# Algorithms and Complexity for Annotated Sequence Analysis

by

Patricia Anne Evans

B.Sc., University of Alberta, 1991
M.Sc., University of Victoria, 1994

A thesis submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
in the Department
of
Computer Science

We accept this thesis as conforming
to the required standard

---

Dr. Michael R. Fellows, Co-Supervisor (Department of Computer Science)

---

Dr. Frank Ruskey, Co-Supervisor (Department of Computer Science)

---

Dr. Valerie King, Departmental Member (Department of Computer Science)

---

Dr. Ben Koop, Outside Member (Department of Biology)

---

Dr. Tao Jiang, External Examiner (Department of Computing and Software,
McMaster University)

Supervisors:

Dr. Michael R. Fellows

Dr. Frank Ruskey

# Abstract

Molecular biologists use algorithms that compare and otherwise analyze sequences that represent genetic and protein molecules. Most of these algorithms, however, operate on the basic sequence and do not incorporate the additional information that is often known about the molecule and its pieces. This research describes schemes to combinatorially annotate this information onto sequences so that it can be analyzed in tandem with the sequence; the overall result would thus reflect both types of information about the sequence. These annotation schemes include adding colours and arcs to the sequence. Colouring a sequence would produce a same-length sequence of colours or other symbols that highlight or label parts of the sequence. Arcs can be used to link sequence symbols (or coloured substrings) to indicate molecular bonds or other relationships. Adding these annotations to sequence analysis problems such as sequence alignment or finding the longest common subsequence can make the problem more complex, often depending on the complexity of the annotation scheme. This research examines the different annotation schemes and the corresponding problems of verifying annotations, creating annotations, and finding the longest common subsequence of pairs of sequences with annotations. This work involves both the conventional complexity framework and parameterized complexity, and includes algorithms and hardness results for both frameworks. Automata and transducers are created for some annotation verification and creation problems. Different restrictions on layered substring and arc annotation are considered to de-

termine what properties an annotation scheme must have to make its incorporation feasible. Extensions to the algorithms that use weighting schemes are explored.

Examiners:

Dr. Michael R. Fellows, Co-Supervisor (Department of Computer Science)

Dr. Frank Ruskey, Co-Supervisor (Department of Computer Science)

Dr. Valerie King, Departmental Member (Department of Computer Science)

Dr. Ben Koop, Outside Member (Department of Biology)

Dr. Tao Jiang, External Examiner (Department of Computing and Software, McMaster University)

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

## 1.1 Overview of Work

The analysis and comparison of sequences of symbols are a fundamental part of computer science and have applications in many other fields of study. Molecular biology, in particular, is a fertile source of problems in this area; protein and genetic molecules can be viewed as long sequences of their basic constituents. This view, however, is a simple one. The function and physical structure of these molecules, both as a whole and their components, is not easily determined from the basic sequence. An informative sequence is frequently accompanied by other information about the sequence and its parts. This auxiliary information can be taken into account when the sequences are analyzed, to improve the results of this analysis. Faced with these different types of information, we can work with them separately to get independent results; work separately and attempt to combine the results into an aggregate; or combine the information in such a way that it can be worked with as a whole. To implement this last option, we must represent the additional information in such a way that it can be used easily with the original basic sequence to produce results about the complete data.

A *basic sequence* is the sequence of base symbols that form the fundamental, unannotated sequence. Mathematically, an *alphabet* is a set of symbols, generally represented by $\Sigma$. Unless otherwise indicated, $\Sigma$ is a finite set with size $|\Sigma|$. A *sequence* over the alphabet $\Sigma$ is a word $x \in \Sigma^*$, with length $|x|$. A sequence can also be referred to as a *string*. For a basic sequence, the alphabet can also be referred to as the set of bases. DNA, for example, consists of sequences with 4 bases, represented by the set $\{A, C, G, T\}$. Protein sequences are sequences that use a set of 20 amino acids as their alphabet.

An *annotation scheme* is a system of representing additional information (beyond that found in the basic sequence) in a way that relates it to the basic sequence. An individual *annotation* for a specific sequence is its associated additional information, as represented according to the chosen annotation scheme. Taken together, a basic sequence and its annotation form an *annotated sequence*. Note that an annotation or an annotated sequence may themselves be sequences. Specific annotation schemes are defined and discussed in chapter 3.

This additional information to be represented as an annotation can come from many different sources. One prominent source of information in molecular biology is the secondary structure of the molecules. While the primary structure of a molecule is the sequence of bases, its secondary structure is how this sequence folds into a three-dimensional structure. Another source is the function of specific substrings of the molecular sequences, and how these substrings can affect overall function and its expression. Auxiliary information can be independent of the basic sequence information, and not directly implied by it. For example, the secondary structures of the proteins actin and heatshock-70 are very similar while their sequences are quite dissimilar [FMKH91]. Other auxiliary information can be directly computed from the sequence itself, but may need to be represented through annotations so it can be used more efficiently. Sequence substrings with particular properties can be located only once and highlighted so that their existence and location can affect any further sequence analysis. A *motif*, or pattern that characterizes a family of sequences, is one type of useful substring. Motifs are heavily used in protein analysis. Other types of important substrings include molecule binding sites, gene promotors, and gene anchors.

A piece of a sequence can be either a substring or a subsequence. A sequence $y$ is a *subsequence* of $x$ if the sequence $x$ can be transformed into $y$ by deleting some symbols from $x$. The order of the remaining symbols must be preserved. On the other hand, a sequence (or string) $y$ is a *substring* of $x$ if $y$ is a contiguous piece of $x$, ie. $x \in \{\Sigma^* y \Sigma^*\}$. A substring is also a subsequence, but it will be called a substring if it is to be viewed as a contiguous piece of the sequence. For example, let $a\ c\ g\ a\ t\ a\ c$ be a sequence. Then $c\ a\ t$ is a subsequence of $a$ **c** $g$ **a** **t** $a\ c$, but not a substring. The sequence $g\ a\ t\ a$ is a substring of $a\ c$ **g a t a** $c$, as well as a subsequence. However, the sequence $t\ a\ g$ is not a subsequence of $a\ c\ g\ a\ t\ a\ c$, as the symbols are not in the same order.

Once substrings have been located, relationships between these substrings can also be defined, located, and superimposed onto the basic sequence to affect subsequent analysis. These relationships can range from a link between a pair of substrings, signifying that they belong together, to a complex tree structure. Molecular bonds between bases in the sequence can be treated as binary relationships between the symbols.

This research presents techniques and tools for analyzing sequences with additional information that can be expressed combinatorially and superimposed onto the sequence. The overlay of this information enables the analysis to take both the basic sequence and the additional information into account simultaneously. I also explore the complexity of analysis that incorporates these representations, and its limitations. Since the combinatorial representations have many naturally occurring parameters, I explore both the conventional complexity of this sequence analysis and its parameterized complexity. Parameterized complexity is discussed in [DF99] and applied to computational biology and sequence analysis in [BDFHW95] and [BDFW95]. The combinatorial representations examined include colourings, which can represent values and string or sequence partitions; coloured substrings and layers of coloured substrings, which can represent more complex relations and structures; and arcs, which represent binary relations, including molecular bonds between bases represented by symbols.

The method of sequence analysis focused on is sequence comparison, particularly pairwise comparison. When two sequences are compared, a measure of their similarity can be determined; this method can be used to find existing sequences that are the most similar to a new sequence. Along with determining how similar two sequences are, pairwise sequence comparison can also determine in what ways they are similar, revealing common sequence elements and fragments. Their edit distance, and how one sequence can be transformed into the other, can also be determined. In order to incorporate the additional information into such comparisons, it needs to be superimposed onto the basic sequence, producing a *combinatorially annotated* sequence. This research examines the problems inherent in producing such an annotation and incorporating it into sequence analysis.

These problems are divided into three categories. If the annotation already exists, it must be checked to ensure that it meets the correct format, and conforms to any restrictions on the complexity of the annotation. If the annotations do not exist, or are in the wrong format, they must be created or transformed. Creation involves searching for key pieces of a sequence, while transformation involves converting from a storage representation to one usable by the comparison algorithms. Finally, annotated sequences are compared in a way that uses both the basic sequences and the annotations to arrive at a joint measure of their similarity. This measure indicates how similar a pair of sequences are if they are compared in a way that preserves their annotation, and the information contained therein.

The centerpiece of this dissertation is a detailed analysis of the longest common subsequence problem for arc-annotated sequences, with a variety of restrictions on the annotation scheme. Several different parameters are also examined. This work shows the level of restriction needed to make the problem feasible, as demonstrated by hardness proofs and parametric algorithms. The main algorithm is also extended to work with several different types of similarity weighting schemes.

## 1.2 Biological Setting

The genetic material for most organisms is encoded in a long macromolecule of deoxyribonucleic acid (DNA). DNA is made from four different nucleotides, which are referred to as bases: adenine, cytosine, guanine, and thymine (represented by A, C, G, and T). The genetic code can thus be viewed as a long sequence that uses these four letters. Each base has a complementary partner with which it fits together. A is paired with T, and C is paired with G. These complementary pairs are essential to the structure of DNA and the replication and transcription of its code. DNA strands occur in pairs, with the bases on one strand fitting together with a complementary sequence of bases on the other strand. These two strands twine together in a long double helix shape. If the strands are separated, the single

DNA strand can be copied, or *replicated*, by building a new complementary strand. It can also be *transcribed* into RNA, a related ribonucleic acid that consists of A, C, G, and U (uracil), which can then be used as instructions for building a protein according to the genetic code. RNA can also, in some circumstances, be *reverse transcribed* into the DNA to alter the genetic code.

Messenger RNA molecules copy the genetic information from DNA. The DNA sequence is composed of genes that code for proteins, which have associated gene promotors and anchors preceding them. The protein coding region of the genes, or *exons*, is also generally interspersed with non-coding regions, called *introns*. The introns are spliced out of the messenger RNA, leaving the part of the sequence that codes for the protein. This resulting sequence is then read in triplets called *codons* to select the amino acids from which proteins are built. Each 3-base sequence has a corresponding amino acid that it encodes. As there are 20 amino acids and $4^3 = 64$ different 3-base codons, some amino acids have several different codons. There are also three termination codons that signal the end of the protein sequence. Since an ordered triple is used to select an amino acid, it is essential that the direction and the reading frame (which position, modulo 3, is the start of the codons) is correct. Reading out of phase can produce a completely different amino acid sequence.

Proteins are built at the ribosomes of a cell, where the messenger RNA picks up complementary transfer RNA. Transfer RNA has a three-dimensional cloverleaf structure built from approximately 80 bases. It also attaches to the required amino acid, bringing it into its correct place in the sequence. The ribosomes also have RNA whose three-dimensional structure enables them to interact with the other molecules physically. Ribosomal RNA can also act as enzymes to build molecules and break them apart. The three-dimensional structure of RNA can thus be extremely important to its function, and evolution (through mutation and editing of the sequence) is likely to preserve common structures.

Once a protein sequence has been built, it folds up into a complex three-dimensional structure. This structure will define how it can bind to proteins and to other molecules and atoms. Proteins can bind together to produce a larger structure, and they can also react as enzymes to build or break apart molecules. The three-dimensional structure is produced by the amino acid residues (the variable side chain of the molecule that determines the amino acid's identity), in reaction with the other residues and the environment. The overall structure of a protein has many different types of substructures. Most notable among these general types are the $\alpha$-helices and $\beta$-strands. These latter substructures frequently form larger substructures called $\beta$-sheets, running in parallel or antiparallel directions. Ion binding sites are also important protein substructures. Proteins can thus be compared by their structures as well as, or instead of, their sequences.

Determining the correct fold (or folds) of a protein is a major open problem in protein analysis. The converse problem, that of finding an amino acid sequence that will produce a particular fold or structure, is also extremely useful, as it would enable a protein to be designed to fit a target surface or binding site in an existing protein. These designer drugs can take the place of absent proteins, or bind to existing proteins in order to stop them from reacting to something else. The link between sequence and structure for both proteins and RNA is of critical importance, but is also in need of greater exploration and analysis tools.

Further information on the background of computational biology may be found in [Wat95].

## 1.3 Definitions

This work uses the following terminology. Some further definitions, particularly the mathematical representation of annotations, are given in chapter 3. Some of these definitions, those related to languages and automata, are taken from [Gur89].

**common subsequence**

A sequence $y$ is a *common subsequence* of sequences $S_1$ and $S_2$ if $y$ is a subsequence of $S_1$ and $y$ is a subsequence of $S_2$.

**colour**

A *colour* is a label associated with a symbol, substring, or other object. All colours are from some finite set of colours $C$. There can also be a "blank" colour, represented numerically by 0.

**arc**

An *arc* is a directed edge $(p_1, p_2) \in P \times P$, where $P$ is the set of positions in the sequence. If $n$ is the length of the sequence, $P = \{1, \ldots, n\}$. An arc can be viewed as a link that connects two symbols that are part of the same sequence. The order of the pair $(p_1, p_2)$ should be consistent with the sequence order, so $p_1 < p_2$.

**graph**

A *graph* $G = (V, E)$ is a set of *vertices* $V$ and a set of *edges* $E \in \{(u, v) \mid u \in V, v \in V, u \neq v \}$. These edges are undirected, so $(u, v) = (v, u)$ for any $v$ and $u$ from $V$.

**independent set**

A pair of vertices $u$ and $v$ are *independent* in a graph $G = (V, E)$ if they are not connected by an edge, ie. $(u, v) \notin E$. A set of vertices $V' \subseteq V$ is an *independent set* in $G$ if $\forall u, v \in V'$, $(u, v) \notin E$.

**clique**

A pair of vertices $u$ and $v$ are *adjacent* in a graph $G = (V, E)$ if they are connected by an edge, ie. $(u, v) \in E$. A set of vertices $V' \subseteq V$ is an *clique* in $G$ if $\forall u, v \in V'$, $(u, v) \in E$.

**vertex cover**

An edge $e$ of a graph $G = (V, E)$ is *incident* on a vertex $v$ if $v$ is one of the endpoints of $e$. A set of vertices $V' \subseteq V$ is a *vertex cover* in $G$ if $\forall e \in E$, $\exists v \in V'$ such that $e$ is incident on $v$ ($\exists u \in V$ such that $(u, v) = e$).

**null string**

The *null string*, represented by $\lambda$, is the string of length 0.

**Kleene closure**

For any language $L$, the *Kleene closure* $L^*$ of that language is defined recursively by:

- $\lambda \in L^*$
- if $x \in L$, then $x \in L^*$
- if $x \in L$ and $y \in L^*$, then $xy \in L^*$
- $L^*$ contains only those words that can be generated by these rules

The symbol $^*$ used in Kleene closure is known as the *Kleene star*. The language $L^+$ is defined as $L^* - \{\lambda\}$.

**regular language**

The set of *regular languages* over an alphabet $\Sigma$ is the set of sets defined by:

- $\emptyset$, $\{\lambda\}$, and $\{a\}$ for each $a \in \Sigma$ are all regular languages

- if $L_1$ and $L_2$ are regular languages, then their union $L_1 \cup L_2$, composition $L_1L_2 = \{xy \mid x \in L_1 \text{ and } y \in L_2\}$, and Kleene closure $L_1^*$ are also regular languages

- all languages which cannot be generated by these rules are not regular

**regular expression**

A *regular expression* is an expression that denotes a regular language algebraically using symbols from the alphabet, union, composition, Kleene star, and parentheses.

**finite-state machine**

A *finite-state machine* or *finite automaton* is a tuple $\langle Q, \Sigma, \delta, q_0, F\rangle$ where

- $Q$ is a finite set of states

- $\Sigma$ is a finite alphabet

- $\delta$ is a relation from $Q \times (\Sigma \cup \{\lambda\})$ to $Q$

- $q_0 \in Q$ is the start state

- $F \subseteq Q$ is the set of accepting states

Informally, a finite automaton is a machine with an input tape, a finite set of states and no other memory. Its output is restricted to "accept" or "reject". A finite-state machine is *deterministic* if $\delta$ is a function. Note that the set of languages accepted by finite automata is exactly the set of regular languages.

**finite-state transducer**

A *finite-state transducer* is a tuple $\langle Q, \Sigma, \Delta, \delta, q_0, F\rangle$ where

- $Q$ is a finite set of states

- $\Sigma$ is the input alphabet
- $\Delta$ is the output alphabet
- $\delta$ is a relation from $Q \times (\Sigma \times \{\lambda\})$ to $Q \times (\Delta \times \{\lambda\})$
- $q_0 \in Q$ is the start state
- $F \subseteq Q$ is the set of accepting states

Informally, a finite-state transducer is a machine with an input tape, a finite set of states, and an output tape. It has no internal memory, but can output more than just "accept" or "reject".

**push-down automaton**

A *push-down automaton* is a tuple $\langle Q, \Sigma, \Gamma, \delta, q_0, Z_0, F \rangle$ where

- $Q$ is a finite set of states
- $\Sigma$ is a finite input alphabet
- $\Gamma$ is a finite stack alphabet
- $\delta$ is a relation from $Q \times (\Sigma \cup \{\lambda\}) \times (\Gamma \cup \{\lambda\})$ to $Q \times \Gamma^*$
- $q_0 \in Q$ is the start state
- $Z_0 \in \Gamma$ is the bottom symbol of $\Gamma$
- $F \subseteq Q$ is the set of accepting states

Informally, a push-down automaton is a machine with a finite set of states an a single stack as memory. Note that the set of languages accepted by push-down automata is exactly the same as the set of languages with context-free grammars.

# Chapter 2

# Background

## 2.1 Sequence Comparison

Much research has already been done on using sequence comparison algorithms for computational biology. Smith and Waterman [SW81] found common molecular subsequences by finding the weighted longest common subsequence of a sequence pair. Needleman and Wunsch [NW70] originally proposed this technique to find similarities between protein sequences. The longest common subsequence algorithm, also discovered independently by others, is a dynamic programming algorithm that find the longest common subsequence between progressively longer prefixes, and runs in time $O(nm)$ (where $n$ and $m$ are the lengths of the two sequences).

The fundamental longest common subsequence algorithm computes the entries for a table $[T[i,j]]_{n \times m}$ where $T[i,j]$ is given by the following recurrence.

$$T[i,j] = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ \max \begin{cases} T[i-1,j] \\ T[i,j-1] \\ T[i-1,j-1] + w(S_1[i], S_2[j]) \end{cases} & \text{otherwise} \end{cases}$$

where $S_1[i]$ is symbol $i$ from sequence $S_1$, $S_2[j]$ is symbol $j$ from sequence $S_2$, and the function $w$ references the weight table

$$w(x,y) = \begin{cases} 1 & \text{if } x = y \\ 0 & \text{otherwise} \end{cases}$$

For each maximum value, the algorithm can also store the location (or locations) of the previous match along the maximum path (or paths). This information can be used to trace back through the table from the final location $T[n,m]$ to find the actual longest common subsequence (or set of such subsequences) instead of just its length. This subsequence and trace can then be used to align the two sequences, matching the corresponding symbols from the longest common subsequence to produce an alignment of the symbols in the two sequences that preserves their order.

The algorithm can be altered to produce a weighted score and associated alignment by changing the weight table. Instead of having a weight of 1 if the symbols match and a weight of 0 otherwise, the weights can be set to reflect the likelihood of different symbol substitutions. To produce an edit distance or similarity measure for a pair of sequences, a penalty for deleting a symbol (or the corresponding symbol insertion) can also be applied. This notion can be extended to have the penalty for a contiguous deleted piece be any nondecreasing function in the length of the piece deleted. Alternatively, the penalty can be applied once for any gap, irrespective of its size. The algorithm proposed by Needleman and Wunsch [NW70] uses weights for symbol matches and a set penalty for each gap in the matched sequence. These variations on the longest common subsequence algorithm are used to find database sequences similar to a query sequence by determining an overall weight of sequence similarity; this method is incorporated into the SEQSEE tool [WB-WRS94], among others. The commonly used FASTA program for finding similar protein sequences [PL88] selects the top sequences using a restricted, faster scoring method that asymptotically takes $O(\frac{nm}{|\Sigma|})$ time. After it selects sequences that have regions of high similarity using the faster technique, these sequences are scored again using the Needleman-Wunsch algorithm to produce their final similarity scores and alignments.

Local alignments that maximize the similarity score for substrings of the sequences can be found by allowing the score to be reset to zero with the alignment restarted if the score drops below zero due to gap penalties [SW81]. This technique, however, is directionally biased in that reversing both input strings can produce a different similarity score than the original score. If a small region of high similarity is separated from a large region of high similarity by gaps, the small region can be discarded from the local alignment if it is to the left of the large region, but included if it is to the right.

The longest common subsequence algorithm is also used by Wagner and Fischer to solve the string-to-string correction problem [WF74]. They produce an essentially identical algorithm that finds the edit distance between two strings, for the edit operations of character substitutions, character insertions, and character deletions. For edit distance (which is larger if the similarity measure is smaller), each operation increases the distance, and a subsequence or alignment of minimum distance is found. They discuss applying this edit distance to spelling correction for programming language keywords, and using it to choose distant keywords to facilitate accurate correction. For these corrections, they propose using a set of weights for character substitution that favours adjacent keystrokes, giving them a lower distance as they would be more common typing errors. Insertion and deletion costs, however, are not altered to favour more common insertion and deletion typing errors, such as doubling a letter.

Hirschberg [Hir75] gives a version of the Wagner-Fischer algorithm that reduces the space requirements by only storing the current and previous rows of the dynamic programming matrix. This version initially only produces the length of the longest common subsequence; the subsequence itself can subsequently be extracted by calling the algorithm recursively for sections of the sequences, using divide-and-conquer methodology. Hirschberg also presents two algorithms for the longest common subsequence problem with improved running time for specific cases [Hir77]. Using $k$ to represent the length of the longest common subsequence, one algorithm takes $O(nk + n \log n)$ time, and runs faster than the general algorithm if the two input sequences are very different, specifically if $k \in o(n)$. The second algorithm takes $O(k(m+1-k) \log n)$ time, and is faster than the general algorithm if the difference between the two input sequences is very minor, specifically if $k \in o(m-k)$. This target difference $k$ is an additional input to the algorithm.

The quadratic running time was improved upon by Masek and Paterson [MP80], in an application of a matrix computation technique by Arlazarov, Dinic, Kronod, and Faradzev [ADKF70] (known as the *Four Russians*). They divide the distance computation table $[T[i,j]]_{n \times m}$ into square submatrices with overlapping edges. The values in each submatrix are dependent on its pair of corresponding substrings and the values in the initial edge vectors (top row and left column); the submatrix calculation produces the values in the final edge vectors (bottom row and right column) that are used as the initial vectors for the submatrices immediately beneath or to the right. This function is first calculated for every possible combination of substrings and initial vectors. This enumeration of combinations requires that the strings' alphabet be finite, and that the edit cost function use only integer multiples of some real number. These restrictions produce a finite set of differences $s$ between the corresponding table values for adjacent submatrices. For submatrices with edge vectors of length $p$, the preprocessing takes $O(p^2)$ time for each of $s^{2p} \cdot |\Sigma|^{2p}$ possible submatrices, yielding total preprocessing time $\in O(p^2 s^{2p} \cdot |\Sigma|^{2p})$. For specific input sequences of lengths $n$ and $m$, the distance table is divided into $\frac{nm}{p^2}$ submatrices. Each submatrix can be looked up in the preprocessed set of submatrices in $O(p)$ time, so the distance calculation after preprocessing takes time $\in O(\frac{nm}{p})$. If $p$ is chosen as $p = \lfloor \frac{\log_{s \cdot |\Sigma|} m}{2} \rfloor$, the total processing time required is $\in O(\frac{nm}{\log m})$.

Subsequent experimentation [MP83] by Masek and Paterson that compared running times for the original algorithm and their alternative calculation technique indicated that although their modification improved the running time when analyzed asymptotically, it only produced faster results for sequences longer than 262418 symbols. Their calculations were restricted to a binary alphabet and used a cost function for edit distance. While not necessarily realistic for the spelling correction originally looked at by Wagner and Fischer, these lengths are realistic for DNA sequences.

Sankoff proposed additional constraints on sequence alignment [San72]. He defined the deletion/insertion index of an alignment $A$ as the number of gaps in the matching subsequence that are of different length in one sequence than in the other. If the gaps have the same length, then the substrings spanned by the corresponding gaps can be transformed into each other through direct symbol substitution. If the gaps are of different lengths, a deletion or insertion must have occurred. More formally, the deletion/insertion index ($DI$) of $A$ is the number of successive pairs of pairs $(i_1, j_1), (i_2, j_2) \in A$ such that $i_2 - i_1 \neq j_2 - j_1$. Sankoff gives an algorithm that finds a longest common subsequence alignment $A$ under the constraint $DI(A) \leq q$, for any $q \geq 0$, in time $O(nmq)$. For each $q \in \{0, 1, 2, \ldots\}$, a matrix $V_q = [V_q[i,j]]_{n \times m}$, where $V_q[i,j]$ is the length of the longest common subsequence path $P$ satisfying $DI(P) \leq q$, can be constructed from $V_{q-1}$ in time $O(nm)$.

The longest common subsequence algorithm is used to align only two sequences. For sets of more than two sequences, it can be applied repeatedly to produce *pairwise* alignments for each pair of sequences in the set. Finding the longest common subsequence of all of a set of sequences was shown to be NP-complete by Maier [Mai78]. If the number of sequences is fixed at $k$ with maximum length $n$, their longest common subsequence can be found in $O(n^{(k-1)})$ time, through an extension of the pairwise algorithm [IF92]. The parameterized complexity of the longest common subsequence problem for multiple sequences was examined by Bodlaender, Downey, Fellows, and Wareham [BDFW95] for different parameter combinations. If parameterized by number of sequences, the problem is $W[t]$-hard for all $t$; if parameterized by target common subsequence length, it is $W[2]$-hard. If both parameters are used, the resulting problem is complete for $W[1]$. However, these hardness results are for alphabets of arbitrary size. If the alphabet size is used as another parameter, thus parameterizing LCS by the number of sequences $k$ and the alphabet size $|\Sigma|$, the problem remains $W[t]$-hard for all $t$. If, on the other hand, $|\Sigma|$ is kept constant, the parameterized complexity of LCS parameterized by $k$ is unknown, and looks likely to be a significant open problem. The alphabet size can also be used as a parameter in conjunction with the target subsequence length to make the problem more feasible. Specifically, if both the alphabet size $|\Sigma|$ and the target common subsequence length $l$ are parameters, then the set of sequences can be tested for a common subsequence of length at least $l$ in time $\in O(|\Sigma|^l kn^2)$ by testing each possible sequence of length $l$ against all input sequences.

