 Discussion of Results

In order to achieve computationally tractable results, the search region of the target sequence was limited to 100 nucleotides in length. In doing so, motifs longer than 100 nucleotides or those that are not completely contained within the search region, are not predicted.

The bulk of the results obtained through application of the methodologies presented herein against experimental data sequences yielded a significant number of candidate consensus secondary structure motifs. However, the bulk of the motifs obtained were, for the most part, generic structures with a low amount of specificity. In some circumstances, a higher degree of specificity was obtained. Out of the 1000 highest ranked results using the previously discussed scoring scheme, there were 193 motifs that positively correlated with the experimentally derived structures. The remaining motifs did not predict any experimentally derived base pairs. Out of the positively correlated results, the positive predictive values and coverage percentages in Table 5.3 are obtained.

Attribute	**Mean**	**Minimum**	**Maximum**
Coverage	7.8%	7.0%	21.1%
Positive Predictive Value	35.2%	23.5%	100.0%

Table 5.3 - Attributes of positively correlated motif predictions.

Using the same experimental Cmyc sequences, predictions generated from mfold have a mean positive predictive value of 12.3% with a coverage of 35.0%. This indicates that

the methodologies described in this work are capable of generating more accurate, but fewer results, in comparison to mfold.

6 Conclusion and Discussion

6.1 Conclusion

A data driven search methodology for determining consensus secondary structure motifs was described and successfully implemented. The methodology primarily utilized suffix array driven techniques to delineate both the target solution space as well as processing of the solution space against a set of data sequences in order to yield candidate consensus secondary structure motifs. As far as it is known to the author, this work represents the first suffix array based methodology for enumerating secondary structure motifs.

The results yielded from the experimentation on the Cmyc sequence data consisted of a large number of primarily more generalized secondary structure motifs, although some degree of specification was determined. Although the degree of convergence of the secondary structure motifs desired was not directly achieved, a significant proportion of the highest ranked predicted motifs exhibited positive correlation with experimentally derived secondary structures. The results of the application of the search methodologies are promising and merit further investigation.

Due to the computational intensity of the methodologies developed, the search process could not be applied as exhaustively as intended. It is thought that with further optimization of the processes involved in the construction of the solution space as well as

the search techniques utilized, the methodologies discussed in this document can yield more optimal results.

6.2 Discussion

The task of assembling all possible combinations of stem segments as discussed in section 3.2 is quite onerous in terms of space complexity, due to its combinatorial nature. One possible resolution involves the usage of a Cartesian tree[35] to manage the combination space. The Cartesian tree could be organized such that the maximum gap size is ascribed to the root of the tree. All sub-trees to the left of the root would have stems with a right end index less than that of the root. All sub-trees to the right of the root would have stems with a right end index greater or equal to that of the root. As the tree constructed is a Cartesian tree, these properties are necessarily recursive. Generating combinations of stems would then involve recursively processing the various internal nodes of the tree and ensuring that the nesting and adjacent criteria are met. Although the time complexity to generate all of the stem combinations would still be $O(2^n)$, the space complexity would be $O(n)$. Another possible resolution involves investigating the results of Bouthinon and Soldano[39] to determine if it is possible to incorporate inter-stem distance information into their methodologies.

The motif assembly and data sequence search methodologies presented within are not capable of detecting a class of secondary structure motifs known as pseudoknots. These are valid, but less frequently occurring secondary structures where a pair of stem

segments overlap[39]. It is desirable to detect such secondary structures and future work toward this end is worthwhile.

The task of searching the structural motif chain of nodes against the suffix array on a character by character basis is a time consuming process. Although linear with respect to the number of characters in the suffix array, it is combinatorial with respect to the number of 'N' type of generic nodes in the node chain. There are two possibilities to optimize this process that merit further investigation. It may be possible to develop a variant of a minimized deterministic finite automaton to represent the structural motif node chain. If so, then an adaptation of the Baeza-Yates and Gonnet regular expression search could be utilized. This search methodology requires a logarithmic amount time. Another possibility is to pre-process the structural motif node chain in some manner such that longest common extension queries with wildcards, as described by Gusfield[14], can be applied. Coupled with the palindrome search capabilities of the suffix array, opportunities for optimization may exist.

The scoring scheme utilized in this work proved to be suboptimal. Although a significant number of the highest ranked predicted motifs positively correlated with experimentally derived structures, a large proportion did not. An ideal scoring scheme would be such that the biologically relevant motifs are ranked highest. A scoring function based on Minimum Description Length principle was successfully applied to find regular motifs by Brazma et al[46]; its application to secondary structure motifs should be explored.

Alternatively, a more complex function could be derived that would take into account favourable stacking of adjacent base-pairs[13 47]. Further research in this area is desirable; a superior scoring scheme would greatly enhance the usability of the techniques detailed in this work.