Producing a *multiple* sequence alignment, one which incorporates all sequences in a set into a single alignment which minimizes some edit cost function, is also NP-complete as the multiple sequence longest common subsequence problem is a special case of it [Mai78]. Another measure of an alignment is its length. An alignment of minimum length produces the *shortest consistent supersequence* of the input sequences. If there are only two sequences, their shortest consistent supersequence is directly related to their longest common subsequence. For $k$ sequences and a fixed binary alphabet, the problem of finding the shortest consistent supersequence is NP-complete [Mai78]. Hallett examines the parameterized complexity of the shortest common supersequence problem [Hal96], and gives the following results. If the number of sequences $k$ is used as a parameter, the shortest consistent supersequence problem is W[1]-hard, and remains hard even if the alphabet size $|\Sigma|$ is also a parameter. If the target supersequence length $l$ is the parameter instead, then the problem is fixed-parameter tractable; there are at most $l^l$ possible supersequences. Fixed-parameter tractable results, however, are useful when the parameter can be made small; since the target subsequence length $l$ must be at least as large as the length of the longest input sequence, this result is not of practical use. Biologically, sequence alignment and finding supersequences and superstrings is a problem involved in building sequences from experimentally produced fragments [Kar93]. Jiang and Timkovsky discuss conditions under which shortest consistent superstrings, a problem inherent in DNA sequencing through hybridization, can be found in polynomial time [JT95].

## 2.2 Structure Analysis

Other research instead focuses on analyzing the structure of a sequence or group of sequences. Structures and other features are searched for in sequences, and used to label those sequences that contain them. If a sequence has a known structure or feature, a database of labeled sequences can be searched for other sequences that are known to have that structure or feature. This work involves specific kinds of sequences as they occur in molecular biology.

If a feature is a specific string that can be a substring of the sequences, it can be searched for using the Boyer-Moore string search algorithm [BM77], which runs in $O(n+m)$ time, where $n$ is the sequence length and $m$ is the pattern or search string length. Motifs for protein families, binding sites, structures, and other features are frequently given by a regular expression; any sequence that contains a string from the regular expression's language as a substring is part of the family, or contains the feature. Searching for strings from a language defined by a regular expression can be done with a scanning algorithm that uses an automaton table [AC75]. If the string pattern is represented by a regular expression of length $m$, the table is built in $O(m)$ time, and used to search a sequence of length $n$ in $O(n)$ time. If the pattern is instead represented by a finite collection of strings whose lengths sum to $l$, the automaton table takes $O(l)$ time to build from the set. This algorithm of Aho and Corasick [AC75] is a generalization of the table-based technique for string pattern matching given by Knuth, Morris, and Pratt [KMP77].

A protein motif can also be characterized using a weight matrix, or profile. Each position in the matrix corresponds to the combination of a particular symbol and a position in the motif, and contains the probability of that symbol occurring at that position. These weights are usually expressed relative to the general background frequency of the symbols. The matrix is usually generated from a multiple alignment of all known sequences in the family that aligns the motif instances together [WP84]. Proteins that have been aligned based on structural information can produce a profile of the amino acid residues in the common structures [GME87, GLE90]; these profiles can be altered to reflect the greater likelihood of edits (insertions, deletions, and substitutions) on the protein's surface than in the core of its structure. A sequence of length $n$ can be compared to the profile expressed by the matrix of a $m$-position motif in $O(nm)$ time. If a family has more than one motif, searching for members of that family can be made more accurate by comparing each sequence to each of the $k$ motif matrices in $O(nmk)$ time [BG97]. Some motifs, however, have a gap or repetition of variable length; using symbol probability matrices does not capture these motifs well since parts of the pattern do not have a fixed position. Allowing insertions and deletions in the alignment of the sequence to the profile matrix [GLE90] allows for these conditions.

Graph theory has applications in matching protein structure and topology. Koch, Kaden, and Selbig apply graph theory to the topology of $\beta$ structures in proteins [KKS92]. They define a graph with one vertex for each $\beta$ strand; a pair of vertices is linked by a sequential edge if they are neighbours along the protein sequence, and a pair is linked by a topological edge if they are adjacent in the protein's structure. If the graph is laid out in the order of its sequential edges, it becomes a sequence with symbols linked by arcs that represent the topological edges. Instead of using arcs, the relative topological distances of the strands are noted. Similarly, the graph can be represented as a sequence of its topological edges and the strands' relative sequential distances are noted. Searching for specific topologies using this representation is done by a substring search. Mitchell, Artymiuk, Rice, and Willett also use graphs to represent the topology of protein structures [MARW89]. They represent both helices and $\beta$-strands by vertices, each labeled with the corresponding structure's linear axis, and link them with edges labeled with the angle and distance between the pair of axes. The graphs representing topological structure are searched for substructures using by finding subgraph isomorphisms under angular and distance tolerance constraints. The subgraph isomorphism algorithm used is due to Ullmann [Ull76], which uses a depth-first tree search method with pruning. Asymptotically, the algorithm is exponential; subgraph isomorphism is NP-complete [Coo71].

Protein structures can be predicted using a method known as *threading*. In threading, a protein of unknown structure is compared to a known structure and evaluated. This comparison is between a sequence and a structure, not the pair of sequences; the sequence that corresponds to the known structure is not looked at. Instead, the pairs of amino acid residues from the new sequence that would be close together in the known structure are examined, and the likelihood of the protein-structure combination is evaluated [JTT92]. Tools are available, such as that described in [MJT96], that align a sequence to each of an entire library of candidate three-dimensional structures, and sorts the resultant models. The affinity of each amino acid for its physical and chemical environment (as provided by the structure) is tested to determine the plausibility of the structure [MJT96].

Finding structures and other features is an important part of using annotation. While definite feature identification still must generally be done or at least confirmed by hand, some modeling, conversion, and prediction are done by computer. Formal methods and grammars have been developed to characterize and model molecular structures [Sea95, YK95], and these grammars can produce tree annotations for molecular sequences. With Dong, Searls also worked on linguistic-based methods for describing the structure of genes and other features with formal grammars [SD93, DS94]. These grammars are a model for the structure, and can be used to search for patterns in protein-encoding DNA sequences and predict the corresponding gene structure.

Some modeling of physical structure is done by looking for topological and hierarchical structure patterns through heuristics for combinatorial pattern discovery [WC+94, WZS95], an application of data mining.

After a multiple alignment is produced, common structures can be found for sets of homologous sequences through a comparative analysis of the phylogeny of the sequences [HK93]. RNA secondary structures, represented as ordered trees, can be compared and classified to detect similar structures and structural mutations [MSOM89]. This classification is independent both of the sequences and of the means used to determine the structures. Wang and Zhang [WZ99] apply thermodynamic folding algorithms to RNA sequences to determine stem substructures. These stems form forests, which then can be matched to the forest of stems from another RNA molecule. Their algorithm has been tested on three sequences of viral RNA and correctly found the main common structural elements [WZ99].

Genetic sequences can also be compared visually. If the comparison is to be done of the genes rather than the detailed sequence of bases, the two-dimensional gel electrophoresis images of DNA can be compared to each other by matching points from the images. Akutsu et al. [AKOF99] give a $O(n^4)$ algorithm for matching two one-dimensional images of length $n$, and show that the two-dimensional problem is NP-complete. They further give a heuristic for this latter problem. In these problems, the point matching is to be tolerant of the non-uniform distortion that tends to occur in gel electrophoresis and scanning.

## 2.3 Parameterized Complexity

Since these problems are combinatorially rich, there are many variants which need to be analyzed. This analysis produces hardness results and algorithms both in the conventional complexity framework and in the parameterized complexity framework [ADF93, DF92, BDFHW94].

Classical polynomial complexity, its reductions, and $NP$-hardness are discussed in depth by Gary and Johnson [GJ79]. Parameterized complexity, on the other hand, was introduced more recently by Downey and Fellows [DF99, ADF93, DF92], and can be used to examine a problem's complexity with respect to its parameters.

Many problems have naturally occurring parameters that describe some aspect of the problem instance, including properties of correct solutions. These parameters can be used to slice a problem $L$ into slices, with one slice $L_k$ for each parameter value $k$. For a polynomial time algorithm with a fixed parameter to show fixed-parameter tractability, the parameter must occur only in the polynomial's coefficient, and not in the degree.

A language $L = \{\langle x, k\rangle \mid k \text{ is the parameter value}\}$ is *uniformly fixed-parameter tractable* (**FPT**) if there is a constant $\alpha$ and an algorithm $\Phi$ such that $\Phi$ decides if $\langle x, k\rangle \in L$ in time $O(f(k) \cdot n^{\alpha})$ where $n = |x|$ and $f : N \rightarrow N$.

If the function $f$ is recursive, then $l$ is *strongly uniformly fixed-parameter tractable.*

$L$ reduces to $L'$ by a uniform parameterized reduction if there is an algorithm $\Phi$ which transforms $\langle x, k\rangle$ into $\langle x', g(k)\rangle$ in time $f(k)|x|^{\alpha}$, and $f, g : N \rightarrow N$ are arbitrary functions, and $\alpha$ is a constant independent of $k$, and $\langle x, k\rangle \in L$ if and only if $\langle x', g(k)\rangle \in L'$ .

As before, if $f$ is recursive then the reduction is termed a *strong uniform parame-*

*terized reduction.*

These parameterized reductions are used with a grouping of problems into classes defined by the complexity of decision circuits. Circuits can have gates of two types; a *small* gate has bounded fan-in, while a *large* gate has unbounded fan-in.

**circuit depth**

The depth of a circuit $C$, $d(C)$, is the maximum number of gates (of any size) on a path in $C$ from input to output.

**circuit weft**

The weft of $C$, $w(C)$, is the maximum number of large gates on a path in $C$ from input to output.

A family of circuits $F$ has bounded depth if there is some constant $h$ such that $\forall C \in F$, $d(C) \leq h$. Similarly, $F$ has bounded weft if there is some constant $t$ such that $\forall C \in F$, $w(C) \leq t$.

A circuit family $F$ is a *decision circuit family* if for all $C \in F$, the circuit $C$ has a single output. For such a decision circuit $C$, $C$ *accepts* input vector $x$ if the output is equal to 1 on input $x$. The *weight* of $x$ is the number of ones in the vector.

For a family $F$ of decision circuits, let the parameterized decision circuit problem be $L_F = \{(C, k) \mid C \in F$ and $\exists$ input $x$ of weight $k$ such that $C$ accepts $x\ \}$.

**Membership in $W[t]$, $W[SAT]$, and $W[P]$**

A parameterized problem $L$ belongs to the class $W[t]$ if $L$ reduces to $L_{F(t,h)}$ for the family $F(t, h)$ of decision circuits with bounded weft $t$ and bounded depth $h$. $L$ belongs to $W[SAT]$ if $L$ reduces to $L_F$ for the family of all decision circuits with gates of fan-out 1. $L$ belongs to $W[P]$ if it reduces to $L_F$ where $F$ is the family of all decision circuits.

This collection of classes can be expressed as a hierarchy, with

$$FPT \subseteq W[1] \subseteq W[2] \subseteq \cdots \subseteq W[t] \subseteq \cdots \subseteq W[SAT] \subseteq \cdots \subseteq W[P]$$

Some common problems known to be $NP$-complete in classical complexity fall into different classes in the $W$ hierarchy when their natural parameters are used. For example, if the parameter is the desired size of the vertex subset, Vertex Cover $\in FPT$, Clique is $W[1]$-complete, and Dominating Set is $W[2]$-complete. The $W$ hierarchy is considered generally orthogonal to the classical polynomial complexity classes; for example, Vapnik-Chervonenkis Dimension is believed not to be $NP$-hard, yet is $W[1]$-complete [DEF93]. This contrasts with Vertex Cover, which is $NP$-complete but is in $FPT$ if its natural parameter, the size of the desired vertex cover, is used.

While usually used to provide an alternative to NP-hardness, the tools of parameterized complexity can also be applied to problems known to be polynomial. Since the length and number of the sequences to be analyzed are both large, the time complexity of algorithmic solutions is critical. The fixed-parameter tractability of a problem can indicate how limiting the range of a parameter can reduce the degree of the polynomial time complexity. This technique seems particularly suited to long sequences with relatively restricted annotation (such as restricted cutwidth, as described in chapter 7).

# Chapter 3

# Types of Annotation

## 3.1 Overview

The purpose of having annotation schemes is to express additional information about a sequence in a way that will enable the information to be analyzed and manipulated along with the sequence. The additional information that we want to represent can be superimposed onto the basic sequence in a variety of ways. This chapter discusses the different types of annotation being considered, what information they can represent, various restrictions on the annotation, and the choice of annotation format. It also discusses some other work that incorporates annotations into sequence similarity and alignment.

All forms of sequence annotation discussed in this thesis involve simple combinatorial objects such as colours, single links between symbols, substrings, and small trees. These objects are superimposed onto parts of the sequence, and can occur anywhere along its length. Each type of annotating object can be used by itself, or combined in layers of different types. Figure 3.1 on page 33 gives an example of each of the three main types of annotation that are investigated.

## 3.2 Colouring

The simplest form of annotation is symbol colouring, where each position in the sequence has a colour as well as a symbol. The annotated sequence is thus a pair of sequences of the same length, one over the symbol alphabet $\Sigma$, and one over the set of colours $C$. In addition to the colours in $C$ there is a blank colour (numerically represented by 0), which indicates that there is no information superimposed onto that symbol. The blank colour 0 is left out of the colour set to ensure that it remains the same, no matter which set of colours is being used to highlight subsequences and substrings. A correctly coloured sequence consists of the original sequence $S_X = x_1 x_2 \ldots x_n$, where $\forall i \in \{1, 2, \ldots, n\}$ $x_i \in \Sigma$, and the corresponding sequence of colours $L_X = l_1 l_2 \ldots l_n$ where $\forall i \in \{1, 2, \ldots, n\}$, $l_i \in C \cup \{0\}$. This pair of sequences can also be viewed as a single sequence over an extended alphabet $\Sigma \times (C \cup \{0\})$, so $\langle S_X, L_X \rangle = \langle x_1, l_1 \rangle, \langle x_2, l_2 \rangle, \ldots, \langle x_n, l_n \rangle$. A symbol-colour pair $\langle x_i, l_i \rangle$ can also be written as $\langle S_X[i], L_X[i] \rangle$ to indicate that it is from $\langle S_X, L_X \rangle$. A language of valid coloured sequences, $\{\langle S_X, L_X \rangle\} = L \subseteq \{\Sigma \times (C \cup \{0\})\}^*$, is called a *same-length regular relation* if $L$ is regular.

Symbol colours can be used to highlight important features of a sequence. Colours can either classify the underlying sequence pieces, or simply indicate the importance of matching them in any alignment or comparison score. For protein sequences, colouring can highlight a family motif, binding site, local features of the secondary structure such as $\alpha$-helices, $\beta$-strands, and turns, or some other significant sequence or substring. Colours can be used to indicate structural features for DNA or RNA sequences as well. For these genetic sequences, colours can also be applied to non-structural features of the code, indicating the function of pieces such as gene promotors and anchors. Exon sequences, which provide the code for protein production, can be coloured at different levels. A colour can indicate

- that the symbol is part of an exon region
- what protein is being coded for, or to what group the protein belongs
- what amino acid the 3-symbol codon containing the current symbol corresponds to

Many of these biological applications also apply to substring colouring. Same-length regular relations are also used in computational linguistics, particularly phonology (as in [BE94]). Colours can be used to encode phonological production and rewrite rules.

Some examples of how sequence colouring can be restricted are:

- eliminating certain symbol-colour pairs or colour juxtapositions
- requiring that the coloured regions be sparse, ie., less than a certain percentage of the sequence can be coloured
- bounding the length of contiguously coloured pieces
- requiring that each coloured substring be from an associated language (particularly regular languages for regular relation applications)
- providing a pattern (regular or otherwise) for the sequence of colours

and these restrictions can be used alone or in any combination to delineate the set of valid coloured sequences.

## 3.3 Substrings and Trees

Colouring substrings is very similar to colouring symbols, except that the colour is applied to entire substrings of the sequence. This colouring thus both marks the substring as a contiguous piece of the sequence, and classifies it using the colour. A sequence with substring annotations over the alphabet $\Sigma$ and the colour set $C$ is a sequence of pairs $S_X^1 = \langle S_X[1], L_X[1]\rangle\langle S_X[2], L_X[2]\rangle \ldots \langle S_X[h], L_X[h]\rangle$ where

- $\forall i \in \{1, 2, \ldots, h\}$, $S_X[i] \in \Sigma^*$
- $\forall i \in \{1, 2, \ldots, h\}$, $\exists c \in C$ such that $L_X[i] = c$, or $L_X[i] = 0$
- the original sequence $S_X = S_X[1]S_X[2]\ldots S_X[h]$ (so $S_X$ is the concatenation of the individual substrings $S_X[i]$)
- $S_X$ has a same-length sequence of colours $L'_X = c_1^{|S_X[1]|}c_2^{|S_X[2]|}\ldots c_h^{|S_X[h]|}$.

When more than one sequence is being examined, the subscript $X$ in $S_X$ and $L_X$ is replaced by a number indicating which sequence is being referred to. The sequence of substrings $S_X^1$ can be viewed as a sequence of ordered trees of height 1, where each tree $M_X[i]$ has a root node containing its colour $c_i$ with the individual symbols in the substring $S_X[i]$ as children. These children appear in the tree $M_X[i]$ in the same order as in the substring $S_X[i]$. Note that substrings coloured blank should be of length 1 (individual symbols only) since there is no information to group them together into a substring.

Since $S_X^1$ is itself a sequence, it can also be annotated with substrings. The above definition of one level of substring annotation can be used repeatedly to produce any number of levels. These layers of substrings can also be viewed as a tree. For $k$

levels of substring annotation, given a family of colour sets $\{C_j \mid j \in \{1, 2, \ldots, k\}\}$ and the original sequence $S_X$, an annotated sequence $S_X^k$ is defined recursively by

$$S_X^k = \begin{cases} \langle S_X[1], L_X[1]\rangle\langle S_X[2], L_X[2]\rangle \ldots (S_X[h], L_X[h]) & \text{if } k = 1 \\ \langle S_X^{k-1}[1], L_X^{k-1}[1]\rangle\langle S_X^{k-1}[2], L_X^{k-1}[2]\rangle \ldots \langle S_X^{k-1}[h], L_X^{k-1}[h]\rangle & \text{otherwise} \end{cases}$$

where

- $\forall j \in \{2, 3, \ldots, k\}$, $\forall i \in \{1, 2, \ldots, |S_X^j|\}$, $\exists c \in C_j$ such that $L_X^{j-1}[i] = c$, or $L_X^{j-1}[i] = 0$
- $\forall i \in \{1, 2, \ldots, h\}$, $\exists c \in C$ such that $L_X[i] = c$, or $L_X[i] = 0$
- $\forall j \in \{2, 3, \ldots, k\}$, $S_X^j = S_X^{j-1}[1]S_X^{j-1}[2] \ldots S_X^{j-1}[|S_X^j|]$
- $S_X = S_X[1]S_X[2] \ldots S_X[h]$

The essential difference between symbol colouring and substring colouring is that all the symbols in a substring are treated as a single contiguous unit. The symbols in a subsequence, even if contiguous, are instead treated as a collection of symbols. An algorithm that aligns a substring-annotated sequence must match a single substring to another single substring. Many applications of symbol colouring are also potential applications of substring colouring; which type of colouring should be used depends on how you want to use the annotated sequences. Any restrictions given for symbol colouring can also apply to substring colouring. Some, such as requiring that all substrings of each colour be from a language associated with that colour, are more appropriate for substring colouring. Substring colouring can be used to highlight textual words, grammatical parts or constructs, and other complete contiguous features of the sequence. For genetic sequences, gene promotors and anchors can be coloured separately and linked together to form a higher level substring. Each

3-symbol codon can be a substring, coloured with the corresponding amino acid. Using substring colouring instead of sequence colouring for these applications requires that the alignment at the basic sequence level be derived from the alignment at the higher levels of the annotation. Thus marking codons as substrings means that comparison is done first at the level of the protein sequence that the annotation represents, and then at the DNA level within that original comparison.

Substring and symbol colouring can be mixed on the same level or on separate levels by considering a coloured symbol as a substring of length 1.

## 3.4 Arcs

Another object that can be used in annotations is the arc. An arc is a link or edge that joins two symbols of the sequence. A sequence $S_X = x_1 x_2 \ldots x_n$, where $\forall i \in \{1, 2, \ldots, n\}$, $x_i \in \Sigma$, has a corresponding set of arcs $P_X \subset n \times n$ where $\forall (i_1, i_2) \in P_X$, $i_1 < i_2$. The order of an arc's two endpoints is thus consistent with the order of the sequence.

An arc can be applied to a sequence to represent binary relations between sequence symbols or sequence substrings. Arcs can be used to join symbol pairs that are chemically bonded in the represented biological sequence. This application is particularly relevant to RNA sequences, whose chemical bonds between base pairs can be described by overlaying these arcs onto the sequence. For a higher-level sequence of substrings, arcs can join related features, such as gene anchors with promotors.

Arc placement for a sequence can be restricted by

- allowable endpoint pairs, eg. C always linked to G and vice versa

- each symbol can only be linked once
- arcs cannot cross each other
- describing permitted arc nesting structure with a regular or context-free language
- limiting the number of arcs "active" at any position in the sequence

either alone or in combination.

A sequence with coloured substrings can also be annotated with arcs at any level. Using arcs to link entire substrings instead of a large set of parallel arcs can reduce the number of arcs used, which in turn can greatly reduce the time required to compare the annotated sequences. The linked substrings can be superimposed onto a mixed substring/symbol colouring, where endpoints of the parallel arcs are coloured as symbols, and any other symbols and substructures are coloured as substrings. These substrings can in turn be composed of pairs of linked substrings.

## 3.5 Annotation Format

In the preceding sections, the different types of sequence annotation are defined mathematically. When they are used, their format needs to be chosen to enable efficient and effective manipulation by both computers and humans. Since sequences are analyzed largely without annotation, an annotated sequence database should consist of an existing sequence database with a separate and parallel database of annotations, linked by a common key and order. This second database may also include the basic sequence to enable people to understand the meaning of the an-

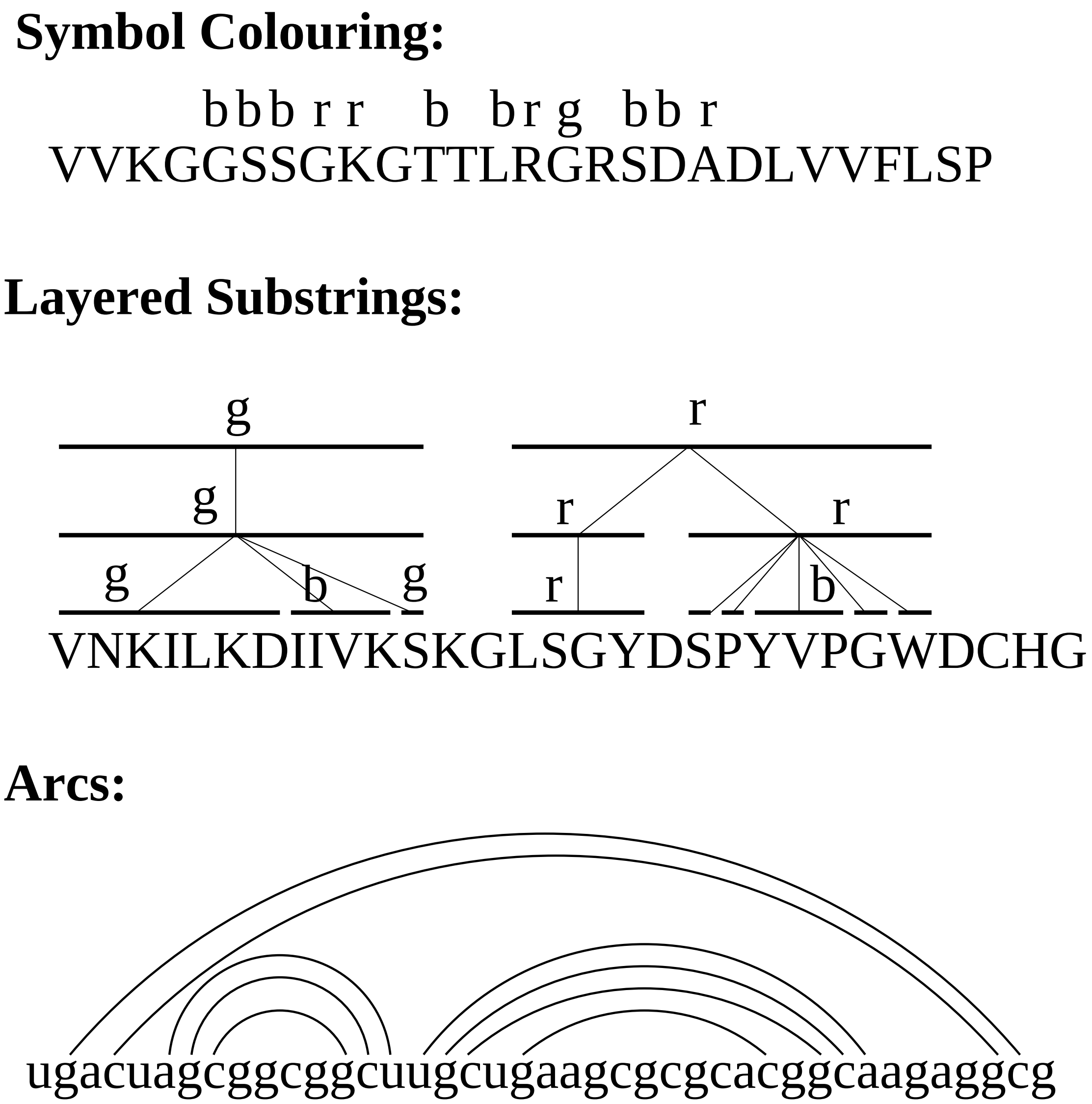


Figure 3.1: Examples of different types of annotation

notated sequences when inspecting the file. The original sequence database should be kept unchanged so it can be analyzed using methods for unannotated sequences.

Symbol colouring is best stored, viewed, and processed as a sequence of colours of the same length as the basic sequence. If the colour sequence is sparse, containing long strings of blanks, it should be stored as an ordered list of the coloured strings. Each member of the list includes its range of positions in the sequence followed by the sequence of colours occurring at that range.

A substring or layered substring annotation could also be stored as a sequence or several sequence of colours, but this representation can lose the border between identically coloured substrings. To preserve these boundaries, store the substrings of each layer as an ordered list, with each element of the list giving the substring's colour and range of sequence positions. For sequences with multiple levels of substrings, each level is listed separately, from highest to lowest level. If arcs are added to any of these levels, their list is also included in that level. While these lists are an effective storage structure, they need to be transformed into a forest of trees for efficient computer manipulation. Each tree in the forest starts with a substring from the highest level of annotation as its root, with all substrings within that range at the next level down as its children, and so on. The leaves of these trees are the symbols from the basic sequence. This data structure can also be used visually, or instead each level can be displayed as a sequence of colours with inserted substring terminators.

Arc annotation is also best stored as an ordered list, ordered primarily by starting endpoint and secondarily by final endpoint for arcs with a common first endpoint. For efficient processing and graphical viewing, the arcs need to be superimposed onto the sequence. This must enable the arcs to be accessed from their endpoint symbols and followed from endpoint to endpoint.

If the arcs do not cross or share endpoints, then they also can be represented and stored as a sequence of balanced starting and final endpoint markers, much like a sequence of spaced balanced parentheses. For a basic sequence $S_X$, the sequence of balanced arc endpoints is $Q_X \subset \{(,), blank\}^{|S_X|}$.

## 3.6 Related Work with Annotated Sequences

While the work discussed in chapter 2 does not incorporate annotations into the basic sequence analysis, some other recent work does analyze both sequence and annotation together. Part of this research involves superimposing this information onto the sequence and analyzing the resulting annotated sequence; however, this work has been limited in its use of the annotated information.

In particular, arcs between sequence symbols are used in an annotation to represent bonds in RNA secondary structure. Bafna, Muthukrishnan, and Ravi [BMR96] present an algorithm for computing a sequence distance with additional weighting to increase the similarity score when arcs are matched. The algorithm produces an alignment $A$ of strings $S_1$ and $S_2$ that maximizes the sum

$$l(A) = \sum_{1 \le k \le m'} \gamma(S_1[k - gap[1,k]], S_2[k - gap[2,k]]) \;\; +$$
$$\sum_{1 \le k < l \le m'} \delta(k - gap[1,k], l - gap[1,l], k - gap[2,k], l - gap[2,l])$$

where the alignment $A$ is given by a $2 \times m'$ matrix, where the first row contains $S_1$ and the second row contains $S_2$, in order with inserted spaces. Let $i_1 = k - gap[1,k]$, $i_2 = l - gap[1,l]$, $j_1 = k - gap[2,k]$, and $j_2 = l - gap[2,l]$, making their notation consistent with that defined in section 3.4. The function $\gamma(a,b)$ is the weight of matching symbol $a$ with $b$ (equivalent to $w(a,b)$ in the longest common subsequence algorithm), while $\delta(i_1, i_2, j_1, j_2)$ is the weight of matching the arc $(i_1, i_2)$ from $P_1$ with $(j_1, j_2)$ from $P_2$, if both these arcs exist. The $gap[x, k]$ function produces the number of spaces in row $x$ before position $k$, and is used to determine the location in the alignment matrix of the symbol in position $k$ of sequence $S_X$. Each sequence $S_X$ has a corresponding set of arcs $P_X$.

AlignRNA algorithm:

for all intervals $(i_1, i_2)$ with $1 \leq i_1 < i_2 \leq m$, and $(j_1, j_2)$ with $1 \leq j_1 < j_2 \leq n$, examined in increasing order of width, compute

$$Align[i_1, i_2, j_1, j_2] = \max \begin{cases} Align[i_1, i_2 - 1, j_1, j_2] + \gamma(S_1[i_2], blank) \\ Align[i_1, i_2, j_1, j_2 - 1] + \gamma(blank, S_2[j_2]) \\ Align[i_1, i_2 - 1, j_1, j_2 - 1] + \gamma(S_1[i_2], S_2[j_2]) \end{cases}$$

if $\exists i_3, j_3$ where $i_1 \leq i_3 < i_2$ and $j_1 \leq j_3 < j_2$, such that $(i_3, i_2) \in P_1$ and $(j_3, j_2) \in P_2$, compute

$$Align[i_1, i_2, j_1, j_2] = \max \begin{cases} Align[i_1, i_2, j_1, j_2] \\ Align[i_1, i_3 - 1, j_1, j_3 - 1] \ + \\ \qquad Align[i_3 + 1, i_2 - 1, j_3 + 1, j_2 - 1] \ + \\ \qquad \delta(i_3, i_2, j_3, j_2) + \gamma(S_1[i_2], S_2[j_2]) \end{cases}$$

Note that the alignment is not actually generated by the algorithm given, though it can be modified to create the alignment as well as finding the longest common subsequence length.

The weighting $\delta$ is only looked at if the arcs match; there is no attempt to give any weight, positive or negative, to matching the endpoints without matching the induced arc. This differs from the approach discussed in chapter 7, which cannot align both of the endpoints of an arc on one sequence unless the corresponding positions on the other sequence are also linked by an arc. The difference between these two approaches changes the meaning of an arc not being present. For the

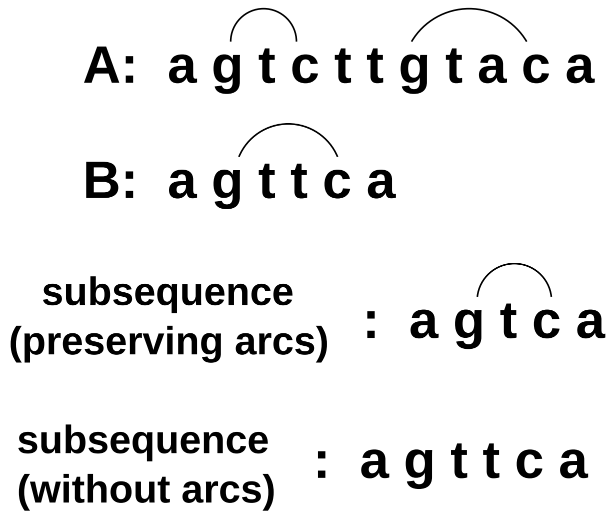


Figure 3.2: Aligning arc-annotated sequences: without arcs (AlignRNA) versus preserving induced arcs

AlignRNA algorithm, the lack of an arc in the representation means that there may or may not be an arc, since it will align exactly the same with a linked pair as with an unlinked pair. On the other hand, the arc-preserving longest common subsequence discussed in chapter 7 will not align an unlinked pair with a linked pair, and can be extended to incorporate arc mismatch penalties that use the arc weight function $\delta$ completely. Figure 3.2 illustrates an example of the different longest common subsequence results. By not detecting arc mismatch, the AlignRNA algorithm avoids the algorithmic time complexity caused by having the match of one arc endpoint affect the potential match of the other.

The AlignRNA algorithm runs in time $O(n^2m^2)$, where $n$ and $m$ are the lengths of the two sequences. This time complexity is not very useful for long sequences. The algorithm's behaviour is independent of the actual arc structure.

While it also using arcs to represent RNA bonds, the work of Corpet and Minchot [CM94] and Sankoff [San85] only allows the matching of entire substructures to aid in aligning the sequences. Similarly, substring colouring based on known secondary structure is used by [RSDC94] to assist in the prediction of the structure of a related sequence. Information from both protein structural and sequence similarity is merged in attempts to balance the disparate information and correlate the results into an overall measure of similarity [PA92, MDM95]. Lenhof et al. include pseudoknots, RNA secondary structures produced by pair bonds that cross, in their graph-based work on RNA sequence alignment [LRV98]. Their algorithm, however, aligns sequences where only one sequence has an associated structure. Like the work before it, the links between base pairs are used to enhance the alignment by contributing to its score.

Zhang et al. also address aligning RNA sequences with respect to their pair bonds in [ZWM99]. They treat each base pair as a unit, so a base pair from one sequence must be matched to a base pair in another sequence. To compute the edit distance between two RNA structures, they allow delete, insert, and relabel operations which can be applied either to a base pair or a single unpaired base. If bonds are allowed to cross (producing a structure with pseudoknots), they show that the problem is NP-complete. If there are no crossing pairs, then they reduce the problem to a tree-edit distance discussed in [Zha98]. If only one RNA sequence has crossing pairs, an algorithm can find the best match between the two structures and eliminate the unmatched crossing bonds in time $O(n^3 \log n)$ [ZWM99].

One variant of the fundamental alignment algorithm attempts to favour protein sequence alignments that match motifs by enhancing the scores for continuous pieces of exact or close matches [Tay94]. Taylor proposes an altered alignment measure that increases the scores for short pieces of contiguous matches. The algorithm maintains a running product of the current match's score and those of preceding adjacent matches, and adds it to the weight gained by matching those positions.

This scheme encourages alignments which match contiguous pieces, while reducing the potentially overpowering effect of long sequences of adjacent matches. Taylor uses this algorithm to produce improved results for aligning protein sequences with short motifs [Tay94]. His results also show a decrease in the biased algorithm's dependence on the value of the gap penalty. However, not all contiguous matched pieces are necessarily motifs. The algorithm is biased towards matching any such common substring, which may not be rare or discriminatory enough to be a motif. Since this algorithm does not actually use the motif, it can instead match any similar or identical pieces and hypothesize that the pieces are motif instances. Taylor's product-based scheme also requires an entire substring to be closely matched; any gaps within the substring's alignment resets the running product to zero.

The existing work that analyzes primary sequence information in tandem with secondary information is limited and quite specific to its applications. The existing tools which include labeling are specific to structure prediction, and do not enable other information to be annotated. Some string search and alignment tools can be manipulated into using additional labeling instead of just the basic sequence. For example, sequences with symbol colouring can be forced into a tool that compared non-coloured sequences by merging each symbol and colour pair into a single joint symbol. Any match weight table would have to be redefined accordingly for the colour-extended alphabet, and both the original sequences and meaning of the colours would not be apparent to people viewing the combined sequence. This user manipulation is indirect, seldom done, and should not be necessary. Flexible algorithms and tools need to be developed to manipulate and analyze annotated sequences that can incorporate a variety of information, and produce alignments based on both sequence and annotation.

# Chapter 4

# Annotation Verification

Annotation verification involves checking a sequence's annotation to ensure that it conforms to all the restrictions for the given annotation scheme. The annotation's format can represent information that is not a valid annotation for any sequence in the scheme, independent of specific restrictions. Annotations thus need to be checked to ensure that they encode some valid annotation, and then can be checked further to ensure that they are a valid annotation for their specific sequence. Algorithms for checking annotations would be applied to annotations that are user-defined or have an external source, and can also assist in creating and interpolating annotations. This chapter investigates the problems of enforcing certain annotation restrictions, and gives algorithms that solve some of these problems. When possible, finite automata are used as verifiers. This focus on automata emphasizes the limited resources needed to verify the annotated sequences, and the regularity of classes of validly annotated sequences. Where automata are not possible due to the need to measure a length or match arc endpoints, linear time algorithms are given.

## 4.1 Colour Restrictions

A coloured sequence from a colouring scheme that includes colour restrictions needs to be examined to make sure that it conforms to the chosen restrictions. As discussed in chapter 3, sequence colouring can be restricted by possible symbol-colour pairs, colour juxtapositions, allowed range for the length of a colour segment, and sparseness of each colour.

Before any of these restrictions are checked, the sequence of colours must be verified as an annotation in that sequence scheme; it must be of the same length as the sequence it annotates, and it must also use only colours from the set allowed by the representation. These characteristics are verifiable in linear time by measuring the length of both sequences and matching each colour against the set of allowed colours $C$.

Once the sequence of colours has passed this check, it can be scanned for the different colour restrictions. For restrictions on allowed colour-symbol pairs, the pair at each position in the coloured sequence can be checked in linear time by a finite automaton that recognizes all strings in $(\Sigma \times C) - \{\langle x, c\rangle \mid \text{symbol } x \text{ may not have colour } c\}$. Illegal colour juxtapositions, eg. if a red symbol may not be followed by a green symbol, can also be checked for by a finite automaton by having a different state for each colour; all transitions whose symbol has that colour would have the corresponding state as its destination. Verifying length restrictions is linear, and can be done with a single pass through the entire coloured sequence. The length of each coloured piece is computed, and compared against the allowed range of lengths for that colour. The sparseness of colours can also be checked in linear time by counting the cumulative length of each colour.

## 4.2 Valid Substrings

Any annotated sequence from a layered coloured substring annotation scheme must be checked for restrictions at every level. It must also be checked to verify that it actually contains properly layered substrings, since the annotation format can encode invalid layers. To be valid, the substring boundaries at each level can only occur at positions that are also substring boundaries at the level immediately below. While the mathematical representation and the tree representation of the annotation enforces this requirement, the file representation does not. This representation, with an ordered list of substring colours and ranges for each level of the annotation, can encode overlapping substrings that are not legitimate annotations.

Verifying that an ordered set of ordered lists corresponds to a legal coloured substring annotation has two stages; individual layers and pairs of adjacent layers need to be checked. First, each list, which corresponds to a layer of substrings, must be checked to ensure that the substrings are ordered, do not overlap, and cover the entire length of the sequence. The following simple algorithm performs this first stage of verification.

**Algorithm 4.1.** Individual Layer Verification.

Let $n$ be the length of the sequence. For each member of the list, $first$ is the index position of the start of the substring and *last* is the last position in the substring.

start at the beginning of the list.
if $first \neq 1$, return *false*

```
while not at the end of the list
        let old = last
        advance to next member of list
        if first ≠ old + 1 return false
if at end of list
        if last ≠ n return false
return true
```

Secondly, each pair of adjacent layers must be checked to ensure that each substring in the lower layer is entirely part of exactly one substring in the upper layer.

**Algorithm 4.2.** Adjacent Layer Containment Verification.

Let *list1* refer to the higher level list, and *list2* refer to the lower level list. For each member of each list, $first$ is the start of the substring and $last$ is the last position in the substring.

```
start at the beginning for list1 and list2.
while not after the end of list2
        if first of list2 > last of list1
           advance to next member of list1
        else if last of list2 > last of list1
                return false
        advance to next member of list2
return true
```

## 4.3 Coloured Regular Languages

If colours are used in the annotation to highlight words from regular languages, a sequence with a valid annotation can be verified with a finite automaton. To show that this verification can be done in linear time using finite automata, the languages of validly annotated sequences are shown to be finite state recognizable. The use of finite automata to recognize valid annotations means that not only can the verification be done in linear time, it can be done without having to store the annotated sequence internally. This can be particularly useful for long genetic sequences, for which finite automata would still only have to use some constant amount of space; the entire sequence does not have to be stored.

Let $L$ be any regular language on an alphabet $\Sigma$ (so $L \subseteq \Sigma^*$). Let $\mathcal{R}$ be the set of all regular languages on $\Sigma$. Note that $\mathcal{R} \subseteq \mathcal{P}(\Sigma^*)$, where $\mathcal{P}(\Sigma^*)$ is the powerset of the set of all words over the alphabet $\Sigma$. If $e$ is some regular expression, let $L(e)$ be the regular language defined by $e$. Similarly, if $M$ is some finite automaton, let $L(M)$ be the regular language that it accepts.

Let $C$ be some set of colours to be used to annotate sequences, and let $0 \notin C$ be the blank colour used to indicate non-highlighted symbols. We now explore some features of *coloured languages*, which are subsets of $\Sigma \times C$. The colours in $C$ are used to highlight strings from regular languages that appear in annotated sequences. While an entire sequence may not be in the specific language, the coloured pieces are; different colours can also be used to highlight strings from different regular languages. Note that since each symbol is coloured exactly once, the sequence of colours $p$ is always the same length as the basic sequence of symbols $w$.

To start colouring words from regular languages, we first convert an uncoloured language into a language with a single colour applied to the entire string. Let

$H_0 : \mathcal{R} \times C \to \mathcal{P}(\Sigma \times (C \cup \{0\}))$ be defined by $H_0(L, c) = \{\langle w, c_{|w|}\rangle \mid w \in L\}$.

**Lemma 4.3.** If $L$ is a regular language over alphabet $\Sigma$, then $H_0(L, c)$ (for any $c \in C$) is a regular language over alphabet $\Sigma \times (C \cup \{0\})$.

**Proof:** Since $L$ is a regular language, then there must be some regular expression $e$ such that $L = L(e)$. Produce $e'$ from $e$ by substituting $\langle a, c\rangle$ for each occurrence of $a$ in $e$, $\forall a \in \Sigma$. This altered expression $e'$ is still a regular expression, so $L(e') = H_0(L, c)$ is a regular language. □

Instead of the entire string being from some regular language, it can instead contain a highlighted subsequence where that subsequence is from the language. For a regular language $L$, the $L$-subsequence annotated language $H_1 : \mathcal{R} \times C \to \mathcal{P}(\Sigma \times (C \cup \{0\}))$ is defined by $H_1(L, c) = \{\langle w, p\rangle \mid \langle w', p'\rangle$ is $\langle w, p\rangle$ with all $\langle x, 0\rangle$ pairs removed, and with $\langle w', p'\rangle \in H_0(L, c)$ $\}$.

**Theorem 4.4.** If $L$ is a regular language over the alphabet $\Sigma$, then $H_1(L, c)$ (for any $c \in C$) is a regular language over the alphabet $\Sigma \times (C \cup \{0\})$.

**Proof:** Let $M$ be a finite state machine accepting $L$, $M = (Q, \Sigma, \delta, q_0, F)$. Construct $M' = (Q, \Sigma \times (\mathcal{C} \cup \{0\}), \delta', q_0, F)$ where $\delta' : Q \times (\Sigma \times (\mathcal{C} \cup \{0\})) \to Q$ is defined by:

$$\delta'(q, (x, i)) = \begin{cases} \delta(q, x) & \text{if } i = c \\ q & \text{otherwise} \end{cases}$$

This modified automaton $M'$ accepts $H_1(L, c)$. □

Coloured languages where an entire coloured substring is from a given regular language are now examined. Let the single $L$-substring annotated language $H_2 : \mathcal{R} \times C \to \mathcal{P}(\Sigma \times (C \cup \{0\}))$ be defined by $H_2(L, c) = \{\langle w, p\rangle \mid p = 0^r c^s 0^t$, $w = xyz$ with $|x| = r$, $|y| = s$, $|z| = t$, and $y \in L$ $\}$. So for any regular language $L$ and colour

$c$, $H_2(L, c)$ is the set of all strings such that a single contiguous instance of a word from $L$ is highlighted using colour $c$.

**Theorem 4.5.** If $L$ is a regular language over alphabet $\Sigma$, then $H_2(L, c)$ (for any $c \in C$) is a regular language over alphabet $\Sigma \times (C \cup \{0\})$.

**Proof:** From Lemma 4.3, $H_0(L, c)$ is a regular language and has regular expression $e'$. Then $H_2(L, c) = L((\Sigma \times \{0\})^* e' (\Sigma \times \{0\})^*)$, and is a regular language. □

Just as a single instance of a word from a regular language can be highlighted, so can several instances of potentially different words from the same regular language. Let the multiple $L$-substring annotated language $H_3(L, c)$ be the language of annotated strings $\langle w, p \rangle \in (\Sigma \times \{0, c\})^*$ such that each contiguous string of $c$s highlights an instance of a string from $L$. Note that these strings can be adjacent.

**Corollary 4.6.** If $L$ is a regular language over alphabet $\Sigma$, then $H_3(L, c)$ (for any $c \in C$) is a regular language over alphabet $\Sigma \times (C \cup \{0\})$.

**Proof:** Since $H_0(L, c)$ is a language represented by the regular expression $e'$, then the language $H_3(L, c) = L([(\Sigma \times \{0\})^* e']^* (\Sigma \times \{0\})^*)$, and is also regular. □

The finite-state verification of coloured regular languages will be used in the next chapter to show the finite-state searchability of the regular languages; that strings from the languages can be found using a pair of finite-state transducers, as defined in section 5.3. To enable the longer strings to be found and coloured (instead of only trivial ones), it can be useful to require the annotation to colour strings in a way that does not leave out any valid extensions. Therefore, languages whose coloured sections are maximal and cannot be extended any further are examined.

We first examine the set of highlighted *suffix-closed* strings from a regular language $L$. These strings are those with a highlighted instance of a string from $L$ where the highlighted segment of the string cannot be extended any further (to produce a different string from $L$). Let the $L$-suffix-closed annotation language be $H_4(L,c) = \{\langle w,p\rangle \mid p = c^s0^t,\ w = yz \text{ with } |y| = s, |z| = t,\ y \in L, \text{ and } \forall i, 0 < i \leq t,\ yz_1z_2\ldots z_i \notin L\ \}$. Note that this is not entirely a substring annotation, as the highlighted string must start at the beginning of the sequence.

**Lemma 4.7.** If $L$ is a regular language of alphabet $\Sigma$, then $H_4(L,c)$ (for any $c \in C$) is a regular language over alphabet $\Sigma \times (C \cup \{0\})$.

**Proof:** Let $M$ be a deterministic finite automaton that accepts $L$, so $L = L(M)$. We need to construct a finite automaton $M'$ that accepts $H_4(L,c)$. Note that $M$ must be deterministic for this construction of $M'$ to work. The constructed $M'$ does not need to be deterministic, though the following construction does produce a deterministic machine. Figure 4.1 contains an example of the parts of this construction.

Since our finite automaton $M$ is deterministic, $M = \langle Q, \Sigma, q_o, F, \delta\rangle$ where $\delta$ is a function. Let $M^0$ be a copy of $M$ altered to accept $H_0(L,c)$. Thus $M^0 = \langle Q^0, \Sigma \times \{0,c\}, q_0^0, F^0, \delta'\rangle$ where $Q^0$ is a copy of the set of states $Q$, $q_0^0 \in Q^0$ is a copy of the start state $q_0$, and $F^0 \subseteq Q^0$ is a copy of the set of accept states $F$. For each state $q \in Q$, let $q^0$ refer to the copy in $Q^0$ of state $q$. We define $\delta'$ by:

$$\delta'(q^0, \langle a,c\rangle) = \delta(q,a)^0 \quad \forall a \in \Sigma \ \ \forall q \in Q$$

with $\delta'(q^0, \langle a,0\rangle)$ to be defined later.

Also let $M^f$ be a copy of $M$, coloured with 0 and altered to reject $H_1(L,0)$. So $M^f = \langle Q^f, \Sigma \times \{0,c\}, q_0^f, Q^f - F^f, \delta'\rangle$ where $Q^f$ is a copy of the set of states $Q$, and $F^f \subseteq Q^f$ is a copy of the original set of accept states $F$. The start state copy $q_o^f$ will not be used as a start state (and is thus only included to adhere to the definition of a finite automaton). For each state $q \in Q$, let $q^f$ refer to the copy in $Q^f$ of state $q$.

Note that the set of accept states for $M^f$ is $Q^f - F^f$, instead of $F^f$. We defined $\delta'$ by:

$$\delta'(q^f, \langle a, 0\rangle) = \delta(q, a)^f \quad \forall a \in \Sigma \;\; \forall q^f \in Q^f - F^f$$

$$\delta'(q^f, \langle a, 0\rangle) = q^f \quad \forall a \in \Sigma \;\; \forall q^f \in F^f$$

with $\delta'(q^f, \langle a, c\rangle)$ to be defined later. So we now have two machines, one that accepts $L$ that has been coloured with $c$, and one that accepts everything except strings that start with an instance of $L$ (coloured with 0, the blank colour). Now we need to join these machines together. Let $R$ be a "reject" state that is not part of $Q^0$ or $Q^f$. Then let

$$\delta'(R, \langle a, c\rangle) = \delta'(R, \langle a, 0\rangle) = R \quad \forall a \in \Sigma$$

$$\delta'(q^0, \langle a, 0\rangle) = R \quad \forall a \in \Sigma \;\; \forall q^0 \in Q^0 - F^0$$

$$\delta'(q^f, \langle a, c\rangle) = R \quad \forall a \in \Sigma \;\; \forall q^f \in Q^f$$

and join these machines together with

$$\delta'(q^0, \langle a, 0\rangle) = \delta(q, a)^f \;\; \forall a \in \Sigma \;\; \forall q^0 \in F^0$$

into a single machine $M'$. This new, merged machine $M'$ is represented by $\langle Q', \Sigma \times \{0, c\}, q'_o, F', \delta'\rangle$ where the set of states is $Q' = Q^0 \cup Q^f \cup \{R\}$, the start state is $q'_o = q^0_0$, and the set of accept states is $F' = F^0 \cup (Q^f - F^f)$. The transition function $\delta'$ has already been defined.

**Claim.** The language accepted by $M'$ is $L(M') = H_4(L, c)$.

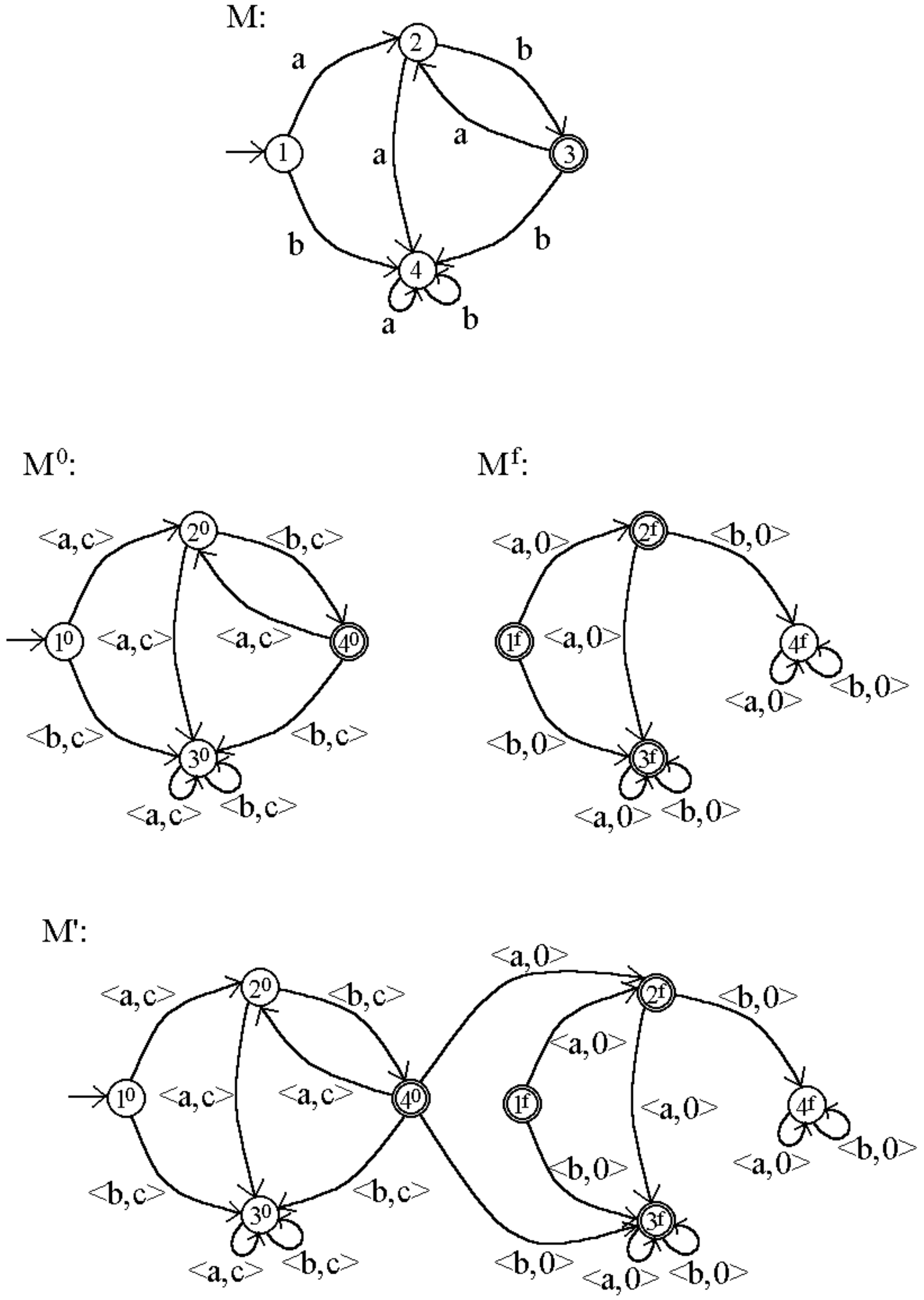


Figure 4.1: Steps of finite automaton construction for the suffix-closed language $H_4(L(a(ba)^*b))$

**Proof of Claim:**

*Case 1.* Consider $\langle w, p\rangle \in H_4(L, c)$. By the definition of $H_4$, $w = yz$ with $p = c^{|y|}0^{|z|}$, and $y \in L$ with $yz_1z_2 \ldots z_i \notin L\ \forall i, 0 < i \leq |z|$.