7 Appendix

7.1 IUPAC Codes[48]

Symbol	Meaning
A	Adenine
C	Cytosine
G	Guanine
T	Thymine (DNA) Uracil (RNA)
M	A or C
R	A or G
W	A or T
S	C or G
Y	C or T
K	G or T
V	A or C or G; not T
H	A or C or T; not G
D	A or G or T; not C
B	C or G or T; not A
N	A or C or G or T

Table 7.1 - Various IUPAC Codes.

7.2 Additional Simulation Results

The following results in Table 7.2 and Table 7.3 indicate the mean coverage and mean positive predictive values obtained by independently varying the gap size and confidence levels respectively. For purposes of execution, the maximum amount of results returned by each iteration is capped to the same value.

Gap Size	Mean Coverage	Mean Positive Predictive Value
80	7.6%	31.8%
100	7.7%	33.03%
120	7.8%	33.9%
140	7.8%	33.9%
160	7.8%	33.9%

Table 7.2 - Results obtained by varying the gap size.

Confidence Level	Mean Coverage	Mean Positive Predictive Value
70	7.6%	31.8%
85	7.7%	33.03%
100	0%	0%

Table 7.3 - Results obtained by varying the confidence level.

7.3 Execution Statistics

The execution statistics of the experiment conducted and described in Section 5.2 required 19 hours, 23 minutes, and 39 seconds to complete.

[1] C. Hellen, and P. Sarnow. Internal ribosome entry sites in eukaryotic mRNA molecules. *Genes and Development.* 15 (13): 1593 – 1612, 2001.
[2] B. Alberts et al. *Essential Cell Biology, 2nd ed.* 5: 169 – 7: 265, 1998.
[3] D. Searls, The Computational Linguistics of Biological Sequences. *Artificial Intelligence and Molecular Biology.* 2:47-120. (AAAI/MIT Press, 1993).
[4] Abbreviations and Symbols for Nucleic Acids, Polynucleotides and their Constituents. *IUPAC-IUB Commission on Biochemical Nomenclature (CBN).* http://www.chem.qmul.ac.uk/iupac/misc/naabb.html
[5] G. Pesole, and S. Liuni. UTRdb: A specialized database of 5'and 3' un-translated regions of eukaryotic mRNAs. *Nucleic Acids Research.* 30 (1): 335 – 340, 2002.
[6] D. K. Chiu, and T. Kolodziejczak. Inferring consensus structure from nucleic acid sequences. *CABIOS* (7): 347 – 352, 1991.
[7] R. R. Gurell, and C. R. Woese. Higher order structural elements in ribosomal RNAs: pseudo-knots and the use of noncanonical pairs. *Proc. Natl Acad. Sci. USA* (87): 663 – 667, 1990.
[8] R. R. Gutell, A. Power, G. Z. Hertz, E. J. Putz, and G. D. Stormo. Identifying constraints on the higher-order structure of RNA: continued development and application of comparative sequence analysis methods. *Nucl. Acids Res.* (20): 5785 – 5795, 1992.
[9] R. Nussinov, G. Piecznik, J. R. Griggs, and D. J. Kleitman. Algorithms for loop matching. *SIAM J. Appl. Math* (35): 68 – 82, 1978.
[10] M. S. Waterman. Secondary structure of single-stranded nucleic acids. Foundations and Combinatorics, *Advances in Mathematics Supplementary Studies* (1): 167 – 212, 1978.
[11] M. Zuker, and P. Stiegler. Optimal computer folding of large RNA sequences using thermodynamics and auxiliary information. *Nucleic Acids Research* (9): 133 – 148, 1981.
[12] A. Brazma, I. Jonassen, I. Eidhammer, and D. Gilbert. Approaches to the automatic discovery of patterns in biosequences. *Journal of Computational Biology* (5): 279 – 305, 1998.
[13] M. Zuker, D. H. Matthews, and D. H. Turner. Algorithms and thermodynamics for RNA secondary structure prediction: A practical guide. *RNA Biochemistry and Biotechnology.* NATO ASI Series, Kluwer Academic Publishers, 1999.
[14] D. Gusfield. *Algorithms on Strings, Trees, and Sequences.* Cambridge University Press, 1999.
[15] B. Billoud, M. Kontic, and A. Viari. Palingol: A declarative programming language to describe nucleic acid's secondary structures and to scan sequence databases. *Nucleic Acids Research.* 24 (8): 1395 – 1404, 1996.
[16] P. Pevzner. *Computational Molecular Biology: An Algorithmic Approach.* The MIT Press. 8: 135 – 151, 2000.
[17] R. A. Baeza-Yates, G. H. Gonnet. Fast test searching for regular expressions or automaton searching on tries. *Journal of the ACM.* 43 (6): 915 – 936, 1996.
[18] I. Jonassen, J. F. Collins, and D. G. Higgins. Finding flexible patterns in unaligned protein sequences. *Protein Science* (4): 1587 – 1595, 1995.
[19] P. Weiner. Linear pattern matching algorithms. *Proceedings of the 15th IEEE Symposium on Switching and Automata Theory.* 15: 1 – 11, 1973.
[20] E. McCreight. A space-economical suffix tree construction algorithm. *Journal of the Association of Computing Machinery.* 23 (2): 262 – 272, 1976.
[21] S. Kurtz. Foundations of sequence analysis. *Lecture notes for a course in the winter semester of 2000/2001.* 4: 35 – 58, 2002.
[22] P. Bieganski, J. Rield, and J. Carlis. Generalized Suffix Trees for Biological Sequence Data: Applications and Implementation. *The Regents of the University of Minnesota*, 1994.
[23] C. Lewis. *http://homepage.usask.ca/~ctl271/810/approximate_matching.shtml*, University of Saskatechewan. 2003.
[24] R. Giegerich and S. Kurtz. From Ukkonen to McCreight and Weiner: A unifying view of linear-time suffix tree construction. *Algorithmica.* 19: 331 – 353, 1997.