Run $M'$ with input $\langle w, p\rangle$. Since $y \in L$, $M'$ will be at some accept state $q_f \in F^0$ after the first $|y|$ pairs ($\langle y, c^{|y|}\rangle$) have been processed. If $|z| = 0$ then the machine stops here and accepts $\langle w, p\rangle$.

If $|z| > 0$, the next transition $\delta'(q_f, \langle z_1, 0\rangle)$ moves the machine into the set of states from $M^f$. Since $yz_1z_2 \ldots z_i \notin L\ \forall i, 0 < i \leq |z|$, all of the states from $M^f$ that are reached while $\langle z, 0^{|z|}\rangle$ is processed correspond to non-accept states of $M$, and are thus accept states of $M'$.

*Case 2.* Consider $\langle w, p\rangle \notin H_4(L, c)$. If $p \notin L(c^*0^*)$ then $\langle w, p\rangle$ will be rejected. So let $w = yz$ with $p = c^{|y|}0^{|z|}$.

If $y \notin L$, the computation of $M'$ will be in a non-accept state $q_y$ after $\langle y, c^{|y|}\rangle$ has been processed. If $|z| = 0$, the machine stops there and rejects $\langle w, p\rangle$. Otherwise, the next transition $\delta'(q_y, \langle z_1, 0\rangle) = R$ will send $M'$ into the reject state $R$ (from which there is no escape).

If $y \in L$, then since $\langle w, p\rangle \notin H_4(L, c)$ there must be some $i$, $0 < i \leq |z|$ such that $yz_1z_2 \ldots z_i \in L$. If there is more than one such $i$, we look at the smallest of them. After $\langle y, c^{|y|}\rangle$ has been processed, the machine is at an accept state $q_f$, and the next transition $\delta'(q_f, \langle z_1, 0\rangle)$ moves the machine into the states from $M^f$. After $\langle z_1, 0\rangle\langle z_2, 0\rangle \ldots \langle z_i, 0\rangle$ has been processed, $M'$ is at some state $q_z$ in $M^f$ that corresponds to an accept state of $M$. Thus $Q_z$ is not an accept state of M', and $\delta'(q_z, \langle a, 0\rangle) = q_z\ \forall a \in \Sigma$, so $M'$ will remain in this state.

Thus the finite automaton $M'$ accepts exactly those strings in $H_4(L, c)$. $\square$

Once we can ensure that the highlighted string is suffix-closed, we can add any unhighlighted symbols to its start. Let the single $L$-suffix-closed substring annotation language be $H_5(L,c) = \{\langle w,p\rangle \mid w = xyz,\ p = 0^r c^s 0^t,\ |x| = r,\ |y| = s,\ |z| = t,\ y \in L, \text{ and } \forall i, 0 < i \leq t, yz_1z_2\ldots z_i \notin L\ \}$, which is the language of such annotated strings.

**Corollary 4.8.** If $L$ is a regular language over alphabet $\Sigma$, then $H_5(L,c)$ (for any $c \in C$) is a regular language over alphabet $\Sigma \times (C \cup \{0\})$.

**Proof:** Since $H_4(L,c)$ is a regular language, it can be represented by some regular expression $e$. Then $(\Sigma \times \{0\})^* e$ is a regular expression for $H_5(L,c)$. □

The construction of a machine to accept a suffix-closed highlighted language can also be used to show that a similar prefix-closed highlighted language is regular. Let the $L$-prefix-closed annotated language be $H_6(L,c) = \{\langle w,p\rangle \mid w = xy, p = 0^r c^s, |x| = r, |y| = s,\ y \in L, \text{ and } \forall i, 0 \leq i < r,\ x_{r-i}\ldots x_r y \notin L\ \}$. Then suffix-closure is used to show that prefix-closure also preserves regularity.

**Lemma 4.9.** If $L$ is a regular language over alphabet $\Sigma$, then $H_6(L,c)$ (for any $c \in C$) is a regular language over alphabet $\Sigma \times (C \cup \{0\})$.

**Proof:** If each string in $H_6(L,c)$ is reversed, the reversed language $H_6(L,c)^R = \{\langle w,p\rangle \mid w = yx, p = c^s 0^r, |x| = r, |y| = s,\ y \in L^R, \text{ and } \forall i, 0 < i \leq r,\ yx_1x_2\ldots x_i \notin L^R\ \}$. This language $H_6(L,c)^R$ is identical to $H_4(L^R,c)$. Since $L$ is regular, so is $L^R$, and by Lemma 4.7, so is $H_4(L^R,c)$. □

Once suffix-closed and prefix-closed highlighted languages can be accepted, a regular language can be constructed that is both prefix-closed and suffix-closed. Let the single $L$-maximal substring annotation language be $H_7(L,c) = \{\langle w,p\rangle \mid w = xyz,\ p = 0^r c^s 0^t,\ |x| = r,\ |y| = s,\ |z| = t,\ y \in L, \text{ with } \forall i, 0 \leq i < r,\ x_{r-i}\ldots x_r y \notin L, \text{ and } \forall j, 0 < j \leq t,\ yz_1\ldots z_i \notin L\ \}$. Then $H_7(L,c)$ is the language of prefix-closed and suffix-closed highlighted single instances of strings from $L$.

**Theorem 4.10.** If $L$ is a regular language over alphabet $\Sigma$, then $H_7(L,c)$ (for any $c \in C$) is a regular language over alphabet $\Sigma \times (C \cup \{0\})$ and is finite state recognizable.

**Proof:** Construct a deterministic finite automaton $M^p$ that accepts $L_6(L,c)$. To make the language it accepts be also suffix-closed, follow the construction of $M'$ given in the proof of Lemma 4.7, using $M^p$ instead of $M^0$. □

Note that $H_7(L,c)$ is both prefix-closed and suffix-closed, but is not necessarily closed if both a prefix and suffix are applied together. Those regular languages which require both a prefix and suffix to be applied together in order to extend the string into a valid superstring (eg. $y \in L$, $ay \notin L$, $yb \notin L$, but $ayb \in L$) are not eliminated by these constructions. However, for these languages the existence of the prefix would have to be encoded in all the states of the finite automaton that are traversed in the computation for the string with the prefix. These states would not be traversed for the string without the prefix. Thus this type of regular language would be the union of two or more regular languages, such as $L((ab)^*a + (ba)^*b) = L((ab)^*a) \cup L((ba)*b)$. It cannot be extended by adding either a prefix or a suffix, but could be extended if the correct prefix-suffix pair are both applied. This definition of maximality thus does not eliminate languages that can only be extended in this manner. For cases like this, however, the different languages in the union should be verified and searched for independently.

Any of the previously defined annotated languages can be repeated without altering their regularity, and different languages can be highlighted with different colours.

**Corollary 4.11.** For any finite set of colours $C$ and set of associated regular languages $\mathcal{L} = \{L_c \mid \forall c \in C\}$, the annotated language of maximally coloured strings is $H_8(\mathcal{L}, C) = \{\langle w, p\rangle \mid \forall c \in C$, each c-annotated substring of $w$ is suffix-closed, prefix-closed, and in $L_c$ $\}$. $H_8(\mathcal{L}, C)$ is regular and thus finite state recognizable.

**Proof:** For each $c \in C$, find $H_7(L_c, c)$, which by Theorem 4.10 is regular. Then an annotated string with a single suffix-closed and prefix-closed highlighted substring from any of the languages in $\mathcal{L}$ would be in the regular language

$$\bigcup_{c \in C} H_7(L_c, c)$$

and let

$$e\left(\bigcup_{c \in C} H_7(L_c, c)\right)$$

be a regular expression for this language. To get multiple highlighted strings from potentially different regular languages, use the regular expression

$$e\left(\bigcup_{c \in C} H_7(L_c, c)\right)^+ .$$

This expression shows that the language $H_8(\mathcal{L}, C)$ is regular, and thus is finite state recognizable. $\square$

For example, an annotated language with many highlighted suffix-closed and prefix-closed instances from regular languages $L_1$ and $L_2$ (coloured by $c_1$ and $c_2$, respectively) would be represented by the regular expression $(e(H_7(L_1, c_1) \cup H_7(L_2, c_2))^+$.

However, the potential overlap between a prefix of one string (or language) and the suffix of another can reduce the overall language's ability to reject strings that are not fully annotated, and whose highlighted sections can be extended. This, in turn, reduces the searchability of fully extended regular language strings.

If multiple strings from several different regular languages are highlighted in this manner, they can represent a single layer of coloured substrings that mark words from regular languages. Thus these annotations can be validated by a finite automaton if we confine the highlighted words to those from regular languages. If additional layers of coloured substrings are added to the annotation, they can mark regular structures formed from the coloured words. These structures can also be verified by a finite automaton.

**Definition.** An *$m$-level factored regular expression* is a regular expression $e$ over an alphabet $\Sigma$ together with:

- A sequence of $m$ finite alphabets $\Sigma_1, ..., \Sigma_{m+1}$, where $|\Sigma_1| = 1$ and $\Sigma_{m+1} = \Sigma$.
- For each letter $a \in \Sigma_i$ for $i = 1, ..., m$, a regular expression $e_a$ over the alphabet $\Sigma_{i+1}$.

The regular expression $e$ is obtained by starting with the regular expression $e_a$ for $a \in \Sigma_1$ and progressively substituting to obtain an expression over $\Sigma$.

A multilevel factored regular expression is annotated onto a sequence using colours in the same manner as a single set of regular expressions. Instead of using a single colour as the annotation symbol $c$ in $\langle y_i, c \rangle$, $c$ is the sequence formed by concatenating the sequence of colours $c_a \in \Sigma_h$ for all levels $h \in \{1, \ldots, m\}$ along the path from the root to the substring of the symbol $y_i$.

A multilevel coloured substring annotation of a sequence $w$ can be checked to determine if it parses the sequence according to the structure indicated by a factored regular expression. An annotation symbol thus must encode the sequence symbol's combined colours, with one colour from each level of the layered substrings produced by the factorization.

**Theorem 4.12.** For any fixed constant $m$, the language of valid parse annotations for an $m$-level factored regular expression is regular, and thus finite state recognizable.

**Proof:** The substituted overall regular expression $e$ is regular. The only change from the languages of maximally coloured strings, examined in Corollary 4.11, is in the colours that annotate the symbols, which indicate the parsing of the factored regular expression. $\square$

These theorems do not include any means for distinguishing between adjacent substrings from the same language, since they are given the same colour. To indicate where boundaries between adjacent substrings occur, more annotation symbols need to be introduced. Since the same-length colour sequence criterion prevents adding a symbol to separate the substrings, the terminating symbol needs to be a colour for the last symbol in a substring. Since this colour will still need to indicate what colour its symbol's substring is, the colour set $C = \{1, \ldots, k\}$ can be expanded to $C' = C \cup \{1', \ldots, k'\}$ with $\{1', \ldots, k'\}$ used for substring termination. The automata constructed above can be modified to incorporate these special colours by changing the colour at the accept states of the individual automata $M_i$ from $i$ to $i'$. Note that since the automata that incorporate different colours to highlight different languages are constructed through conversion to and from regular expressions, there is the potential for an exponential increase in the number of states.

## 4.4 Arc Restrictions

Arc annotations can be restricted in a variety of different ways, and sequences with such annotations need to be verified to ensure their conformance to the restrictions. This verification is particularly necessary if the algorithms being applied to compare the annotated sequences are dependent on those restrictions.

As with substring annotations, this verification is done in different ways for different formats. For arcs superimposed onto the sequence, one pass through the sequence can verify unique endpoints, noncrossing arcs, and the arcs' cutwidth or nesting depth. For a list, the arcs must first be verified to be in ascending order of initial endpoints. Then a similar pass down the list can verify the additional restrictions. For either the list or the superimposed arcs, verifying either that the arcs do not cross or the nesting depth requires a stack, as used in this simple algorithm.

**Algorithm 4.13.** Verifying Non-crossing Arcs and finding Nesting Depth.

Let $first$ be the starting endpoint of the arc, and $last$ be the final endpoint of the arc. Let $stack$ be the value at the top of the stack. Start by pushing $n + 1$, where $n$ is the sequence length, onto the stack.

Set $counter$ to 0.
Set $max$ to 0.
for each member of the arc list (in ascending order)
    if $first < stack$
      if $last < stack$
        push $last$
        add 1 to $counter$
        if $counter > max$
          let $max = counter$
      else return $false$
    if $first > stack$
      pop stack
      subtract 1 from $counter$
return $true$
$max$ contains the nesting depth

To determine cutwidth of crossing arcs, we cannot use a stack. The cutwidth is easier to determine from a superimposed set of arcs; to determine cutwidth from an ordered list of arcs, first superimpose their endpoints onto the sequence.

**Algorithm 4.14.** Determining Arc Cutwidth.

Let $n$ be the length of the sequence. Let $first$ be the starting endpoint of the current arc, and $last$ be the final endpoint of the arc.

```
Set counter to 0.
Set max to 0.
for all i from 1 to n
    let a[i] = 0
for each member of the arc list (in order)
    add 1 to a[first]
    subtract 1 from a[last]
for all i from 1 to n
    add a[i] to counter
    if counter > max
        max = counter
max contains the cutwidth
```

## 4.5 Arc Structure

Arcs can also be restricted in the pairs of symbols they can link. For each sequence symbol $x \in \Sigma$, there can be a set of symbols $\Sigma_x \subseteq \Sigma$ that it can be linked to. These sets can also be unidirectional, specifying that an arc that has $x$ as its starting endpoint must have a member of $\Sigma_x$ as its final endpoint. This enables some symbol pairs to be linked only in a specified order.

Verifying that all arcs meet these restrictions depends on the representation of the arcs. If they are represented as a list of pairs ordered by initial endpoint, verification is accomplished linearly by looking at the endpoints of each arc, and comparing the final endpoint to the set that can be linked to the starting endpoint.

If the arcs do not cross or share endpoints, then they can instead be represented by a sequence of spaced arc endpoints with the same length as the basic sequence $S_X$. This annotation, $Q_X \subset \{s, f, \_\}^{|S_X|}$, can be verified, both as a valid representation of nested arcs and as a set of arcs that only links valid pairs, by a deterministic push-down automaton.

**Theorem 4.15.** The language $L = \{(S_X, Q_X) \mid Q_X$ represents nested arcs, and $S_X \in \Sigma^*\}$ is recognizable by a deterministic push-down automaton.

**Proof:** Given alphabet $\Sigma$, a push-down automaton for $L$ is:

For each character pair in sequence,

if the current character pair is:

$(x, \_)$ for any $x \in \Sigma$ : do nothing

$(x, s)$ for any $x \in \Sigma$: push $s$ onto stack

$(x, f)$ for any $x \in \Sigma$: if stack is nonempty, pop top of stack
otherwise reject

If stack is empty after entire string has been processed, accept; otherwise reject. □

**Theorem 4.16.** The language $L = \{(S_X, Q_X) \mid Q_X$ represents nested arcs that links a symbol $x$ to $y \in \Sigma_x$, and $S_X \in \Sigma^*$ $\}$ is recognizable by a deterministic push-down automaton.

**Proof:** Given alphabet $\Sigma$ and set of link restriction alphabets $\Sigma_x$ $\forall x \in \Sigma$ such that $\Sigma_x \subseteq \Sigma$, a push-down automaton for $L$ is:

For each character pair in sequence,

  if the current character pair is:

    $(x, \_)$ for any $x \in \Sigma$ : do nothing

    $(x, s)$ for any $x \in \Sigma$: push $(x, s)$ onto stack

    $(y, f)$ for any $y \in \Sigma$:

      if top of stack is $(x, s)$ and $y \in \Sigma_x$, pop top of stack

      otherwise reject

If stack is empty after entire string has been processed, accept; otherwise reject. $\Box$

# Chapter 5

# Annotation Creation

## 5.1 Sources of Creation Problems

Annotations that can be verified by computer could also be created by computer. Given a set of restrictions, an annotation potentially can be computed that meets these requirements. However, if the restrictions are not very specific then the annotations thus generated may not be very informative. Creating annotations also includes assembling the information from raw data and massaging existing information into the correct form. User-defined colourings can be automated to find substrings that meet certain criteria and colour them appropriately. This automation involves string search algorithms to look for useful patterns, such as substring matching, subsequence matching, and approximate string matching [CL94].

Existing secondary information needs to be transformed into the annotations so that it can be used for sequence analysis and manipulation. Although there is a lot of information about sequences available, it is usually stored in databases as separate attributes for the sequence entry, not in a manner that would enable these attributes to be incorporated into the sequence algorithms. The algorithms, then, do not usually incorporate this information. For example, a computed sequence alignment could be checked to see if the relationships in an attached list are preserved by the alignment, but there is no means of computing such an alignment to ensure that the relationships are preserved. For annotations that are stored in a format other than the one used for sequence analysis, creation also includes converting from the storage to the analysis format.

## 5.2 Colour Interpolation

Creating annotations can also include interpolation, where a subset of the annotation is already known. This information can either come from incomplete previously known information or it can be transferred from a related annotated sequence. While this principle could be applied to any form of annotation, we specifically look at coloured sequences. Given a partially coloured sequence and a set of restrictions, a complete annotation that is consistent with both the partial colouring and the restrictions can be computed.

One annotation interpolation problem, for coloured sequences where each colour is limited in the length of each segment and in its possible symbols, is the following Limited Colour Length Interpolation Problem:

*Input:* sequence $S_X = S_X[1]S_X[2]\ldots S_X[n]$, where $\forall i \in \{1,\ldots,n\}\ S_X[i] \in \Sigma$
finite alphabet $\Sigma$
label set $C$
labeled sample $L_s \subset C \times \{1,\ldots,n\}$
length maxima $MAX = (MAX_j : j \in C,\ MAX_j \in \{1,\ldots,n\})$
sets of allowed symbols for each colour, $\mathcal{S} = (\Sigma_j \subseteq \Sigma : j \in C)$

*Output:* sequence labeling $L_X = L_X[1]L_X[2]\ldots L_X[n]$ where $\forall i \in \{1,\ldots,n\}$, $L_X[i] \in C$, $S_X[i] \in \Sigma_{L_X[i]}$, and if $\exists c \in C$ such that $\langle c,i\rangle \in L_s$ then $L_X[i] = c$. Also, $\forall i \in \{1,\ldots,n\}$, $\min_{L_X[j]\neq L_X[i]}(j > i) - \max_{L_X[j]\neq L_X[i]}(j < i) - 1 \leq MAX_{L_X[i]}$ . If no valid $L_X$ exists that is consistent with $L_s$, then *false* is output instead.

An example instance of this problem is:

*Input:* $S_X = attgcgat$

$\Sigma = \{a,c,g,t\}$

$C = \{b,r,y\}$

$L_s = \{\langle b,4\rangle, \langle b,6\rangle, \langle r,7\rangle\}$

$MAX = (MAX_b, MAX_r, MAX_y) = (3,2,2)$

$\mathcal{S} = (\Sigma_b, \Sigma_r, \Sigma_y) = (\{a,c,g\}, \{a,g,t\}, \{c,t\})$

An output for this instance is $L_X = ryybbbrr$. This output is not unique, although some instances with strict restrictions and dense samples may have a unique output. Some instances may not have any correct colouring.

A correctly interpolated annotation can be computed in a variety of ways. Here are two algorithms which extend a partial annotation to a full annotation that complies with the colour and length restrictions.

**Algorithm 5.1.** Greedy Interpolation.

Start by computing a two-dimensional grid $M$
containing the colour possibilities at each position.
Let $C$ be ordered, so $C = \{c_1, c_2, \ldots, c_{|C|}\}$.
for $i$ from 1 to $n$
    for $j$ from $c_1$ to $c_{|C|}$

```
        if S_X[i] ∈ Σ_j
          let M[i, j] = 0
        else let M[i, j] = −1
    let L_X[i] = 0
for each member ⟨x, y⟩ of L_s
    for j from c_1 to c_|C|
        if j = x
          if M[y, j] = −1
            return false
          else let M[y, j] = 1
        else let M[y, j] = −1
    let L_X[y] = x
for j from c_1 to c_|C|
    let M[0, j] = −1
```

Then extend each coloured point from the sample as far as possible.

Let the sample $L_s$ be ordered, so $L_s = \{\langle x_1, y_1\rangle, \langle x_2, y_2\rangle, \ldots, \langle x_{|L_s|}, y_{|L_s|}\rangle\}$,

where $\forall h \in \{1, \ldots, |L_s| - 1\}$, $y_h < y_{h+1}$.

If for any such $h$, $y_h = y_{h+1}$, the algorithm would already return *false*.

```
let h = 1
while h ≤ |L_s|
    let i = y_h
    let c = x_h
    let max = MAX_c − 1
```

Extend the colour to the left by:

while $max > 0$ and $M[i-1, c] \neq -1$

subtract 1 from $i$

let $L_X[i] = c$

for $j$ from $c_1$ to $c_{|C|}$

if $j = c$

let $M[i, j] = 1$

else let $M[i, j] = -1$

subtract 1 from $max$

Extend the colour to the right by:

let $i = y_h$

while $max > 0$ and $M[i+1, c] \neq -1$

add 1 to $i$

subtract 1 from $max$

if $M[i+1, c] = 1$ (next position already has colour $c$)

let $oldi = i$

while $M[i+1, c] = 1$

add 1 to $i$

subtract 1 from $max$

if $max \leq 0$

let $M[oldi] = -1$

else (can add next run of samples to this string)

add $i - oldi$ to $h$

let $L_X[oldi] = c$

for $j$ from $c_1$ to $c_{|C|}$

if $j = c$

let $M[oldi, j] = 1$

else let $M[oldi, j] = -1$

else for $j$ from $c_1$ to $c_{|C|}$

if $j = c$

let $M[i, j] = 1$

let $L_X[i] = c$

else let $M[i, j] = -1$

if $max = 0$

let $M[i + 1, c] = -1$

add 1 to $h$

After interpolating the sample points, fill in anything uncoloured.

let $c = 1$

for $i$ from 1 to $n$

if $L_X[i] = 0$

if $M[i, c] = 0$

let $M[i, c] = 1$

let $L_X[i] = c$

else (can't use the colour of the last position)

let $c = 1$

while $c \leq c_{|C|}$

if $M[i, c] = 0$

let $M[i, c] = 1$

let $L_X[i] = c$

add 1 to $c$

The resulting complete sequence of colours that annotates the input sequence while preserving the sample and the restrictions is $L_X$.

Instead of treating each coloured position in the sample as separate, another algorithm could try to link adjacent samples of the same colour. This algorithm is similar to the greedy interpolation, but chooses its initial direction of extension based on the neighbouring sample colours.

**Algorithm 5.2.** Linking Interpolation.

Start by computing the grid $M$ as before,
and let the sample $L_s$ be ordered as for greedy interpolation.

let $h = 1$

while $h \leq |L_s|$

let $i = y_h$

let $c = x_h$

let $max = MAX_c - 1$

if $x_h = x_{h+1}$

first extend the colour to the right

then extend the colour to the left

else (next sample point has a different colour)

first extend the colour to the left

then extend the colour to the right

add 1 to $h$

After extending all coloured positions from the sample,
fill in any uncoloured positions as for the greedy algorithm.

Both these versions of interpolation run in time $\in O(n \cdot |C|)$.

## 5.3 Searching for Regular Languages

For symbol or substring colouring that highlight subsequences or substrings that are words from a regular language or languages, an annotation can be extracted from the sequence using a pair of finite state transducers. For a given language $L$ of annotated strings, the corresponding search problem is defined by:

$L$-Search
*Input:* A string $X \in \Sigma^*$.
*Output:* A string $Y \in \Gamma^*$ with $|Y| = |X|$ and $(X, Y) \in L$ (if any exists, otherwise a failure message).

**finite state searchability**
A language $L \subseteq (\Sigma \times \Gamma)^*$ of annotated strings is *finite state searchable* if there is a pair of finite state transducers that solve $L$-Search:

- $M_1 = (Q_1, s_1, \delta_1, \Sigma, \Delta)$, where $\delta_1 : Q_1 \times \Sigma \to Q_1 \times \Delta$
- $M_2 = (Q_2, s_2, \delta_2, (\Sigma \times \Delta), \Gamma)$, where $\delta_2 : Q_2 \times \Sigma \times \Delta \to Q_2 \times \Gamma$

These transducers solve $L$-Search in the following way:

1. $M_1$ processes the string $X$ left-to-right and computes an intermediate annotation $Z$ of $X$, where $(X, Z) \in (\Sigma \times \Delta)^*$, where $\Delta$ is a finite intermediate alphabet, and

2. $M_2$ processes $(X, Z)$ right-to-left and computes either $Y$ such that $(X, Y) \in L$ or a special annotation $Z_0$ indicating that no such $Y$ exists.

**Theorem 5.3.**[BEF96] A language $L$ of annotated strings is finite state searchable if $L$ is finite state recognizable.