[25] S. Kurtz. Reducing the space requirements of suffix trees. *Software – Practice and Experience.* 29 (13): 1149 – 1171, 1999.
[26] U. Manber, and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing.* 22 (5): 935 – 948, 1993.
[27] M. Abouelhoda, et al. The enhanced suffix array and its applications to genome analysis. *WABI 2002,* LNCS 2452: 449 – 463, 2003.
[28] M. Abouelhoda, et al. Optimal exact string matching based on suffix arrays. *SPIRE 2002*, LNCS 2476: 31 – 43, 2002.
[29] J. Karkkainen, and P. Sanders. Simple linear work suffix array construction. *Proc. 13th International Conference on Automata, Languages and Programming.* Springer, 2003.
[30] P. Ko, and S. Aluru. Space efficient linear time construction of suffix arrays. Pattern Matching, 2003.
[31] D. Kim et al. Linear-time construction of suffix arrays. *CPM 2003*, LNCS 2676: 186 – 199, 2003.
[32] B. Smyth. Computing Patterns and Strings – Errata & Commentary.
http://www.computing.edu.au/~smyth/patterns.shtml.
[33] T. Kasai et al. Linear-time longest-common-prefix computation in suffix arrays and its applications. *CPM 2001*, LNCS 2089: 181 – 192, 2001.
[34] M. Bender and M. Farach-Colton. The LCA problem revisited. *Proc. 4th Latin American Symposium on Theoretical Informatics.* LNCS 1776: 88 – 94, 2000.
[35] J. Vuillemin. A unifying look at data structures. *Communications of the ACM.* 23 (4): 230 – 241, 1980.
[36] S. Nilsson. Treaps in java. *Dr. Dobbs' Journal.* 267: 40 – 44, 1997.
[37] R. Seidel and C. Aragon. Randomized search trees. *Proc. 30th IEEE FOCS.* 1999.
[38] S. Aluru. Suffix trees and suffix arrays. *Lecture Notes, Iowa State University.* 1: 1 – 22, 2003.
[39] D. Bouthinon and H. Soldano. A new method to predict the consensus secondary structure of a set of unaligned RNA sequences. *Bioinformatics.* 15 (10): 785 – 798, 1999.
[40] S. Muggleton. Inverse entailment and Progol. *New Generation Computing Journal,* (13): 245 – 286, 1995.
[41] P . Gardener. RNA secondary structure fitness.
http://www.massey.ac.nz/~ppgardne/results/review/node33.html. Massey University. 2001.
[42] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley. 1979.
[43] J. Quesne et al. Derivation of a structural model for the c-myc IRES. *Journal of Molecular Biology.* 310: 111 – 126, 2001.
[44] D.H. Mathews et al. Expanded Sequence Dependence of Thermodynamic Parameters Provides Robust Prediction of RNA Secondary Structure. *J. Mol. Biol.* (1999).
[45] Pseudoviewer 2. http://wilab.inha.ac.kr/pseudoviewer2/. *Web Intelligence Lab*, School of Computer Science and Engineering, Inha University. 2003.
[46] A. Brazma, E. Jonassen, J. Ukkonen, and J. Vilo. Discovering patterns and subfamilies in biosequences. *Proc. 4th International Conference Intelligent Systems for Molecular Biology*, ISMB '96, 34 – 43, 1996.
[47] Y. Ding, C. Lawrence. A Bayesian statistical algorithm for RNA secondary structure prediction. *Computers & Chemistry* 23 (3-4): 387 – 400, 1999.
[48] A. Cornish Bowden. Nucl Acid Res 13: 3021 – 3030, 1985.