The converse to this theorem can be shown to be false by various counterexamples, such as (for $\Sigma = \Gamma = \{0, 1\}$):

$$L = \{(X, X) : X \in \Sigma^*\} \cup \{(X, 1^{|X|}) : \text{ Turing machine } M_X \text{ halts }\}$$

which is undecidable but trivially finite state searchable.

This theorem can be used to show finite state searchability for the $L$-Search problems corresponding to the finite state recognizable languages shown to be verifiable in Chapter 4. For finite state searching, the languages that enforce maximal annotation need to be used to avoid finding trivial and blank annotations. Note, however, that this maximality only requires that the strings found cannot be extended by either a prefix or a suffix; those strings that can only be extended by the addition of both a prefix and a suffix are still acceptable. These last type of extensible string will only exist if the language is the union of two or more regular languages, and thus strings from these languages can and should be searched for independently. Each of the following corollaries follows directly from the above theorem and from the finite state recognizability of the languages in question.

**Corollary 5.4.** The $L$-Subsequence Annotation Search problem ($H_1(L, c)$ where $L$ is regular), with
*Input:* A sequence $w \in \Sigma^*$.
*Output:* A sequence $p$ of length $|w|$, where $p \in \{0, c\}^*$, such that the coloured subsequence $w'$ of $w$ (consisting of those symbols $w_i$ for which $p_i = c$) belongs to $L$ is finite state searchable.

**Proof:** follows directly from Theorem 5.3 and Theorem 4.4. $\square$

**Corollary 5.5.** The Single $L$-Maximal Substring Annotation Search problem ($H_7(L, c)$ where $L$ is regular), with

*Input:* A sequence $w \in \Sigma^*$.

*Output:* A annotation sequence $p$ of length $|w|$, $p \in \{0, c\}^*$, such that $p = 0^r c^s 0^t$, $|x| = r$, $|y| = s$, $|z| = t$, $y \in L$, with $\forall i, 0 \leq i < r$, $x_{r-i} \ldots x_r y \notin L$, and $\forall j, 0 < j \leq t$, $y z_1 \ldots z_i \notin L$

is finite state searchable.

**Proof:** follows directly from Theorem 5.3 and Theorem 4.10. $\square$

**Corollary 5.6.** The Multiple $\mathcal{L}$-Maximal Substring Annotation Search problem ($H_8(\mathcal{L}, C)$ where each $L \in \mathcal{L}$ is regular), with

*Input:* A sequence $w \in \Sigma^*$.

*Output:* A annotation sequence $p$ of length $|w|$, $p \in (C \cup \{0\})^*$, such that each c-annotated substring of $w$ is suffix-closed, prefix-closed, and in $L_c$

is finite state searchable.

**Proof:** follows directly from Theorem 5.3 and Theorem 4.11. $\square$

**Corollary 5.7.** The problem of computing a parse annotation for an $m$-level factored regular expression is finite state searchable.

**Proof:** follows directly from Theorem 5.3 and Theorem 4.12. $\square$

## 5.4 Format Translation

### 5.4.1 The Need for Format Translation

The information stored in the annotation can only be integrated into the sequence comparison if it is easily accessible along with the sequence. Storage formats, on the other hand, are more likely to be chosen based on how they are generated and to enable people to understand the information when the annotation is visually inspected. The storage formats described in chapter 3, which store layers of coloured substrings as a collection of ordered lists instead of as the trees that would be more useful to the comparison algorithm, are examples of this preference for a more readable format. If the format used for storage is not suitable for inclusion in comparison algorithms, it must be converted to a more suitable format. This conversion can be considered to be necessary preprocessing of the data, changing it from an external storage format into a more internal format that is more efficient for the comparison programs to use. Lists of substrings need to be converted to sequences of substrings, and layers of these substrings must be similarly transformed into an ordered sequence of ordered forests. Arc annotations stored as lists of arcs need to be superimposed onto the sequence.

### 5.4.2 Coloured Substring Formats

For coloured substrings and layers of coloured substrings, the collection of ordered lists must be converted into the corresponding sequence of ordered forests. This translation can be accomplished by a breadth-first search style algorithm that constructs the forests from a level traversal.

**Algorithm 5.8.** Breadth-First Forest Construction.

Start with one unmarked node as the root of a tree that links the forests together in order. Make the members of the highest level list the children of this root. Process each pair of adjacent lists, from highest level to lowest. Let *list1* refer to the higher level list, and *list2* refer to the lower level list. For each member of each list, $first$ is the start of the substring and $last$ is the last substring.

start at the beginning for *list1* and *list2*.
while not after the end of *list2*
    if $first$ of *list2* $>$ $last$ of *list1*
        advance to next member of *list1*
    else if $last$ of *list2* $>$ $last$ of *list1*
        return FALSE
    else (*list2* substring is contained in *list1* substring)
        make current member of *list2* a child of current member of *list1*
        advance to next member of *list2*

For a layered substring annotation with $k$ levels, this algorithm runs in time $\in O(kn)$. If each higher-level substring includes at least 2 substrings from the level immediately below it, the algorithm runs in time $\in O(n)$.

Coloured substrings can also be represented by a same-length sequence of colours. This annotation can be converted to a list of coloured substrings by a single pass along the sequence. If an extra set of related colours is used as coloured substring terminators, the conversion must look for them to end each substring. On the other hand, if these special colours are not used as terminators, the conversion process instead separates substrings by the change in colour. A substring ends at a sequence position if the next position has a different colour. The conversion process needs to know if terminating colours are being used in the representation, since breaking into substrings by colour change would otherwise turn each instance of a terminating colour into its own substring of length 1.

### 5.4.3 Arc Formats

For arcs, an ordered list is easily superimposed onto the sequence by indexing the sequence positions with arc endpoints, and adding a link between the pair of endpoints at their position in the sequence.

A set of nested arcs represented by a same-length spaced endpoint sequence, $Q_X \in \{s, f, blank\}^*$ where $\#_s(Q_X) = \#_f(Q_X)$, needs to be converted to an ordered list of arcs. This ordered list could then be used for storage or to subsequently superimpose the arcs onto the basic sequence. To extract the correct pair of endpoints for each arc from the same-length annotation sequence, each starting endpoint is pushed onto a stack and then paired with the corresponding final endpoint.

**Algorithm 5.9.** Spaced Endpoint Sequence to Arc Conversion.

Let $n$ be the length of the sequences.

Let $A$ be the indexable list of arcs.

Set $counter$ to 1.

for $i$ from 1 to $n$

  if $Q_X[i] = s$

    push $(i, counter)$ onto stack

    add 1 to $counter$

  if $Q_X[i] = f$

    pop top value $(x, y)$ from stack

    let arc $A[y] = (x, i)$

For any sequence with a spaced endpoint sequence annotation, this algorithm runs in time linear in the length of the sequences.

# Chapter 6

# Comparing Coloured Sequences

## 6.1 Coloured Symbols

Sequences annotated with coloured symbols or substrings can be compared in ways that incorporate the annotations. Coloured symbol annotations, where the basic sequence has a corresponding same-length sequence of colours, can be incorporated into sequence comparisons by extending the alphabet. Instead of a single symbol representing just the basic sequence symbol at that position, it instead would represent both the basic symbol (from $\Sigma$) and its colour (from $C$), and come from the set $\Sigma \times C$. This new extended alphabet can be reduced in size if there are any colour restrictions for some symbols, as described in section 3.2.

For the longest common subsequence of two sequences with coloured symbols, the coloured subsequence must match both symbol and colour. For a sequence comparison score with weighted match and mismatch values, using symbol and colour together will produce a large extended table of values of size $(|\Sigma| \cdot |C|)^2$. This table can be used with the classic longest common subsequence algorithm [NW70, SW81] to find an alignment weighted for both colour and symbol match.

The colours and base symbols can be looked at separately to observe how they interact. Colours can be used either as an additional symbol at each position that should be matched, or weighted relative to the sequence match. However, the colour can be instead a weight, one that increases or decreases the value of matching the symbol at that position. For this and the other algorithms in this chapter, the base cases, $T[i,j] = 0$ if either $i = 0$ or $j = 0$, are used as for the original LCS algorithm.

**Algorithm 6.1.** Finding the comparison score of two sequences with coloured symbols, using relative colour and symbol weighting.

for $i$ from 1 to $n$

  for $j$ from 1 to $m$

$$\begin{aligned} T[i,j] = \max(&T[i-1,j], T[i,j-1], \\ &T[i-1,j-1] + w(S_1[i], S_2[j]) \cdot cw_m(L_1[i], L_2[j]) + cw_a(L_1[i], L_2[j])) \end{aligned}$$

with original weight table $w$ for symbol match, and colour weight tables $cw_m$ and $cw_a$ for colour match. The colour match can scale the symbol weight value from $w$ using the multiplicative weight $cw_m$ with $cw_a(x,y) = 0$ for all colours $x$ and $y$, add to it using $cw_a$ with $cw_m(x,y) = 1$ for all colours $x$ and $y$, or combine them by using both $cw_m$ and $cw_a$. The comparison score is found in $T[n,m]$.

Colours, or other labels, can also be used to indicate the relative importance of the symbols that they are applied to. Symbols of high importance should contribute more to the overall similarity score. It is not necessary, however, to match these importance colours, as they do not indicate an attribute of the symbol that needs to be matched. If the colours are used in this manner to indicate a weighting for the symbols, the colours are not matched but instead refer to separate tables of scaling factors. The symbol colours indicate how important the symbols are in the comparison.

**Algorithm 6.2.** Finding the comparison score of two sequences with coloured symbols, using colours to indicate importance.

for $i$ from 1 to $n$

  for $j$ from 1 to $m$

$$T[i,j] = \max(T[i-1,j], T[i,j-1],$$
$$T[i-1,j-1] + w(S_1[i], S_2[j]) \cdot cw_1(L_1[i]) \cdot cw_2(L_2[j]))$$

with original weight table $w$ for symbol match, and colour weight factors $cw_1$ and $cw_2$. Either list of factors may be all equal to 1, indicating that symbol importance is not used for that sequence.

To use these factors, at least one of the sequences must be annotated with colours indicating symbol importance. For comparison scores that are calculated as part of a database search, the database sequences should be annotated. This scheme has been implemented for a database of protein sequences, where each sequence is coloured to indicate the location of the family motif [ECU96]. Each sequence in the database which contains one or more motifs is annotated with the position of each motif and a sequence of colours for the motif. Within the motif, assigning a colour to a position indicates the number of possible amino acids at that position in the regular expression for the motif. The values in this sequence are later used to index a table of annotation weights. Using a table of small importance weights encourages the algorithm to align query sequences with the motif, and increases scores for database sequences whose motif is matched well by the query sequence. Our implementation, which uses the comparison calculation given in Algorithm 6.2, uses a commonly used table of weights for symbol matches, and also incorporates commonly used penalties for gaps in the alignment. The table of annotation weights was determined experimentally to produce high comparison scores for proteins from the same family as the query sequence. The code was modified from the SEQSEE search code [WBWRS94], and the search results were compared to the results of the unmodified code (without importance weights). After experimentation to determine a good set of importance weights, the modified search produced better results than the unmodified search. The top scoring sequences were largely those that were more closely related, and some non-related sequences that scored highly in the original search were no longer returned [ECU96].

This approach differs from that of Taylor [Tay94] both in that the common substring or subsequence is recognized as a genuine motif, and that a match of the entire substring is not necessary. Protein motifs frequently have gaps, wildcard symbols, and less important bases; motif colouring enables an alignment to favour a match that primarily preserves the significant base symbols. Some editing in the motif is also permitted, as the two sequences may have different instances of the same motif, with some symbols substituted. The scoring method of Algorithm 6.2 allows these gaps or substitutions to occur in the motif, while Taylor's algorithm would reset the running product to zero.

## 6.2 Coloured Substrings

When sequences annotated with coloured substrings are compared, the comparison needs to preserve substring membership. Specifically, if $i_1$ and $i_2$ are part of the selected subsequence and are matched to $j_1$ and $j_2$, $i_1$ is in the same substring as $i_2$ if and only if $j_1$ is in the same substring as $j_2$. Preserving substrings is useful if the substring colouring indicates pieces with a specific function or structure.

When the colouring is based on the entire contiguous piece rather than individual symbols, keeping the symbols in each substring together helps preserve the meaning of the annotation. A close match would match a substring of one sequence with a corresponding substring of the other sequence, rather than matching bits of it to parts of many different substrings.

The longest common subsequence problem preserving coloured substrings takes as input two sequences of coloured substrings, $S_1^1$ and $S_2^1$, and a target length $l$. Let $S_1$ and $S_2$ be the original unannotated base sequences as defined in Chapter 3. The solution returns "yes" if and only if there is a mapping $MS \subset \{1, \ldots, |S_1|\} \times \{1, \ldots, |S_2|\}$, $|MS| = l$, such that

- the mapping is one-to-one and preserves the order of the subsequence :

$$\begin{aligned} &\forall (i_1, j_1) \in MS \text{ and } (i_2, j_2) \in MS \\ &\qquad i_1 = i_2 \text{ if and only if } j_1 = j_2 \\ &\qquad i_1 < i_2 \text{ if and only if } j_1 < j_2 \end{aligned}$$

- the substrings of the sequences and their colours are preserved :

  $\forall (i_1, j_1) \in MS$ and $(i_2, j_2) \in MS$, $i_1$ and $i_2$ are both part of some substring $S_1[i]$ with $L_1[i] = c$ if and only if $j_1$ and $j_2$ are both part of some substring $S_2[j]$ with $L_2[j] = c$ .

To solve this problem, the substring annotation is used to guide the sequence matching. Since the symbols in each substring from one sequence can only be matched to the symbols in some substring from the other sequence, the sequence of coloured substrings can be aligned as if the substrings were individual symbols. This produces a two-level version of the classic algorithm that finds the longest common subsequence.

**Algorithm 6.3.** Finding the length of the longest common subsequence of
$S_1^1 = \langle S_1[1], L_1[1]\rangle\langle S_1[2], L_1[2]\rangle \ldots \langle S_1[h_1], L_1[h_1]\rangle$ and
$S_2^1 = \langle S_2[1], L_2[1]\rangle\langle S_2[2], L_2[2]\rangle \ldots \langle S_2[h_2], L_2[h_2]\rangle$
that preserves coloured substrings.

for $i$ from 1 to $h_1$
  for $j$ from 1 to $h_2$
    $T[i,j] = \max(T[i-1,j], T[i,j-1], T[i-1,j-1] + cw(S_1^1[i], S_2^1[j]))$
    with weight table

$$cw(\langle S_1[i], L1[i]\rangle, \langle S_2[j], L_2[j]\rangle) = \begin{cases} 0 & \text{if } L_1[i] \neq L_2[j] \\ lcs(S_1[i], S_2[j]) & \text{if } L_1[i] = L_2[j] \end{cases}$$

where $lcs(S_X, S_Y) = T[|S_X|, |S_Y|]$, as computed by

for $i$ from 1 to $|S_X|$
  for $j$ from 1 to $|S_Y|$
    $T[i,j] = \max(T[i-1,j], T[i,j-1], T[i-1,j-1] + w(S_1[i], S_2[j]))$
    where $w$ is determined by

$$w(x,y) = \begin{cases} 1 & \text{if } x = y \\ 0 & \text{otherwise} \end{cases}$$

## 6.3 Layers of Coloured Substrings

Just as the substring annotation can be extended to any number of levels, the above algorithm can also be extended to become a recursive algorithm that finds the substring-preserving longest common subsequence for any number of levels.

**Algorithm 6.4.** Finding $Slcs(S_1^k, S_2^k)$, the length of the longest common subsequence of $S_1^k$ and $S_2^k$ that preserves coloured substrings.

If both input sequences $S_1^k$ and $S_2^k$ are basic sequences ($k = 0$), then $S_1^k = S_1$ and $S_2^k = S_2$. We can compute $Slcs(S_1, S_2)$ by:

for $i$ from 1 to $|S_1|$

    for $j$ from 1 to $|S_2|$

        $T[i,j] = \max(T[i-1,j], T[i,j-1], T[i-1,j-1] + w(S_1[i], S_2[j]))$

        where $w$ is determined by

$$w(x,y) = \begin{cases} 1 & \text{if } x = y \\ 0 & \text{otherwise} \end{cases}$$

Otherwise ($k \geq 1$), let $h_1 = |S_1|$ and $h_2 = |S_2|$. The input sequences are sequences of coloured substrings, $S_X = S_1[1 \ldots h_1]$ and $S_Y = S_2[1 \ldots h_2]$. The length of their longest common subsequence that preserves coloured substrings is computed recursively by:

for $i$ from 1 to $h_1$

    for $j$ from 1 to $h_2$

        $T[i,j] = \max(T[i-1,j], T[i,j-1], T[i-1,j-1] + cw(S_1^1[i], S_2^1[j]))$

        with weights determined by

$$cw(\langle S_1[i], L_1[i]\rangle, \langle S_2[j], L_2[j]\rangle) = \begin{cases} 0 & \text{if } L_1[i] \neq L_2[j] \\ Slcs(S_1[i], S_2[j]) & \text{if } L_1[i] = L_2[j] \end{cases}$$

These algorithms find the length of the longest common subsequence that preserves the coloured substrings, but they can be changed to use weights for sequence symbol matches, and weight the colour matching relative to subsequence match.

To give bonus scores $b(c)$ for coloured substrings matched, depending on their colour $c$, change the weight determination to :

$$cw(\langle S_1[i], L_1[i]\rangle, \langle S_2[j], L_2[j]\rangle) = \begin{cases} 0 & \text{if } L_1[i] \neq L_2[j] \\ b(L_1[i]) + Slcs(S_1[i], S_2[j]) & \text{if } L_1[i] = L_2[j] \end{cases}$$

To use weights for both symbol and colour matches and mismatches, change the calculation of $w$ and $b$ so that $w(s_1, s_2)$ is determined by a table of weights for each pair of potentially corresponding symbols, and $b(c_1, c_2)$ is a table of weights for each pair of potentially corresponding colours. The weight calculation $cw$ is changed to:

$$cw(\langle S_1[i], L_1[i]\rangle, \langle S_2[j], L_2[j]\rangle) = b(L_1[i], L_2[j]) + Slcs(S_1[i], S_2[j])$$

While these changes add the colour weights to the total symbol weights for the substring, the colour match weights could instead be small factors that multiply the symbol weights. As with all weighting schemes, usable numbers need to be derived through experimentation, and are dependent on the particular application of the sequences and annotation.

To analyze the running time of the recursive algorithm, assume that the substrings have at most $k$ levels. For each level $l$ of coloured substrings, the algorithm takes $O(n_l, m_l)$ time, where $n_l$ is the number of substrings at level $l$ of the layered substring forest $S_1^k$, and $m_l$ is the number of substrings at level $l$ of the layered substring forest $S_2^k$. If the substrings have at most $k$ levels, the recursive algorithm runs in time $\in O(knm)$, where $n$ is the length of the unannotated base sequence $S_1$ and $m$ is the length of the unannotated base sequence $S_2$. If $k$ is unbounded, it can be at most $O(\max(n, m))$, so the algorithm runs in time $O(n^2 m)$, assuming without loss of generality that $n \geq m$.

This recursive technique can also be applied to mixed annotations that incorporate arcs as well as substrings. This application is discussed in chapter 8.

# Chapter 7

# Comparing Arc-Annotated Sequences

## 7.1 Introduction

When arcs are used to link sequence symbols to represent nonsequential information, comparing the resulting annotated sequences is complex. To incorporate both the arcs and sequences into an overall measure of similarity, the arcs can be weighted relative to sequence symbol matching, used to guide the sequence alignment, or simply preserved by the sequence alignment. Due to its simplicity, this last method of information incorporation is used, avoiding the complications of application-specific weighting. Extensions that incorporate weights are discussed in chapter 8.

The resulting annotated sequence similarity is measured by the longest common subsequence of the two sequences that preserves their arcs. To preserve arcs, a subsequence that selects both endpoints of an arc from one sequence must map those endpoints to the endpoints of some arc from the other sequence. While arc preservation does not directly involve weighting the arcs relative to the sequence symbols, algorithms for arc preservation can be modified to use weights, much as the original pairwise longest common subsequence algorithm is modified to use weights for different symbol matches and mismatches.

Using arcs to represent nonsequential information allows for varying restrictions on the complexity of the arc structure. Arcs can be unlimited, have unique endpoints, be allowed to nest only instead of cross, limited to a single chain of non-nesting arcs, or arcs can be not present at all. The two sequences can have different restrictions. The resulting subsequence will have its arc annotation restricted by the most limited of the two input restrictions.

These different combinations of restrictions alter the computational complexity of the longest common subsequence problem, and need to be examined separately. This examination produces algorithms for more restrictive arc structures, and hardness results for those less restrictive. This computational complexity can be examined both in the classical complexity framework and in the parameterized complexity framework. For parameterized complexity, parameters natural to the types of restriction, such as cutwidth, nesting depth, and subsequence length, are considered.

## 7.2 The General Arc-Preserving LCS Problem

A problem $\Pi$ of finding a common subsequence of length $k$ which preserves induced arcs takes as input the target length $k$, and the pair of annotated sequences $(S_1, P_1)$ and $(S_2, P_2)$. These annotated pairs consist of the sequences $S_1$ and $S_2$ over some fixed alphabet $\Sigma$, with arc annotations $P_1 \subset \{1, \ldots, |S_1|\}^2$ and $P_2 \subset \{1, \ldots, |S_2|\}^2$. The length of $S_1$ and $S_2$ are $n$ and $m$ respectively. A solution returns "yes" if and only if there was some mapping $MS \subset \{1, \ldots, |S_1|\} \times \{1, \ldots, |S_2|\}$ between the positions of $S_1$ and $S_2$ such that $|MS| = k$ and

1. the mapping is one-to-one and preserves the order of the subsequence :

   $$\begin{aligned} &\forall (i_1, j_1) \in MS \text{ and } (i_2, j_2) \in MS \\ &\qquad\qquad i_1 = i_2 \text{ if and only if } j_1 = j_2 \\ &\qquad\qquad i_1 < i_2 \text{ if and only if } j_1 < j_2 \end{aligned}$$

2. the arcs induced by the mapping are preserved :

   $\forall (i_1, j_1) \in MS$ and $(i_2, j_2) \in MS$, $(i_1, i_2) \in P_1$ if and only if $(j_1, j_2) \in P_2$

3. the mapping produces a common subsequence :

   $\forall (i, j) \in MS$, $S_1[i] = S_2[j]$.

## 7.3 Restriction Variations

Allowing the arc annotations to be unrestricted makes the general problem $\Pi$ intractable, which is shown in Theorem 7.4 as proven in section 7.5. Because of this hardness, the arc structure must be restricted. These restrictions go naturally with the types of information that the arcs potentially represent. The arcs $P_X$ for a sequence $S_X$ can be required to:

1. not share endpoints

   $$\begin{aligned} &\forall (i_1, i_2) \in P_X \text{ and } (i_3, i_4) \in P_X \\ &\qquad\qquad i_1 = i_3 \text{ if and only if } i_2 = i_4 \\ &\qquad\qquad i_1 \neq i_4 \text{ and } i_2 \neq i_3 \end{aligned}$$

2. not cross

$$\forall (i_1, i_2) \in P_X \text{ and } (i_3, i_4) \in P_X$$
$$i_1 \in [i_3, i_4] \text{ if and only if } i_2 \in [i_3, i_4]$$

3. not nest

$$\forall (i_1, i_2) \in P_X \text{ and } (i_3, i_4) \in P_X$$
$$\text{if } i_1 \neq i_3,\ i_2 < i_3 \text{ or } i_1 > i_4$$

4. not be present, so $P_X = \emptyset$.

These restrictions are used progressively and inclusively to produce five different levels of allowed arc structure for the problem $\Pi$.

- *unlim* - the general problem with no restrictions
- *cross* - restriction 1
- *nest* - restrictions 1 and 2
- *chain* - restrictions 1, 2, and 3
- *plain* - restriction 4 (no arcs)

These different restrictions on the structure of the arc annotations can be applied to either or both of the two sequences being compared. The problem $\Pi$ is varied by these different levels of restrictions as $\Pi(x, y)$ which is problem $\Pi$ with $S_1$ having restriction level $x$ and $S_2$ having restriction level $y$. If the two sequences

$$\begin{array}{lllll}
\Pi(unlim, unlim) & & & & \\
\cup & & & & \\
\Pi(unlim, cross) \supset & \Pi(cross, cross) & & & \\
\cup & \cup & & & \\
\Pi(unlim, nest) \supset & \Pi(cross, nest) \supset & \Pi(nest, nest) & & \\
\cup & \cup & \cup & & \\
\Pi(unlim, chain) \supset & \Pi(cross, chain) \supset & \Pi(nest, chain) \supset & \Pi(chain, chain) & \\
\cup & \cup & \cup & \cup & \\
\Pi(unlim, plain) \supset & \Pi(cross, plain) \supset & \Pi(nest, plain) \supset & \Pi(chain, plain) \supset & \Pi(plain, plain)
\end{array}$$

Table 7.1: Problem inclusions for different levels of restriction on the problem $\Pi(x, y)$

have different restrictions, $S_1$ is considered to be the sequence with the more lenient level of restrictions. This restriction hierarchy give the problem inclusions illustrated in Table 7.1. $\Pi(unlim, unlim)$ is the general arc-preserving longest common subsequence problem, and $\Pi(plain, plain)$ is the unannotated longest common subsequence problem.

## 7.4 Problem Reductions

To show the hardness of some of the variations with fewer restrictions on the arc structure, the following polynomial time parametric and nonparametric reductions are used.

**Lemma 7.1.** Independent Set is polynomially reducible and strongly uniformly parametrically reducible to $\Pi(unlim, plain)$.

**Reduction.** From $k$-Independent Set of a graph $G = (V, E)$, where $V = \{1, 2, \ldots, n\}$.

Let $S_1 = a^n$ with $P_1 = E$. Let $S_2 = a^k$ with $P_2 = \emptyset$.

The graph $G$ has an independent set of size $k$ if and only if there is a mapping set $MS$ of size $k$ for $(S_1, P_1)$ and $(S_2, P_2)$.

**Proof:**

$\Rightarrow$: Assume there is some independent set $V' \subseteq V$ of $G$, $|V'| = k$. By definition of independent set, $\forall u, v \in V'$, $(u, v) \notin E$. For a given subset $V'$, let $p : V' \to \{0, 1, \ldots, k-1\}$ be defined by $p(u) = |\{v \in V' \ : \ v < u\}|$. Let $MS = \{(u, p(u) + 1) \mid u \in V'\}$. Clearly, $\forall (i_1, j_1) \in MS$ and $(i_2, j_2) \in MS$, $i_1 = i_2$ if and only if $j_1 = j_2$, and $i_1 < i_2$ if and only if $j_1 = p(i_1) \ < \ p(i_2) = j_2$. Since $V'$ is an independent set, if $(u, v) \in E = P_1$ then either $(u, p(u)) \notin MS$ or $(v, p(v)) \notin MS$. This preserves arcs since $P_2$ is empty. Trivially, $S_1[u] = S_2[p(u)]\ \forall u \in V$.

$\Leftarrow$: Assume there is a valid mapping $MS$, with $|MS| = k$. Let $I = \{i \mid (i, j) \in MS\}$. $|I| = |MS| = k$. Let $i_1$ and $i_2$ be any two distinct members of $I$. Then let $(i_1, j_1) \in MS$ and $(i_2, j_2) \in MS$, with $j_1 \neq j_2$. Since $P_2$ is empty, $(j_1, j_2) \notin P_2$, so $(i_1, i_2) \notin P_1$. Since $P_1 = E$, the set $I$ of vertices is a size $k$ independent set of $G$. □

**Lemma 7.2.** Clique is polynomially reducible and strongly uniformly parametrically reducible to $\Pi(cross, cross)$.

**Reduction.** From $k$-Clique of a graph $G = (V, E)$, where $V = \{1, 2, \ldots, n\}$.

Construct $S_1[1..(n^2 + 2n)]$ and $P_1$ as follows:

$$S_1 = (ba^n b)^n$$

$$P_1 = \{((u-1)(n+2)+1, u(n+2)) | u \in V\}$$

$$\cup \{((u-1)(n+2)+v+1, (v-1)(n+2)+u+1) | (u, v) \in E\}$$

Construct $S_2[1..(k^2 + k)]$ and $P_2$ as follows:

$$S_2 = (ba^k b)^k$$

$$P_2 = \{((u-1)(k+2)+1, u(k+2)) | u \in V\}$$

$$\cup \{((u-1)(k+2)+v+1, (v-1)(k+2)+u+1) | u, v \in \{1, \ldots, k\}, u \neq v\}$$

The parameter is $k' = k^2 + 2k$, while the maximum sequence length is $n' = n^2 + 2n$. Since the length of the sequences is bounded by a polynomial in $n$, this is a polynomial reduction. The sequence parameter $k'$ is a polynomial only in $k$ and is unrelated to $n$, so this reduction is also a parameterized reduction.

The target subsequence size is the same as the length of the second sequence, so we are really asking if the arced sequence $(S_2, P_2)$ is a subsequence of the arced sequence $(S_1, P_1)$. Figure 7.1 illustrates an example of this reduction.

**Proof:** We need to show that the graph $G = (V, E)$ has a clique of size $k$ if and only if $(S_1, P_1)$ and $(S_2, P_2)$ have an arc-preserving common subsequence of length $k' = k^2 + 2k$.

$\Rightarrow$: Assume there is some clique $V'$ of size $k$ in graph $G$. By definition of clique, $\forall u, v \in V'$, $u \neq v$, edge $(u, v) \in E$. Let $p : V \to \{0, 1, \ldots, k-1\}$ be a function that gives an order on $V'$, where $p(u) = |\{v \in V' | v < u\}|$. Thus $p(u)$ gives the position of vertex $u$ within the clique, starting from 0.

Construct $MS$ as follows.

$$\Phi_1 = \{((u-1)(n+2)+v+1, p(u)(k+2)+p(v)+2) \mid u, v \in V'\}$$

$$\Phi_2 = \{((u-1)(n+2)+1, p(u)(k+2)+1) \mid u \in V'\}$$

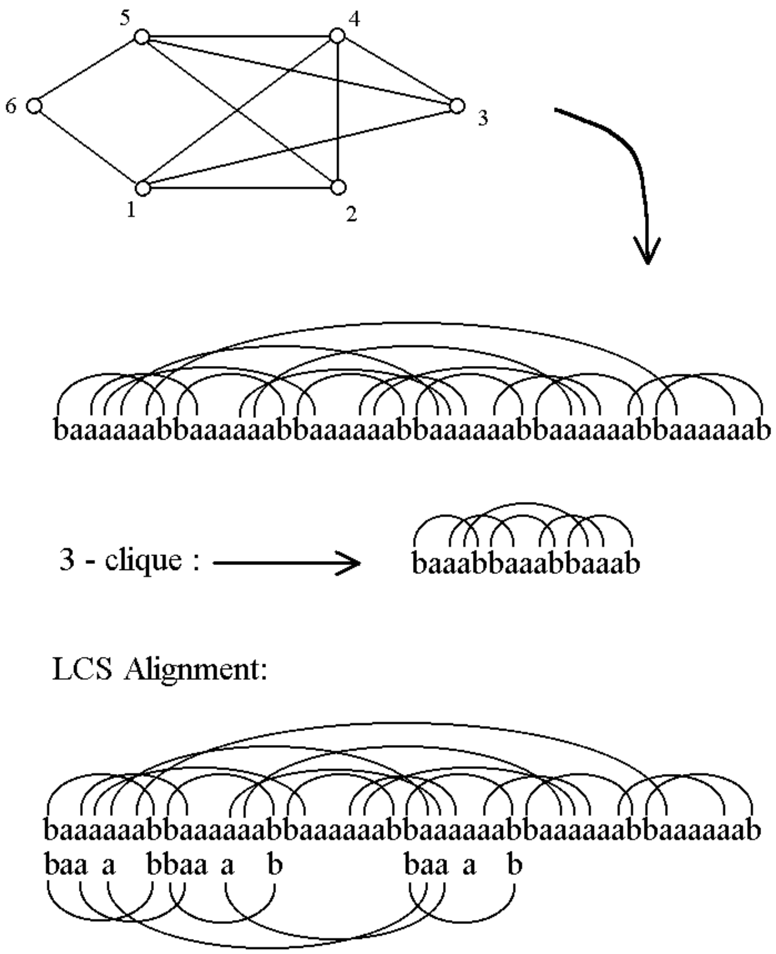


Figure 7.1: Example of transformation from Clique to $\Pi(cross, cross)$

$$\Phi_3 = \{(u(n+2), (p(u)+1)(k+2)) \mid u \in V'\}$$

$$MS = \Phi_1 \cup \Phi_2 \cup \Phi_3$$

Since $\Phi_1$, $\Phi_2$, and $\Phi_3$ do not overlap, $|MS| = k^2 + 2k$. We need to ensure that $MS$ meets the three criteria set out in the general problem.

*Criterion 1.* The mapping is one-to-one and preserves the order of the subsequence.

Observe that $p(u)$ is one-to-one and a strictly increasing function. $\Phi_1$ maps the groups of selected $a$ symbols from $S_1$ to groups of $a$ symbols in $S_2$. The use of $p$, which is strictly increasing on vertices from $V'$, ensures that the order of these groups is preserved and the order within each group is also preserved. $\Phi_2$ maps the $b$ symbols at the beginning of each selected group from $S_1$ to those at the beginning of groups in $S_2$, in order. Since the symbols thus mapped are at the beginning of corresponding groups mapped in set $\Phi_1$, the two sets do not conflict. Similarly, $\Phi_3$ maps the $b$ symbols at the end of each selected group from $S_1$ to those at the end of the corresponding group in $S_2$, and fits with the order established by $\Phi_1$ and $\Phi_2$.

*Criterion 2.* The arcs induced by the mapping are preserved.

$\Phi_1$: $\forall u, v \in V'$, let $i_1 = (u-1)(n+2) + v + 1$ and $i_2 = (v-1)(n+2) + u + 1$. Also let $j_1 = p(u)(k+2) + p(v) + 2$ and $j_2 = p(u)(k+2) + p(u) + 2$. Then, through membership in $\Phi_1$, $(i_1, j_1) \in MS$ and $(i_2, j_2) \in MS$. If $u \neq v$, then $(j_1, j_2) \in P_2$. Since $V'$ is a clique, there must be an edge $(u, v) \in E$ so $(i_1, i_2) \in P_1$. These match pairs $(i_1, j_1)$ and $(i_2, j_2)$ cover all arcs between the $a$ symbols of the pairs in $\Phi_1$.

$\Phi_2 \cup \Phi_3$: $\forall u \in V'$, let $i_1 = (u-1)(n+2) + 1$ and $i_2 = u(n+2)$. Also let $j_1 = p(u)(k+2) + 1$ and $j_2 = (p(u)+1)(k+2)$. Then, through membership in $\Phi_2 \cup \Phi_3$, $(i_1, j_1) \in MS$ and $(i_2, j_2) \in MS$. By the definition of $P_1$, $(i_1, i_2) \in P_1$; similarly, $(j_1, j_2) \in P_2$. These match pairs $(i_1, j_1)$ and $(i_2, j_2)$ cover all arcs between the $b$ symbols of the pairs in $\Phi_2 \cup \Phi_3$. There is no arc that links a symbol from a pair in $\Phi_1$ with a symbol from a pair in $\Phi_2 \cup \Phi_3$.

*Criterion 3.* The mapping produces a common subsequence.

Since order is already preserved, it suffices to check that $MS$ maps a symbol from $S_1$ to an identical symbol from $S_2$. From the definition of $S_1$ by the regular expression $(ba^nb)^n$, $S_1[i] = b$ if and only if $i \bmod (n+2) \in \{0, 1\}$; otherwise, $S_1[i] = a$. Similarly, since $S_2$ was defined by $(ba^kb)^k$, $S_2[j] = b$ if and only if $j \bmod (k+2) \in \{0, 1\}$; otherwise, $S_2[j] = a$.

$\Phi_1$: If $(i, j) \in \Phi_1$, $\exists u, v \in V'$ such that $i = (u-1)(n+2) + v + 1$ and $j = p(u)(k+2) + p(v) + 2$. Since $i \bmod (n+2) = v + 1$ which is not 0 or 1, $S_1[i] = a$. Since $j \bmod (k+2) = p(v) + 2$ which is not 0 or 1, $S_2[j] = a$ also.

$\Phi_2$: If $(i, j) \in \Phi_2$, $\exists u \in V'$ such that $i = (u-1)(n+2) + 1$ and $j = p(u)(k+2) + 1$. Since $i \bmod (n+2) = 1$, $S_1[i] = b$. Since $j \bmod (k+2) = 1$, $S_2[j] = b$ also.

$\Phi_3$: If $(i, j) \in \Phi_3$, $\exists u \in V'$ such that $i = u(n+2)$ and $j = (p(u)+1)(k+2)$. Since $i \bmod (n+2) = 0$, $S_1[i] = b$. Since $j \bmod (k+2) = 0$, $S_2[j] = b$ also.

$\Leftarrow$: Assume that $(S_2, P_2)$ is an arc-preserving subsequence of $(S_1, P_1)$. Therefore $\exists MS \subset \{1, 2, \ldots, n^2 + 2n\} \times \{1, 2, \ldots, k^2 + k\}$ such that $|MS| = k' = k^2 + k$ so $\forall j \in \{1, 2, \ldots, k^2 + k\}$, $\exists i \in \{1, 2, \ldots, n^2 + 2n\}$ such that $(i, j) \in MS$.

$S_1$ is the concatenation of $n$ substrings, where each substring is $ba^nb$. Similarly, $S_2$ is the concatenation of $k$ substrings, each of which is $ba^kb$. We first show that members of a single substring from $S_2$ must be mapped to members of a single substring from $S_1$.

All $b$ symbols from $S_2$ are mapped, and they are in pairs $j_1, j_2$ where for some $h \in \{1, \ldots, k\}$, $j_1 = (h-1)(k+2) + 1$ and $j_2 = h(k+2)$. Each such pair is also linked by an arc, $(j_1, j_2) \in P_2$. The indices $i_1$ and $i_2$ such that $(i_1, j_1)$ and $(i_2, j_2)$ are both $\in MS$ must be such that $(i_1, i_2) \in P_2$ and $S_1[i_1] = S_1[i_2] = b$. There are $n$ possible such pairs for $(S_1, P_1)$, and in this manner $k$ of them are selected.

To preserve the order of the subsequence, all $a$ symbols that are selected from $S_1$ must be enclosed within a selected pair of linked $b$ symbols, as they are in $S_2$. To show this, let $i_3$ be any index such that $S_1[i_3] = a$ and $\exists j_3$ where $(i_3, j_3) \in MS$. Then $S_2[j_3] = a$ and $\exists j_1, j_2$ where $j_1 < j_3 < j_2$ such that $S_2[j_1] = S_2[j_2] = b$ and $(j_1, j_2) \in P_2$. Let $i_1$ and $i_2$ be those indices such that $(i_1, j_1) \in MS$ and $(i_2, j_2) \in MS$. Then since the sequence order is preserved, $i_1 < i_3 < i_2$. Since arcs are also preserved, $(i_1, i_2) \in P_1$. This means that all the $a$ symbols from $S_1$ that are used in $MS$ come from $k$ substrings of $S_1$, with $k$ symbols per substring.

Let $V' = \{\frac{i}{n+2} | (i, j) \in MS \text{ and } j = h(k+2) \text{ for some } h \in \{1, \ldots, k\}\ \}$. Each $i$ is divisible by $n+2$ since it is a final endpoint of an arc (as is $j$) and $S_1[i] = S_2[j] = b$. This set of vertices $V'$ corresponds to the selected substrings of $S_1$. We need to show that $V'$ is a clique of $G = (V, E)$.

$\forall h_1, h_2 \in \{1, \ldots, k\}$, $h_1 \neq h_2$, let $j_1 = (h_1 - 1)(k+2) + h_2 + 1$ and $j_2 = (h_2 - 1)(k+2) + h_1 + 1$. Let $i_1, i_2$ be indices of $S_1$ such that $(i_1, j_1) \in MS$ and $(i_2, j_2) \in MS$. Since $(j_1, j_2) \in P_2$, $(i_1, i_2) \in P_1$. Let $u = \lceil \frac{i_1}{n+2} \rceil$ and $v = \lceil \frac{i_2}{n+2} \rceil$. Note that $u \in V'$ and $v \in V'$ since both $i_1$ and $i_2$ are part of a selected substring of $S_1$. Since $(i_1, i_2) \in P_1$, $i_1 = (u-1)(n+2) + v + 1$ and $i_2 = (v-1)(n+2) + u + 1$. Thus $(u, v) \in E$. This is true for all pairs $(h_1, h_2)$, $h_1 \neq h_2$, and each distinct $(h_1, h_2)$ pair produces a distinct $(u, v)$ edge between vertices of $V'$. Thus $V'$ is a clique of the graph $G$. $\square$

**Lemma 7.3.** Independent Set is polynomially reducible to $\Pi(cross, plain)$.

**Reduction.** From $k$-Independent Set of a graph $G = (V, E)$, where $V = \{1, 2, \ldots, n\}$.

Construct $S_1[1..(n^2+n)]$ and $P_1$ as follows:

$$S_1 = (ba^n)^n$$

$$P_1 = \{((u-1)(n+1)+v+1, (v-1)(n+1)+u+1) \mid (u,v) \in E\}$$

and let $S_2[1..(2nk-k^2+n)] = (b^{n-k+1}a^n)^k b^{n-k}$, with $P_2 = \emptyset$.

The graph $G$ has an independent set $V' \subseteq V$ of size $k$ if and only if there is a mapping set $MS$ of size $nk+n$ for $(S_1, P_1)$ and $(S_2, P_2)$. An example of this reduction is illustrated in figure 7.2.

Notice that a common subsequence of $S_1$ and $S_2$ with length $nk+n$ must contain all the $a$ symbols from sequence $S_2$ ($nk$ of them) and all the $b$ symbols from sequence $S_1$ ($n$ of them).

**Proof:**

$\Rightarrow$: Assume there is some independent set $V' \subseteq V$ of $G$, $|V'| = k$. By definition of independent set, $\forall u, v \in V'$, $(u,v) \notin E$. Define the function $p : V \rightarrow \{0, 1, \ldots, k\}$ by $p(u) = |\{v \ : \ v \in V' \text{ and } v < u\}|$. Define the function $e : C \rightarrow \{0, 1, \ldots, (n-k)\}$ by $e(u) = |\{v \ : \ v \in V - V' \text{ and } v \leq u\}|$. These functions have some useful properties,

$$p(u+1) = \begin{cases} p(u)+1 & \text{if } u \in V' \\ p(u) & \text{if } u \notin V' \end{cases}$$

$$e(u) = \begin{cases} e(u-1) & \text{if } u \in V' \\ e(u-1)+1 & \text{if } u \notin V' \end{cases}$$

which shall be used later. To make use of these properties to join otherwise different cases, the domain of $p$ is extended to include $n+1$, with $p(n+1) = k$.

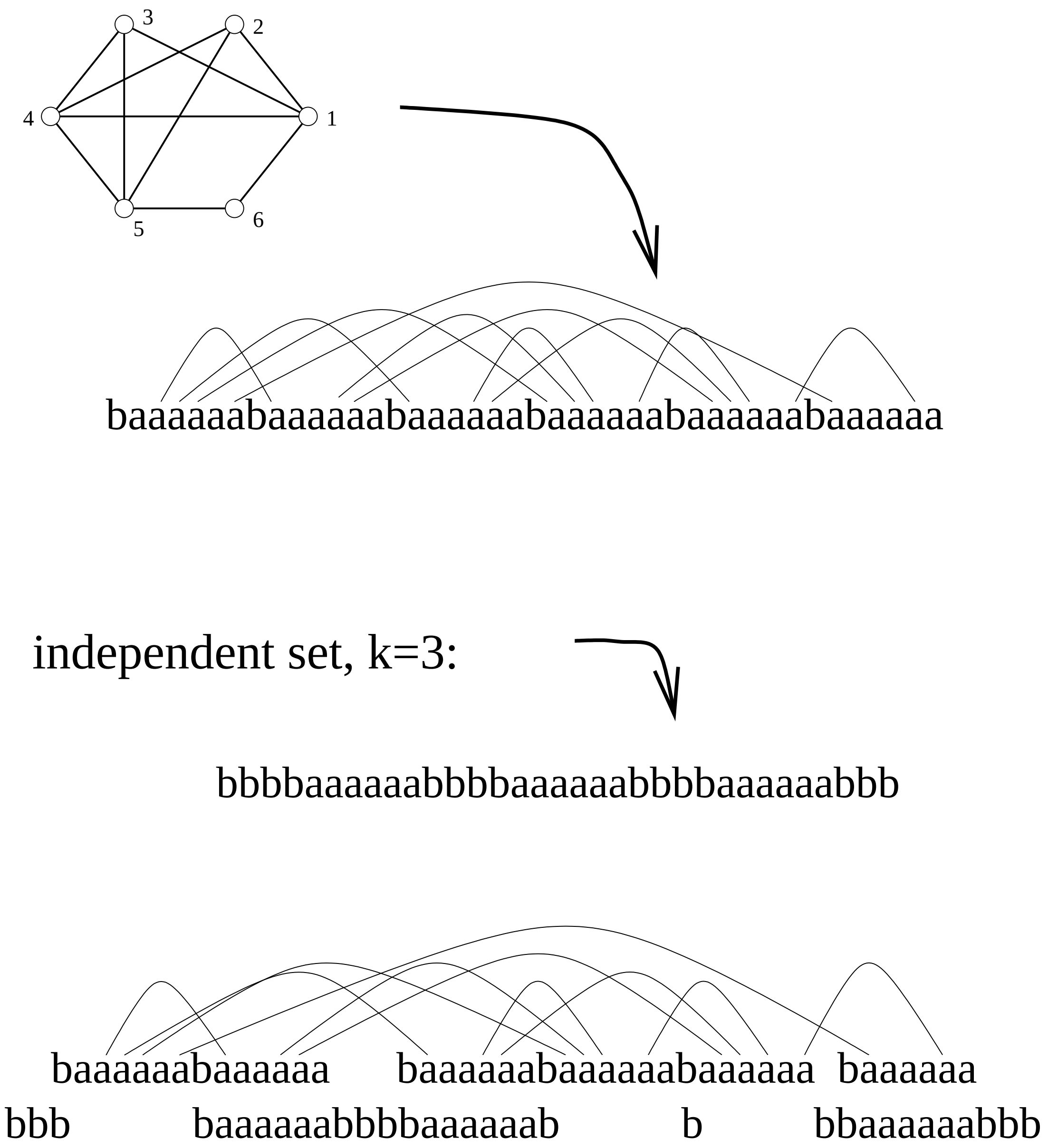


Figure 7.2: Example of transformation from Independent Set to $\Pi(cross, plain)$

Construct $MS$ as follows.

$$\Phi_1 = \{((u-1)(n+1)+1+i, p(u)\cdot(2n-k+1)+n-k+1+i) \mid u \in V',\ 1 \le i \le n\}$$

$$\Phi_2 = \{((u-1)(n+1)+1, p(u)\cdot(2n-k+1)+n-k+1) \mid u \in V'\}$$

$$\Phi_3 = \{((u-1)(n+1)+1, p(u)\cdot(2n-k+1)+e(u)) \mid u \in V-V'\}$$

$$MS = \Phi_1 \cup \Phi_2 \cup \Phi_3$$

Since the three sets $\Phi_1$, $\Phi_2$, and $\Phi_3$ do not overlap, and their sizes are $|\Phi_1| = nk$, $|\Phi_2| = k$, and $|\Phi_3| = n-k$, the entire mapping set $MS$ has size $|MS| = nk+n$.

We need to ensure that this $MS$ satisfies the three necessary criteria.

*Criterion 1.* The mapping is one-to-one and preserves the order of the sequences.

When applied only to $V'$, the function $p$ is one-to-one and strictly increasing. So the mapping done through $\Phi_1$ and $\Phi_2$ is one-to-one (and strictly increasing); the use of $p$ preserves the order of the mapped groups, and the index $i$ used in $\Phi_1$ preserves the order within each group. The subset $\Phi_3$, however, uses $p$ on vertices from $V-V'$, where $p$ is not one-to-one. However, it also incorporates $e(u)$ which is one-to-one and strictly increasing on $V-V'$. The resulting expression $(p(u)\cdot(2n-k+1)+e(u))$ is strictly increasing since $e(u) \le (n-k)$. Since $(u-1)(n+1)+1$ is also one-to-one and strictly increasing, so is the mapping done by $\Phi_3$.

It only remains to check that the union of $\Phi_1 \cup \Phi_2$ with $\Phi_3$ is also one-to-one and strictly increasing (so that the three subsets of $MS$ do not conflict with each other). The proof of this property is by induction on the vertex set $V$.

*Basis:* We start with $1 \in V$. If $1 \in V'$, then pairs from $\Phi_2$ and $\Phi_1$ are used to match the first $ba^n$ groups, mapping $S_1[1..n+1]$ to $S_2[n-k+1..2n-k+1]$. The first $n-k$ characters in $S_2$, all $b$ symbols, are skipped. Otherwise (if $1 \notin V'$), a pair from

$\Phi_3$ is used to match the initial $b$ at $S_1[1]$ to $S_2[0 \cdot (2n-k+1)+(1-0)] = S_2[1]$, and the next $n$ symbols in $S_1$ are skipped. In either case, the mapping is one-to-one and strictly increasing, and matches or skips each symbol in $S_1[1..n+1]$ and in $S_2[1..p(2) \cdot (2n-k+1)+e(1)]$ once each and in sequence order.

*Inductive Hypothesis:* Assume that after the first $m$ vertices in $V$ have been processed, we have matched or skipped each symbol of $S_1[1..m(n+1)]$ and $S_2[1..p(m+1) \cdot (2n-k+1)+e(m)]$, once each and in order. We then need to show that after vertex $m+1$ has been processed, we will have matched or skipped each symbol of $S_1[1..(m+1)(n+1)]$ and $S_2[1..p(m+2) \cdot (2n-k+1)+e(m+1)]$, once each and in sequence order.

*Inductive Step:* For vertex $m+1$, if $m+1 \in V'$ then $S_1[m(n+1)+1..(m+1)(n+1)]$ is matched using pairs from $\Phi_1$ and $\Phi_2$, one-to-one and in order, to $S_2[p(m+1) \cdot (2n-k+1)+n-k+1..(p(m+1)+1) \cdot (2n-k+1)]$. The symbols in $S_2[p(m+1) \cdot (2n-k+1)+e(v)+1..p(m+1) \cdot (2n-k+1)+n-k]$ and $S_2[(p(m+1)+1) \cdot (2n-k+1)+1..(p(m+1)+1) \cdot (2n-k+1)+e(m+1)]$ are skipped. Since $m+1 \in V'$, $p(m+1)+1 = p(m+2)$. Note that if $S_2[x..y]$ is to be skipped and $x > y$, then no symbols from $S_2$ are skipped. On the other hand, if $m+1 \notin V'$, then $S_1[m(n+1)+1]$ is matched by a pair (from $\Phi_3$) to $S_2[p(m+1) \cdot (2n-k+1)+e(m+1)]$, and the next $n$ symbols in $S_1$ are skipped. Since $m+1 \notin V'$, $p(m+1) = p(m+2)$. Either way, once vertex $m+1$ has been processed, all symbols in $S_1[1..(m+1)(n+1)]$ and $S_2[1..p(m+2) \cdot (2n-k+1)+e(m+1)]$ have been either skipped or matched, once each and in sequence order.

Therefore by induction, after all $n$ vertices $\in V$ have been processed, all symbols in $S_1[1..n(n+1)]$ and $S_2[1..p(n+1) \cdot (2n-k+1)+e(n)] = S_2[1..k(2n-k+1)+(n-k)]$ have been either skipped or matched, one-to-one and in strictly increasing order. Thus Criterion 1 is satisfied.

*Criterion 2.* The arcs induced by the mapping are preserved.

Since $P_2 = \emptyset$, we need to show that $\forall i_1, i_2$ such that $(i_1, j_1) \in MS$ and $(i_2, j_2) \in MS$ (for any such $j_1$ and $j_2$), the pair $(i_1, i_2) \notin P_1$.

All arcs are between $a$ symbols, so the only candidates for $(i_1, j_1)$ and $(i_2, j_2)$ are pairs from $\Phi_1$. Assume that there exist $(i_1, j_1) \in \Phi_1$ and $(i_2, j_2) \in \Phi_1$ such that $(i_1, i_2) \in P_1$. Since $(i_1, i_2) \in P_1$, then there exist $u, v \in V$ such that $i_1 = (u-1)(n+1)+v+1$ and $i_2 = (v-1)(n+1)+u+1$, with $(u, v) \in E$. Since $(i_1, j_1) \in \Phi_1$ and $(i_2, j_2) \in \Phi_1$, then $u \in V'$ and $v \in V'$. But $V'$ is an independent set, and $(u, v) \in E$ violates this property. Thus no such pair $(i_1, i_2) \in P_1$ can exist, and Criterion 2 is satisfied.

*Criterion 3.* The mapping produces a common subsequence.

Since order is already preserved, it suffices to check that $MS$ maps a symbol from $S_1$ to an identical symbol from $S_2$. This can be checked for each subset of $MS$.

The set of pairs $\Phi_1$ maps $S_1[(u-1)(n+1)+1+i]$ to $S_2[p(u)\cdot(2n-k+1)+n-k+1-i]$, for all $u \in V'$ and $1 \leq i \leq n$. Since $2 \leq i+1 \leq n+1$ and $S_1 = (ba^n)^n$, $S_1[(u-1)(n+1)+1+i] = a$. Since $n-k+2 \leq n-k+1-i \leq 2n-k+1$ and $S_2 = (b^{n-k+1}a^n)^k b^{n-k}$, $S_2[p(u)\cdot(2n-k+1)+n-k+1-i] = a$.

The set of pairs $\Phi_2$ maps $S_1[(u-1)(n+1)+1]$ to $S_2[p(u)\cdot(2n-k+1)+n-k+1]$, for all $u \in V'$. Since $S_1 = (ba^n)^n$, $S_1[(u-1)(n+1)+1] = b$. Since $S_2 = (b^{n-k+1}a^n)^k b^{n-k}$, $S_2[p(u)\cdot(2n-k+1)+n-k+1] = b$.

The set of pairs $\Phi_3$ maps $S_1[(u-1)(n+1)+1]$ to $S_2[p(u)\cdot(2n-k+1)+(u-p(u-1))]$, for all $u \notin V'$. Since $S_1 = (ba^n)^n$, $S_1[(u-1)(n+1)+1] = b$. Since $1 \leq (u-p(u-1)) \leq n-k+1$ and $S_2 = (b^{n-k+1}a^n)^k b^{n-k}$, $S_2[p(u)\cdot(2n-k+1)+(u-p(u-1))] = b$.

Since $\Phi_1$, $\Phi_2$, and $\Phi_3$ all map between identical symbols, the mapping does produce a common subsequence and Criterion 3 is satisfied.

Thus $MS$ is a valid mapping that provides and arc-preserving longest common subsequence of length $nk + n$.

$\Leftarrow$: Let $MS$ be some mapping set that meets the criteria, with $|MS| = nk + n$. Notice that $S_1$ has $n^2$ $a$ symbols and $n$ $b$ symbols, while $S_2$ has $kn$ $a$ symbols and $(n-k+1)(k+1)-1$ $b$ symbols. Since $k \leq n$, any subsequence of length $nk+n$ that is common to both $S_1$ and $S_2$ must have $kn$ $a$ symbols and $n$ $b$ symbols. This property means that it must match all of the $a$ symbols in $S_2$ and all of the $b$ symbols in $S_1$.

Since all $b$ symbols in $S_1$ are matched, the $a$ symbols in $S_1$ that are matched must be divided into groups of no more than $n$ $a$ symbols. We need to show that there are $k$ such groups.

All $a$ symbols in $S_2$ are matched, and these symbols are divided into $k$ groups of $n$ $a$ symbols each. Since the subsequence symbol order of $S_2$ must be preserved, the common subsequence must have all $a$ symbols in groups of at least $n$. But the sequence symbol order of $S_1$ must also be preserved, which has all matched $a$ symbols in groups of no more than $n$ (since all $b$ symbols in $S_1$ are matched). Therefore all $a$ symbols in the common subsequence are divided into $k$ groups of exactly $n$ $a$ symbols each, with each group separated by one or more $b$ symbols. Since all $b$ symbols in $S_1$ are matched, these $a$ symbols are from $k$ different groups in $S_1$.

Let $V' \subseteq V$ be defined as $V' = \{u \mid \text{ all } a \text{ symbols from group } u \text{ in } S_1 \text{ are matched } \}$. Since $k$ groups are matched, $|V'| = k$. We need to show that $V'$ is an independent set in $G$.

Let $u, v$ be any two vertices $\in V'$. We need to show that $(u, v) \notin E$. Let $i_1 = (u-1)(n+1) + v + 1$ and $i_2 = (v-1)(n+1) + u + 1$. Since $i_1$ is part of group $u$ in $S_1$, $i_1$ is matched to some position $j_1$ of $S_2$. Since $i_2$ is part of group $v$ in $S_1$, $i_2$ is matched to some position $j_2$ of $S_2$. As $P_2 = \emptyset$, $(j_1, j_2) \notin P_2$, so $(i_1, i_2) \notin P_1$. By the construction of $P_1$, if $(u, v) \in E$ then $((u-1)(n+1)+v+1, (v-1)(n+1)+u+1) \in P_1$, so $(u, v) \notin E$. Since this is true $\forall u, v \in V'$, $V'$ is an independent set in $G$. $\square$

## 7.5 Classical Complexity

Since the added arc preservation requirements are checkable in polynomial time, all variants of $\Pi$ are in NP. Unfortunately, many of the less restrictive variations are also NP-hard. The more limited arc structures can be incorporated into the longest common subsequence problem without increasing its complexity, and the middle variants have no currently known polynomial time algorithm.

**Theorem 7.4.** $\Pi(unlim, plain)$ is NP-complete.

**Proof:** The problem $\Pi(unlim, plain)$ is obviously in NP, and Independent Set, a known NP-hard problem, is polynomially reducible to it by Lemma 7.1. $\square$

**Theorem 7.5.** $\Pi(cross, plain)$ is NP-complete.

**Proof:** As Independent Set is also polynomially reducible to $\Pi(cross, plain)$ by Lemma 7.3, it is also NP-complete. $\square$

**Theorem 7.6.** $\Pi(chain, chain)$ can be solved in $O(nm)$ time.

**Proof:** The following section presents an algorithm for $\Pi(cross, cross)$ that takes $O(f(k) \cdot nm)$ time, where $k$ is the maximum arc cutwidth of the sequences. When $k = 1$, this algorithm solves $\Pi(chain, chain)$. $\square$

These results are summarized in Table 7.2.

| | | | | | |
|---|---|---|---|---|---|
| unlim | NP-c | | | | |
| cross | NP-c | NP-c | | | |
| nest | NP-c | NP-c | NP | | |
| chain | NP-c | NP-c | NP | $O(nm)$ | |
| plain | NP-c | NP-c | NP | $O(nm)$ | $O(nm)$ |
| | unlim | cross | nest | chain | plain |

Table 7.2: Classical complexity results for problem $\Pi$ with different restrictions

## 7.6 Parameterized Complexity

As many variants of $\Pi$ are NP-hard or have no currently known polynomial time algorithm, they are good targets for parameterized complexity analysis. There are a few natural parameters for the problems.

- $l$, length of desired subsequence
- $s$, levels of nested arcs (for non-crossing arcs)
- $k$, cutwidth of arc structure
- $d$, bandwidth of arc structure (so $(i_2 - i_1) \leq d$ for any arc $(i_1, i_2)$)

The length of desired subsequence $l$ is independent of the other parameters. The others are related; $s \equiv k$, and $k \leq d$ for all restriction levels except *unlim*.

### 7.6.1 Parameterized by Length

**Theorem 7.7.** The general problem $\Pi(unlim, unlim)$ is in $W[1]$ if parameterized by subsequence length $l$.

**Proof:** For any instance $I = ((S_1, P_1), (S_2, P_2), l)$ of $\Pi(unlim, unlim)$, there is a corresponding decision circuit $C_I$ whose accepted weight $l$ input assignments correspond to a solution of the problem instance $I$ (ie., an arc-preserving subsequence of length l).

From $I$, construct $C_I$ with one input bit for every potential symbol match $(i, j)$. There are thus $nm$ bits of input to the circuit. For every pair of input bits, labeled $(i_1, j_1)$ and $(i_2, j_2)$, add a AND gate with the two bits as the gate's inputs if and only if at least one of the following hold:

- $i_1 = i_2$
- $j_1 = j_2$
- $i_1 < i_2$ and $j_1 > j_2$
- $(i_1, i_2) \in P_1$ and $(j_1, j_2) \notin P_2$
- $(i_1, i_2) \notin P_1$ and $(j_1, j_2) \in P_2$

Add a large OR gate with inputs from:

- the output from all the AND gates
- all circuit inputs $(i, j)$ where $S_1[i] \neq S_2[j]$

Add a NOT gate on the output line from the large OR gate.

This circuit will accept on a weight $l$ input if and only if the input bits correspond to a subsequence mapping between $S_1$ and $S_2$ that preserves arcs. Since the circuit family described has weft 1, the problem $\Pi(unlim, unlim) \in W[1]$. □

**Theorem 7.8.** $\Pi(unlim, plain)$ parameterized by $l$ is W[1]-complete.

**Proof:** The problem is in W[1] by Theorem 7.7, and $k$-Independent Set reduces to it by a strong uniform parameterized reduction in Lemma 7.1, with $l = k$. $\Box$

**Theorem 7.9.** $\Pi(cross, cross)$ parameterized by $l$ is W[1]-complete.

**Proof:** $k$-Clique reduces to $\Pi(cross, cross)$ by a strong uniform parameterized reduction in Lemma 7.2, with $l = k^2 + 2k$, so it is also W[1]-complete. $\Box$

### 7.6.2 Parameterized by Cutwidth and Bandwidth

While the problem $\Pi(cross, cross)$ is NP-complete, and is also W[1]-complete when parameterized by desired subsequence length $l$, the problem becomes fixed-parameter tractable if it is instead parameterized by the arc cutwidth. This cutwidth is defined as the maximum number of arcs that cross or end at any arbitrary position of the sequence. If both sequences have their cutwidth bounded by some $k$, the problem can be solved in time $O(f(k)nm)$ where $f(k)$ is a function in $k$ independent of both $n$ and $m$. Sufficient bounds on $k$ will make the problem tractable.

On initial examination, this problem should be able to be solved by splitting the tables whenever an initial endpoint is encountered. Restricting the cutwidth of the sequences, then, should make this problem fixed-parameter tractable. These tables can be used to store the initial endpoint matches that are made on the various computation paths, and these paths can then be searched when a final endpoint is encountered. This searching can determine if any maximum computation path has matched the initial endpoints that correspond to the final endpoints encountered. However, this network of paths can, potentially, cover all $4^k n^2$ positions in the tables, and may need to be searched completely.

The network of computation paths cannot be searched simply using a breadth-first search technique, nor can we only maintain a list of all arc assignments on the

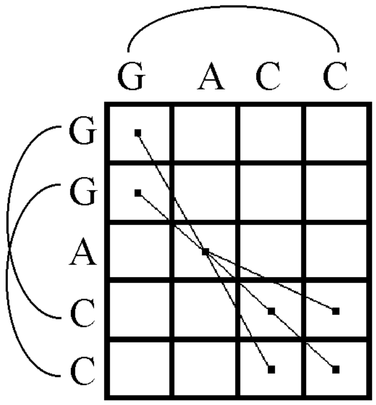


Figure 7.3: Computation Path Network Example

maximum computation paths that lead to each table entry. Since matching the final endpoint depends on whether (and to what) the initial endpoint was matched, the network of paths also includes path combinations that are not allowed. Symbol matches inside an arc would merge lists that need to be kept separate. In Figure 7.3, for example, the entry that matches the two $a$ symbols brings together two computation paths that match initial endpoint $g$ symbols; however, the subsequent matching of the final endpoint symbols can only be done one way for each initial match, not two as the network of paths would indicate. Looking for an initial match in this network, then, would require searching each possible path through the $4^k n^2$ different table positions in order to correctly compute the new table value.

Since searching through the network of paths is very costly, the different initial endpoint matches made on valid computation paths can instead be kept in a tree. Each of the table positions should have its own tree of all valid computation paths. In order to be able to minimize the length of the paths in the tree, each position will need to have its own copy, instead of referencing trees at previous positions.

These paths are kept short by removing initial endpoint matches of arcs that are no longer active (whose final endpoints have been encountered). This editing reduces the length of each path to at most $k$, and the storing of the endpoint matches in a tree means that all paths to be searched are valid ones. Thus the time to search for an initial endpoint match is reduced while both the space used and the complexity of the data structure are increased.

**Theorem 7.10.** $\Pi(cross, cross)$ parameterized by cutwidth $k$ is Fixed-Parameter Tractable, and can be solved in time $O(9^k nm)$.

**Proof:** A solution for $\Pi(cross, cross)$ is found by the following algorithm.

**Algorithm 7.11.**

- Step 1. For each of $P_1$ and $P_2$, partition the set of arcs into $k$ sets, where each set contains a chain of arcs that do not cross or nest. Number these chains from 0 through $k-1$.

- Step 2. For each of $(S_1, P_1)$ and $(S_2, P_2)$, look at each subset of the set of chains. For each subset of chains of $P_1$, create a copy of the sequence $S_1$ with the initial endpoints of all arcs in those chains removed and replaced by some $\lambda \notin \Sigma$. The set of sequences thus created is $\mathcal{S}_1$, and is generally indexed by $h_1$, where $h_1 = \sum_{i \in subset} 2^i$. For $(S_2, P_2)$, create the set of sequences $\mathcal{S}_2$ in the same way, indexing it using $h_2$.

- Step 3. For each combination of $h_1$ and $h_2$, create a two-dimensional table, $n \times m$, that uses strings $\mathcal{S}_1[h_1]$ and $\mathcal{S}_2[h_2]$. These tables will be used to calculate the length of the longest common subsequence. Each table position includes both a value $T^{(h_1,h_2)}[i,j]$, the length of the longest common subsequence so far, and a tree $M^{(h_1,h_2)}[i,j]$ of the initial arc endpoints matched along the computation paths that produce that value. These matches between initial endpoints are tentative assignments that will be checked when the final endpoints of the arcs are encountered.

- Step 4. Calculate the longest common arc-preserving subsequence of $(S_1, P_1)$ and $(S_2, P_2)$ by traversing the tables. A table is considered *active* if one arc from each of the chains in its subset is active.

At each step, the values for that position in all active tables are calculated. The trees are also merged and manipulated. This tree data structure needs to support the following operations.

- *merge*: is applied to a finite list of trees and their corresponding subsequence length values, and returns the merge of those trees that have the maximum corresponding values. When the trees are merged, they are copied and also simplified from the root down by uniting identical children of the same parent node.

- *test*: looks for a given arc assignment pair in the tree; returns *true* if it is found, *false* otherwise.

- *prune*: given a tree and an arc assignment pair, removes all paths that do not contain the pair, and then removes the pair itself.

- *trim*: given a tree, an arc number $k'$, and a flag value, removes all nodes in the tree that contain an arc assignment that involves an arc with that number $k'$. This operation checks either the $i$ or $j$ values, depending on the value of the flag.

- *extend*: given a tree and an arc assignment pair, add the pair to the tree as its new root.

The complexity of each of these operations, except for *extend*, is proportional to the size of the tree, so $\in O(|M^{(h_1,h_2)}[i,j]|)$.

These operations are used to keep the trees up to date as the table values are being calculated. The basic longest common subsequence formula

$$T[i,j] = \max(T[i-1,j], T[i,j-1], T[i-1,j-1] + w(S_1[i], S_2[j]))$$

where

$$w(x,y) = \begin{cases} 1 & \text{if } x = y \\ 0 & \text{otherwise} \end{cases}$$

is used. This calculation is varied when arc endpoints are encountered, as follows:

1. When an initial arc endpoint is encountered, all tables that include that arc's chain in its subset are activated and initialized by copying needed values into the preceding row or column.

2. When a final arc endpoint is encountered, the table without the initial endpoint is calculated normally. The other table, where the initial endpoint was allowed to match, is calculated without matching the final endpoint. These two tables are then *merged* to find the maximum. The trees are *trimmed* to remove all assignments that use that arc.

3. If a pair of initial endpoints is encountered, one from each sequence, the algorithm attempts to match their arcs. The tables are activated and initialized as in 1, but one table – that has both initial endpoints – requires the tree at that position to be *extended* by adding that arc assignment pair.

4. If a pair of final endpoints is encountered, the algorithm must determine (using *test*) if the corresponding pair of initial endpoints are in the tree. If they are, and the maximum value is produced by matching the final endpoints, those endpoints are matched and the tree is *pruned*. Otherwise, the trees and tables are *merged* as in 2.

A more detailed lower-level description of this algorithm is in Appendix A.

After the table computation, the decision algorithm returns *true* if and only if the length of the longest common arc-preserving subsequence, stored in $T^{(0,0)}[n, m]$, is at least $l$, and returns *false* otherwise. This algorithm computes table entries for up to $4^k$ tables, each table having $nm$ entries.

**Time Complexity Analysis:**

The computation of each table entry $M^{h_1,h_2}[i,j]$ takes $O(|M^{h_1,h_2}[i,j]|)$ time. In this algorithm, the trees are kept minimal so that they only store matches of starting endpoints of currently active arcs.

To find the size of $M^{(h_1,h_2)}[i,j]$:

Each path in the tree is a sequence both on $i$ values and on $j$ values. Let $p_1(i')$ be the position of $i'$ among the starting endpoints of active arcs on $S_1$, so $p_1(i') = |\{(i_1, i_2) \ : \ i_1 \leq i' \leq i_2 \text{ and } i_1 \leq i' \ \}|$. Similarly, let $p_2(j')$ be the position of $j'$ among the starting endpoints of active arcs on $S_2$.

At each position in the tree, replace the label $(i', j')$ with $(p_1(i'), p_2(j'))$. The sequences of $p_1(i')$ values and $p_2(j')$ values along any path from root to leaf are strictly decreasing. The number of nodes in such a tree with $(x, y)$ as its root is given by the following recurrence relation.

$$S(1, y) = 1 \ \ \forall y \quad \text{and} \quad S(x, 1) = 1 \ \ \forall x$$

$$S(x,y) = 1 + \sum_{t=1}^{x-1}\sum_{r=1}^{y-1} S(t,r) \quad \forall y > 1, x > 1$$

which converts to

$$S(x,y) = S(x-1,y) + S(x,y-1)$$

and is equivalent to the closed form

$$S(x,y) = \binom{x+y-2}{x-1}$$

Since the root of the entire tree $M^{(h_1,h_2)}[i,j]$ may be blank with all possible children, the number of nodes in $M^{(h_1,h_2)}[i,j]$ is at most $\binom{r+s}{r}$ where $r$ is the number of active arcs from $(S_1, P_1)$ and $s$ is the number of active arcs from $(S_2, P_2)$.

To determine the total size of the trees over all the tables, consider that both sequences have bounded cutwidth $k$, and that the algorithm is currently computing any specific position $[i,j]$ in each of the tables. Each time the tables are split, half of the tables are allowed to include the new assignment, while the other half cannot include it. So for each possible $r$ and $s$, the number of tables that can have $r$ active arcs from $S_1$ and $s$ active arcs from $S_2$ is $\binom{k}{r} \cdot \binom{k}{s}$. Thus the total number of nodes at position $[i,j]$ over all the $4^k$ tables is at most

$$S(k) = \sum_{r=0}^{k}\sum_{s=0}^{k} \binom{k}{r}\binom{k}{s}\binom{r+s}{s}\ .$$

By substituting Vandermonde's convolution, $\binom{r+s}{r} = \sum_{t=0}^{r}\binom{r}{t}\binom{s}{r-t}$, we get

$$\begin{aligned} S(k) &= \sum_{r=0}^{k}\sum_{s=0}^{k}\sum_{t=0}^{r} \binom{k}{r}\binom{k}{s}\binom{r}{t}\binom{s}{r-t} \\ &= \sum_{r=0}^{k}\sum_{s=0}^{k}\sum_{t=0}^{r} \binom{k}{r}\binom{k}{s}\binom{r}{t}\binom{s}{t}\ . \end{aligned}$$

Applying trinomial revision to both $\binom{k}{r}\binom{r}{t}$ and $\binom{k}{s}\binom{s}{t}$ yields

$$S(k) = \sum_{r=0}^{k}\sum_{s=0}^{k}\sum_{t=0}^{r}\binom{k}{t}\binom{k-t}{k-r}\binom{k}{t}\binom{k-t}{k-s}$$

$$= \sum_{t=0}^{k}\sum_{r=t}^{k}\sum_{s=0}^{k-t}\binom{k}{t}^2\binom{k-t}{k-r}\binom{k-t}{k-s}$$

$$= \sum_{t=0}^{k}\sum_{r=0}^{k-t}\sum_{s=0}^{k-t}\binom{k}{t}^2\binom{k-t}{r}\binom{k-t}{s}$$

collapsing the sums on $r$ and $s$ gives

$$S(k) = \sum_{t=0}^{k}\binom{k}{t}^2 2^{2t} \;=\; \sum_{t=0}^{k}\left(\binom{k}{t}2^t\right)^2 \; .$$

To find an upper bound on this sum, we use the Cauchy-Schwarz inequality

$$\sum_{t=0}^{k}\left(\binom{k}{t}2^t\right)^2 \le \left(\sum_{t=0}^{k}\binom{k}{t}2^t\right)^2$$

and notice that $\sum_{t=0}^{k}\binom{k}{t}2^t \;=\; 3^k$ (as an expansion of $(2+1)^k$). Thus

$$S(k) \le 9^k \; .$$

We can obtain an asymptotic estimate for a lower bound on $S(k)$ by looking at one term of the sum, where $t = \frac{2k}{3}$. From this term,

$$S(k) \ge \left(\binom{k}{\frac{k}{3}}2^{\frac{2k}{3}}\right)^2 \; .$$

This term can be expanded using factorials

$$\left(\frac{k!}{\frac{2k}{3}!\,\frac{k}{3}!}\cdot 2^{\frac{2k}{3}}\right)^2$$

and estimated using Stirling's approximation ($n! \sim \sqrt{2\pi n}(\frac{n}{e})^n$) to get

$$\left(\binom{k}{\frac{k}{3}}2^{\frac{2k}{3}}\right)^2 \;\sim\; \frac{9^k}{\frac{4\Pi}{9}k} \; .$$

This approximation reveals that the upper bound of $9^k$ is very close to $S(k)$; it can be off by at most a factor of $\frac{4\Pi}{9}k$.

Since $S(k)$ is the upper bound on the total size of the trees for each position $(r, s)$ over all tables, and there are $nm$ such positions to be computed, the algorithm runs in time $O(9^k nm)$. □

**Corollary 7.12.** $\Pi(cross, cross)$ parameterized by bandwidth $d$ can be solved in time $O(9^d nm)$.

**Proof:** Since the arcs cannot share endpoints, cutwidth $k$ is always no more than bandwidth $d$. Two sequences of bandwidth $d$ thus both have cutwidth $\leq d$, so the above algorithm can be run on them in time $\in O(9^d nm)$. Note that this conversion works because we only need a bound on the cutwidth; the algorithm does not need its exact value. □

**Theorem 7.13.** $\Pi(nest, nest)$ parameterized by nesting depth $s$ can be solved in time $O(s^2 4^s nm)$.

**Proof:** Follows from algorithm 7.11, with modifications to take advantage of non-crossing arcs. For arc structures without intersecting arcs, nesting depth is the natural equivalent to cutwidth for intersecting arcs. Since the arcs do not cross, the tree data structure operations that keep track of the active arc assignments on maximum computation paths can be modified to improve the time complexity. When a final endpoint is encountered, all matches of its corresponding starting endpoint will be at the top of the tree, either the root or the children immediately below a blank root. At most $s^2$ nodes need to be examined. All editing operations also occur at the root or its children, so anything further down the tree can be stored elsewhere and referenced, rather than copying the entire tree for each table entry. The tree operations can be altered to only manipulate the nodes near the root, as follows:

- *merge* : is applied to a finite list of trees $\langle M^{(h_1,h_2)}[i,j]\rangle$ with corresponding values $\langle T^{(h_1,h_2)}[i,j]\rangle$, and returns the merge of those trees based on the values, as follows.
  - compares the values and finds their maxima
  - if only one maximum, *merge* returns the corresponding tree (with its own copy of the root and the root's children)
  - if more than one maxima, the trees are merged by copying their roots and making them children of a new blank root, then simplifying the tree only at the root by uniting any identical children of the root.
- $test(M^{(h_1,h_2)}[i,j],(i',j'))$ : returns *true* if $(i',j')$ in $M^{(h_1,h_2)}[i,j]$, *false* otherwise
  - change to only look at root of tree and its immediate children
  - time complexity $O(s^2)$
- $prune(M^{(h_1,h_2)}[i,j],(i',j'))$ : remove all paths that do not contain $(i',j')$, and remove $(i',j')$, returning the resulting tree
  - if the root is blank, remove all of its children that are not equal to $(i',j')$, and their descendants; remove the children equal to $(i',j')$, moving their descendants up; if the root is equal to $(i',j')$, remove it
  - time complexity $O(s^2)$
- $trim(M^{(h_1,h_2)}[i,j], flag, k')$ : if $flag = 0$, remove all entries $(i',j')$ where $n_1[i'] = k'$; if $flag = 1$, remove all entries $(i',j')$ where $n_2[j'] = k'$. Simplify and return the resulting tree

| | | | | | |
|---|---|---|---|---|---|
| unlim | W[1]-c ($l$) | | | | |
| cross | W[1]-c ($l$) | W[1]-c ($l$)<br>$O(9^k nm)$<br>$O(9^d nm)$ | | | |
| nest | W[1]-c ($l$) | $O(9^k nm)$<br>$O(9^d nm)$ | $O(s^2 4^s nm)$<br>$O(d^2 4^d nm)$ | | |
| chain | W[1]-c ($l$) | $O(9^k nm)$<br>$O(9^d nm)$ | $O(s^2 4^s nm)$<br>$O(d^2 4^d nm)$ | $O(nm)$ | |
| plain | W[1]-c ($l$) | $O(9^k nm)$<br>$O(9^d nm)$ | $O(s^2 4^s nm)$<br>$O(d^2 4^d nm)$ | $O(nm)$ | $O(nm)$ |
| | unlim | cross | nest | chain | plain |

Table 7.3: Parameterized complexity results for problem $\Pi$ with different restrictions

- changed to only look at the root and its children
- time complexity $O(s^2)$

These changes to the tree operations reduce the overall time complexity of the algorithm to $O(s^2 4^s nm)$. □

**Corollary 7.14.** $\Pi(nest, nest)$ parameterized by bandwidth $d$ can be solved in time $O(d^2 4^d nm)$.

**Proof:** Since the arcs cannot share endpoints, nesting depth $s$ is always no more than bandwidth $d$. Two sequences of bandwidth $d$ thus both have nesting depth $\leq d$, so the above algorithm can be run on them in time $\in O(d^2 4^d nm)$. □

These parametric results are summarized in Table 7.3.

For arc-annotated sequences, faster or more space-efficient methods that depend on subdividing the distance table [MP80, Hir75] cannot be applied. The *Four Rus-*

*sians* technique used by Masek and Paterson [MP80], in particular, requires that the values in each submatrix and its final edge vectors be dependent only on the submatrix's initial edge vectors and its corresponding substrings. To correctly match arcs, the submatrix values are also dependent on the possible matches made for initial endpoints of arcs whose final endpoints are in the submatrix. Also, any initial endpoint matches made in that submatrix or in preceding submatrices must be passed on to subsequent submatrices. Thus, the addition of the arc annotation to the input of the longest common subsequence problem produces submatrices that are dependent on more than just the immediately preceding submatrices; it can make a submatrix's values dependent on the location of matches made in any or all of the preceding submatrices. For instances where an annotation with arcs is being aligned against one without arcs, however, the technique may be applicable. In this case, the arcs function as a link of mutual exclusivity, where one endpoint or the other can be matched but not both. Since the algorithm must remember only if the initial endpoint has been matched, not where the match occurs, this extra information is finite for each vector position and could potentially be incorporated into the technique of submatrices.

# Chapter 8

# Algorithm Variations

## 8.1 Using Symbol Weights

The algorithm given in chapter 7 finds the length of the longest common subsequence of two sequences, preserving induced arcs. Each pair of matching symbols, then, is worth 1 to the length sum while a mismatch is worth 0. This simple sum can be replaced by a more complex weighting scheme without compromising the underlying algorithm.

Weights can be assigned to each possible pair of symbols, which is generally used to compute an edit distance or a weighted subsequence length. Each weight would be based in the likelihood or importance of that particular symbol match or mismatch. Any existing weighting scheme $w(x, y)$ can be substituted for the simple one used by algorithm 7.11, provided that the weighting scheme preserves:

$$w(x,y)\begin{cases} = & 0 \quad \text{if } x = \lambda \text{ or } y = \lambda \\ \geq & 0 \quad \text{otherwise} \end{cases}$$

Negative values could be used for the weighting function, but they serve no purpose except to prevent the algorithm from making the corresponding symbol matches.

## 8.2 Layering Sequences with Arcs

Once the algorithm has been adjusted to use weights, those weights can come either from a lookup table or from recursive matching of a lower-level sequence. This recursion is identical to that presented in chapter 6, except that now any or all of the sequences in the layers can include arcs. These arcs must, of course, only link positions within the same substring sequence. They cannot link positions at different levels or from different substrings within the same level.

Using arcs with layered sequences enables an arc structure to be compressed in order to remove redundant arcs. The base pair structures found in RNA molecules contain many arcs that are parallel to each other, forming ladders of linked pairs. Figure 8.1 gives an example of such a ladder structure. Since a small arc cutwidth of each sequence is crucial to the feasibility of the algorithm, these sets of parallel arcs may need to be removed and replaced by single arcs that link substrings, not symbols.

This scheme is limited to matching one entire substring to another; it cannot match one substring to pieces of more than one substring, even if those pieces are adjacent. The structure of the sequences needs to be defined carefully to allow for minor editing, including insertions or deletions of smaller sub-branches. Large scale editing would not be allowed by the substring structure.

At any level, if the sequence of substrings at that level has arc annotation, the algorithm for finding the length of the longest common subsequence while preserving arcs can be used to find its longest common subsequence. If the annotation has $k$

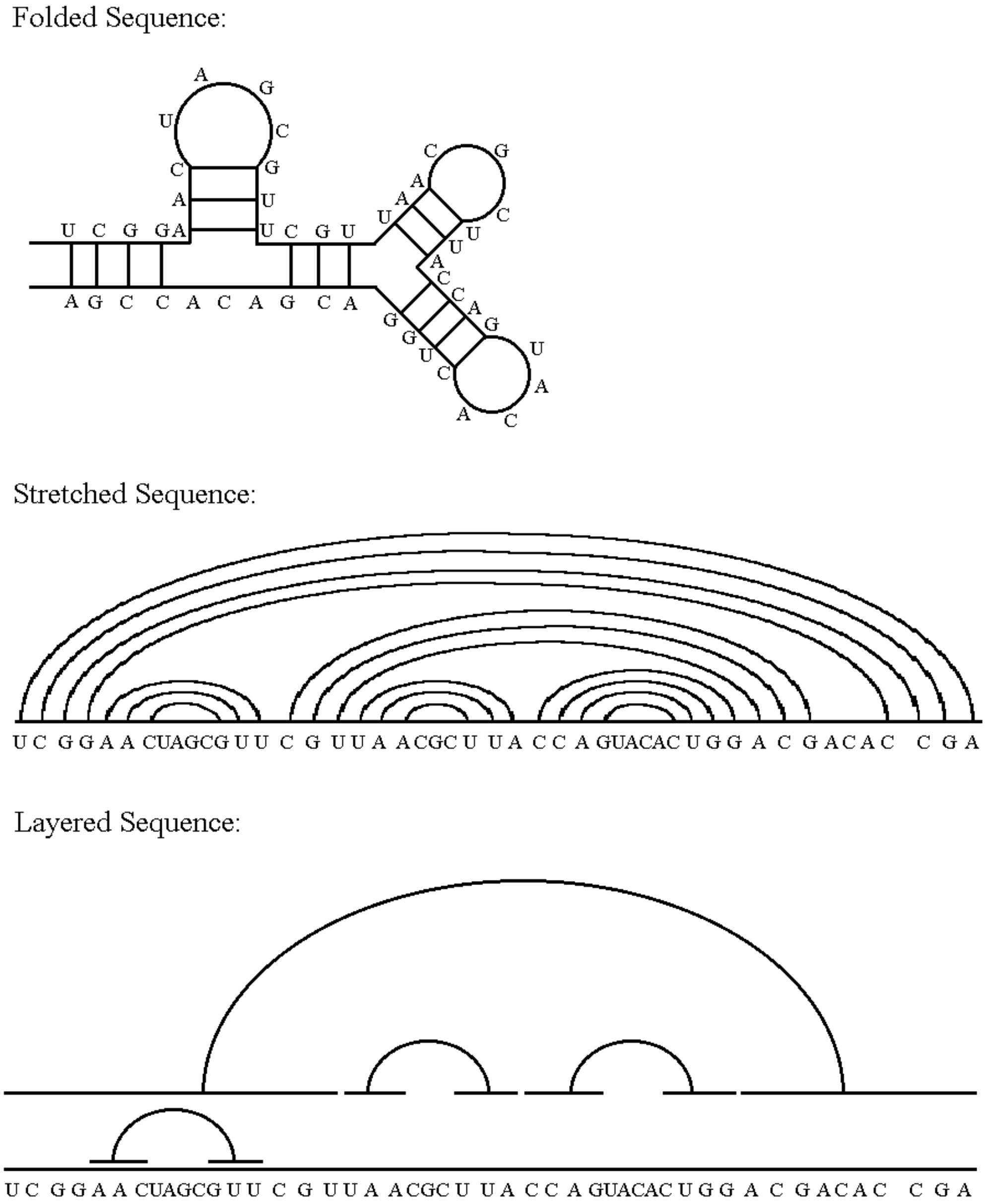


Figure 8.1: Example of Folded RNA sequence represented using Layers

levels of substrings, each with non-crossing arcs with nesting depth $d$, the composite algorithm will run in time $O(k4^d nm)$. If the substring levels were used to compress at least $q$ parallel arcs linking symbols into each single arc linking substrings, the time complexity of the algorithm that compares the original arc-annotated sequences is at least $\Omega(4^{qd} nm)$.

## 8.3 Applying Weights to Arcs

Just as weights can be applied to the matches between pairs of symbols, weights can also be applied to matches between arcs. These weights can be either additive, an amount added directly to the cumulative sum, or multiplicative, a factor that multiplies the contribution of the base pair's match weights to that sum. Using arc weights of either kind requires modifications to the value calculation formula. It also requires that the tree data structure store arc matches from non-maximum paths, since a non-maximum path may turn into a maximum one.

The trees maintained by the algorithm must now include the active arc matches of all valid computation paths, not just those that produce the maximum similarity score. If non-maximal arc assignments are not kept, then some arcs can be eliminated based not on the total value that they would contribute (which includes the arc weight) but based only on the value contributed by their initial endpoints. While eliminating the initial endpoint match would not prevent matching the final endpoints (thus getting the value from that symbol match), it would prevent matching the entire arc and using its arc match weight. A path that does match the initial endpoints, then, could become a maximum path even if it is not a maximum path before the final endpoints are matched.

Since the trees need to include the initial endpoint matches for non-maximum paths, the functions that *merge* and *test* the trees need to be altered so that the matches can be preserved and detected. These functions will be altered as follows:

- *merge*: is applied to a finite list of trees and their corresponding subsequence length values, and returns the merge of those trees. When the trees are merged, they are copied and also simplified from the root down by uniting identical children of the same parent node. Trees that have the same corresponding values are merged as before. However, if one or more of the trees have a corresponding value $x$ that is not maximum, its root has the offset $x - maximum$ attached to it before merging. When trees with offsets are merged, identical children are still united even if their offset is different; the smaller offset is used, and the offset difference is passed down to the children of the larger offset node. This merging is illustrated in figure 8.2.

- *test*: looks for a given arc assignment pair in the tree; returns a non-negative number if it is found, and *false* otherwise. This non-negative number will be the minimum sum of the offset numbers attached to the nodes on a path from the root to the arc assignment pair. If the arc assignment pair occurs more than once, the minimum over all occurrences is used. This number will thus be 0 if the arc assignment pair is found in a maximum computation path, and positive if the arc assignment pair is in a non-maximum computation path.

The alterations to these data structure functions do not affect the worst-case size of the data structures, and thus do not affect the worst-case running time of the algorithm. Having only the maximum computation paths in the tree was not used in the analysis of the original algorithm, and thus all the non-maximum paths that are included in the revised structure were counted in the original analysis as potential maximum paths. While the worst case analysis is not affected, in a typical case

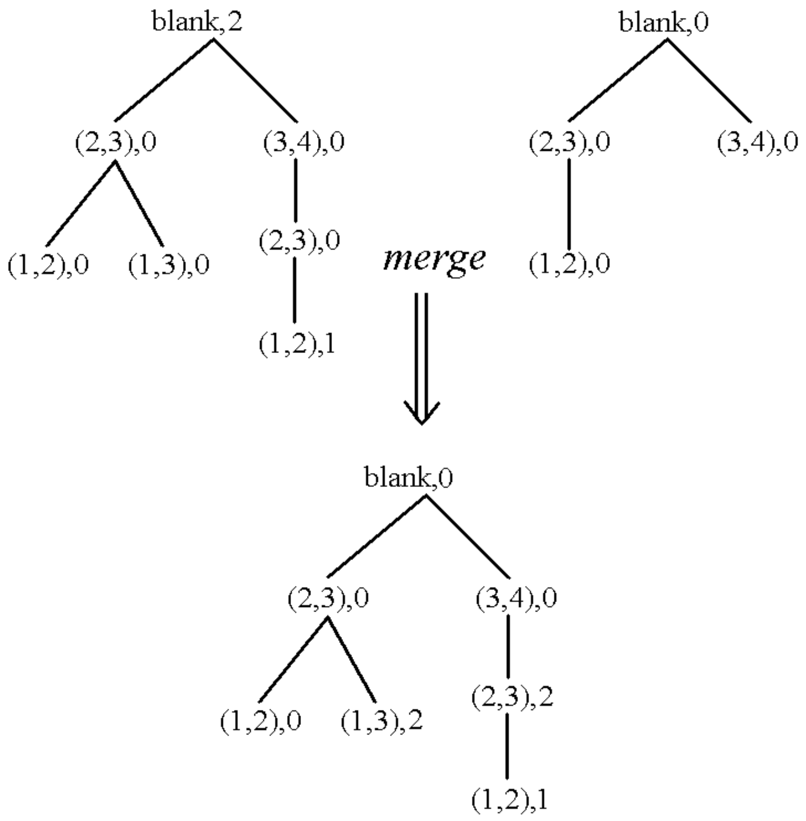


Figure 8.2: Example of merging trees with offset labels

one would expect that the trees would be larger if they include paths that do not correspond to maximum weight subsequences. The revised *test* function is also expected to take longer in a typical case since it must find all occurrences of an arc assignment pair, not just one (similar to *trim* and *prune*).

Since the *test* function has been altered to return an offset value if the arc assignment pair has been found (instead of *true*), the part of the algorithm that uses *true* must also be altered to use that offset. In variation 4, the offset would be subtracted from the formula that calculates the new similarity value for the final endpoint match.

These value calculations would also include the arc weight, which can be either additive or multiplicative. The arc weight can either replace the restriction that requires the arc to be matched if both endpoints are matched, or it can be used in combination with that restriction.

If used in combination with the arc restriction, an additive weight can be any value, even a negative value. If this weight is negative, then a negative value should not be used that is greater in magnitude than any of the $w(x, y)$ weight for symbol matches; otherwise some arc matches will be prevented not by the sequence but only by the weighting scheme. An additive arc weight can be a fixed weight $w_a$, or it can be dependent on the positions of the arc endpoints $i_1, i_2, j_1, j_2$. Such a dependency could be used to increase the weight of the arc if the "stretch" of the two matched arcs is close (so $|(i_2 - i_1) - (j_2 - j_1)| \leq \epsilon$ for some tolerance $\epsilon$).

The use of an additive weight alters the recurrence relation used to calculate the longest common subsequence, but only for variation 4, the case used when a pair of final endpoints is encountered. Should a pair of final endpoints ($i_2$ and $j_2$) be encountered, and the corresponding pair of initial endpoints ($i_1$ and $j_1$) is in the tree, then the value for matching those final endpoints is now calculated by

$$T' = T[i_2 - 1, j_2 - 1] + w(S_1[i_2], S_2[j_2]) + w_a(i_1, i_2, j_1, j_2) - f$$

where $f$ is the offset returned by *test*, and the new table value is calculated as

$$T[i_2, j_2] = \max(T[i_2 - 1, j_2], T[i_2, j_2 - 1], T')$$

from previous table values. The tree is then *pruned* and *merged* with the other trees, using the modified *merge* function.

If a multiplicative arc weight is used in combination with the arc restriction, this factor can be any positive value. If this weight, $w_m$, is greater than 1, it will increase the value contributed by the matched linked pairs. If $w_m$ is less than 1, it will reduce this value. This reduction can only have an effect because the restriction is still in force; the endpoints cannot be matched without including the arcs that link them. Without this restriction, the arcs and the reducing factor would be ignored.

Use of a multiplicative arc weight would alter the calculation of the table value in a manner similar to that required by the use of an additive arc weight, except that the value for matching the final endpoints ($i_2$ and $j_2$) would be calculated by

$$T' = T[i_2 - 1, j_2 - 1] - w(S_1[i_1], S_2[j_1]) + w_m \cdot (w(S_1[i_1], S_2[j_1]) + w(S_1[i_2], S_2[j_2])) - f$$

where $f$ is the offset returned by *test*.

Notice that for this type of weight to not prohibit any arc or endpoint matches (by the weighting scheme rather than by the sequence order or arc mismatch), the result of $w_m \cdot (w(S_1[i_1], S_2[j_1]) + w(S_1[i_2], S_2[j_2]))$ must be at least as large as the amount removed, $w(S_1[i_1], S_2[j_1])$. This property can be achieved by ensuring that

$$w_m \geq \frac{1}{1 + \frac{w(x_2, y_2)}{w(x_1, y_1)}} \qquad \forall x_1, x_2, y_1, y_2 \in \Sigma$$

A reducing factor could be chosen to decrease the effect of matching base pairs of the total score for the sequence comparison. Since RNA and DNA bases are only linked in specific pairs, matching half of a pair makes it more likely that the other half will also be matched, effectively doubling the contributed weight. This phenomenon can give much higher significance to how many base pairs are matched; close matches to parts of the sequence that do not involve linked pairs can be overshadowed and ignored. Using a reducing factor for matched base pairs can dampen their effect on the comparison score without ignoring them entirely.

Without the restriction requiring induced arcs to be matched, a factor reducing the value contributed by matched base pairs cannot be used. The sequences could be aligned normally and subsequently rescored with the reducing factor, but that would only change the final alignment score; it could not alter the alignment itself.

## 8.4 Adding Labels to Arcs

The algorithm given in chapter 7 and the variations above only use one kind of arc; all arcs are considered the same. However, it can be useful to have more than one kind of arc, with arcs only matching if they are the same kind. To do this, the arcs must be labeled. Each arc $(i_1, i_2)$ thus becomes a triple $(i_1, i_2, c)$ as the label $c$ from some set of arc labels $C$ is added to the pair of endpoints.

Adding labels to arcs means that it is no longer sufficient to compare arc endpoints; the arc labels must also be compared. Accordingly, the algorithm needs to be modified in its behaviour when a pair of initial endpoints are encountered (variation 3 of the table calculation). The tables are activated and initialized as before, except for the table that has both initial endpoints. If the two arcs have the same label, the tree is *extended* by adding the arc assignment pair; however, if they do not have the same label then the tree is not *extended.* The value calculation is unchanged.

If arc labels are used, matches and mismatches between arc labels can also be incorporated into the weighting scheme. Allowing arc label mismatches would mean that the change described above to variation 3 (when initial endpoints are matched) would not be done. Instead, variation 4 (when final endpoints are matched) would be modified further. In the event of encountering a pair of final endpoints, the labels of the corresponding arcs are compared, and the result of this comparison can be used to change the value of either additive or multiplicative arc weights, so they can be functions $w_a : C \times C \to \mathbf{Z}$ and $w_m : C \times C \to \mathbf{R}^+$ that have a weight for each possible arc label match and mismatch, in the same way as the original weight function $w(x, y)$ has values for each possible symbol match and mismatch.

If arcs can be labeled, then sequence annotation systems that use arcs to represent bonds can encode and differentiate between different kinds of bonds. If arc labels are incorporated into the layered arc scheme discussed previously, then the arcs linking substrings can represent a different type of link. For example, a simple type of pseudoknot that links pairs from two substrings in order (rather than in reverse, as a fold does) can be represented by a single labeled arc that links the substrings. An example of this representation is given in Figure 8.3.

Theoretically, these weighting schemes that weight different parts of the arc and sequence comparison can be combined. The use in combination of several weighting schemes (for symbols, arcs, arc labels, symbol colours, and substring colours) does increase the experimentation needed to determine specific usable values for the many weights, however.

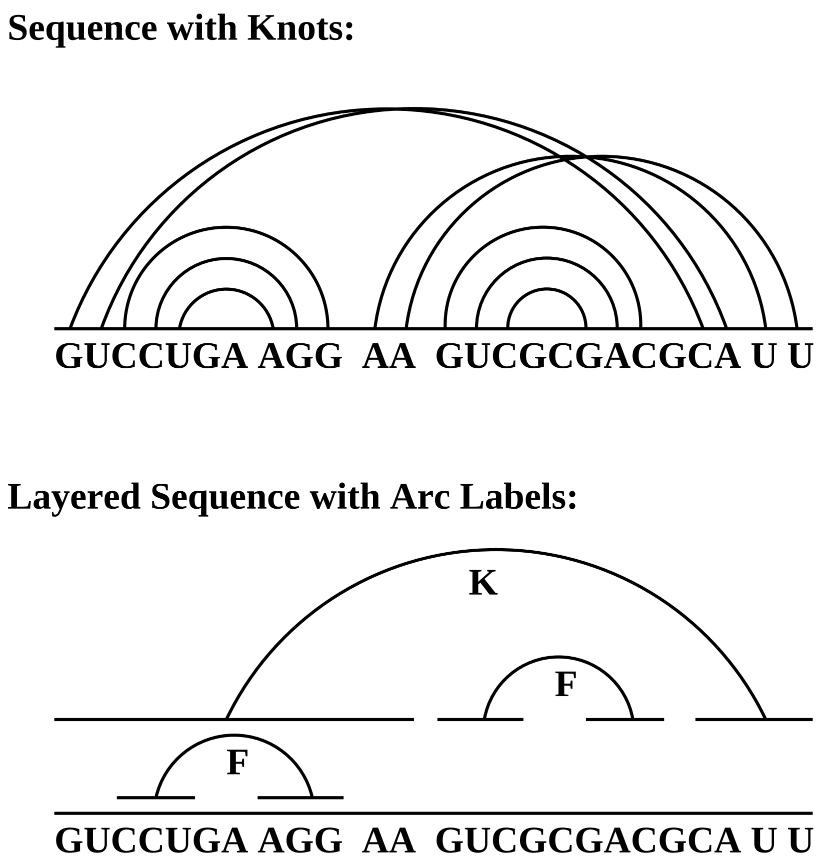


Figure 8.3: Example of RNA Sequence with Knots and Folds represented using Layers

## 8.5 Induced Arc Mismatch Weights

Once the algorithm has been modified to use weights for matching arcs, it can also be altered further, to use weights also when arcs are being mismatched. This refers not to the labels or symbols being mismatched, but to the breaking of the second criterion of the general arc-preserving longest common subsequence problem, that the arcs induced by the mapping are preserved. This criterion can be relaxed to allow for some induced arcs to not be preserved if some appropriate penalty is paid.

To relax this restriction, the algorithm is modified in variation 2 of the table calculation, which is executed when a final arc endpoint is encountered (and is also referred to in variation 4, when a pair of final endpoints is encountered). The first table, where the initial endpoint was not allowed to match, is calculated as before. The second table, which did allow the initial endpoint to match, is now calculated with matching the final endpoint, but also applying the arc mismatch penalty $w_p$ if the arcs do not match. Thus the calculation for that table becomes

$$T[i,j] = \max(T[i-1.j], T[i,j-1], T[i-1,j-1] + w(S_1[i], S_2[j]) - w_p)$$

and the value and tree from that table is merged with those of the other table to find the maximum (and produce the resulting tree). The trees are *trimmed* to remove all assignments that use that arc, as before.

If the penalty $w_p$ is higher than all $w(x,y)$ symbol weights, the altered algorithm will still enforce the original prohibition on induced arc mismatch. If $w_p = 0$, then the altered algorithm solves the same subproblem as the $O(n^2m^2)$ algorithm of Bafna et. al. [BMR96]. For sequences that are very long and have limited cutwidth ($k < \log_9 n$, where $n \geq m$), the altered algorithm described here will have a worst-case time complexity $\in O(n^2m)$. So for sequences with limited cutwidth, this algorithm can exploit that limitation to have a smaller time complexity than the previous best known algorithm for this specific subproblem [BMR96].

As the modifications discussed in this chapter do change some of the detailed pseudocode, the details of these changes are given in the appendix.

# Chapter 9

# Conclusions

## 9.1 Summary of Results

This research defined several annotation schemes using simple combinatorial objects. For each scheme, the problems of verifying, creating, and using the annotation schemes were investigated, producing the following results.

For coloured symbols, restrictions can be verified in linear time, and the classic algorithm for finding the longest common subsequence can be modified to an equivalent algorithm that uses both symbol and colour. The colours can either be another attribute to match, or be applied to one sequence and indicate the importance of matching the symbols at the different positions. This latter case has been applied to encourage sequence matches that closely match protein family motifs.

Coloured substring annotations can be verified according to both correct format and meeting restrictions in linear time. If each coloured substring is restricted to being a member of a regular language corresponding to its colour, the language of correctly annotated sequences is finite state recognizable. Given the sequence, the corresponding annotation is finite state searchable. The longest common subsequence algorithm can use weights to produce an algorithm that will compare two sequences with coloured substring annotations, and produce the longest common subsequence that preserves those substrings and their colours.

If coloured substrings are used in layers to form a tree, the format can still be verified in linear time, and marking according to the parse of a factored regular language is finite state searchable. The corresponding language of validly annotated sequences is finite state recognizable. Converting from the ordered lists of substrings used for storage to a tree or forest organization for the substrings can be done in time linear in the number of substrings and the sequence length. Finding the longest common subsequence that preserves or weights the colours and substrings at each level of the annotation is done through a recursive algorithm that takes $O(kn^2)$ time to compare two sequences of length $n$ with up to $k$ levels of substrings.

The format for arc annotation can be verified in linear time, and converted from a storage list to a superimposed format in linear time. Restrictions on the pairs of symbols that arcs can link, their nesting depth or cutwidth, and whether or not they can cross or share endpoints can also be verified in linear time. Finding the longest common subsequence of two sequences that preserve their arc annotations can be done in time $O(9^k nm)$ for sequences with cutwidth $k$ and in time $O(s^2 4^s nm)$ for sequences with nesting depth $s$. Both results use an algorithm that tests the different possibilities of matching arc endpoints, and keeps track of active endpoint matches in a tree of table computation paths. The results are also converted to use weighting functions for symbols and arc match and mismatch, thus producing an extended solution to a weighted match with arcs problem with polynomial time complexity $\in O(n^2m^2)$ [BMR96]. For long sequences with relatively small nesting depth (eg. $s << \max(\log n, \log m)$), the fixed-parameter algorithm discussed here is more efficient. If the arcs are allowed to cross and the problem is parameterized instead by target subsequence length $l$, finding the longest common subsequence of two arc-annotated sequences that preserves arcs is $W[1]$-complete and NP-complete. Even if one of the sequences is known not to have any arcs as its annotation, the resulting problem is still NP-complete.

## 9.2 Avenues for Future Work

The obvious next step is to apply these annotation schemes to existing databases of sequences with additional information. Symbol colouring has already been applied successfully to protein sequences with family motifs. The flexible annotation schemes discussed can be suitable for many applications of sequence comparison. Arcs are directly applicable to molecular sequences with bonds between pairs of bases in the sequence. Substring and layered substring annotation can be applied to any sequences that have features composed of substrings. Language or text databases, as well as molecular sequence databases, can be annotated with these features and compared using them.

The comparison problems for sequences with arc annotation are defined to preserve the arcs, and are also extended to use a variety of weights for symbol match, symbol mismatch, arc match, and arc mismatch. If the weighting scheme is restricted, and does not include some of these possible weights or only includes a small range, the complexity of the resulting problem can decrease. These altered problems are more similar to that looked at by Bafna, Muthukrishnan, and Ravi [BMR96], and should be investigated using both classical and parameterized complexity. Any algorithm that uses a weighting scheme needs to be looked at and experimented with for a specific application to derive an appropriate set of weights.

The problem of finding the longest common subsequence preserving arcs for nesting arcs is not known to be NP-hard, but also has no yet known polynomial time algorithm. Work can be done in further investigating the complexity of this important problem.

Mixed annotation using layers of substrings connected by arcs are presented here as a combination of the layered substring and arc annotations. Further investigation and application of this feature and structure definition tool looks promising. Since this mixed annotation can condense arc annotations of high nesting depth and cutwidth, the two alternative forms of annotation can be compared. Methods for automatically converting between them can also be developed.

Other automatic annotation conversion methods can be investigated to create annotations in the correct form from the result of the extensive existing feature search and structure search and prediction programs.

# Appendix A

## A.1 Details for Algorithm 7.11

The algorithm described in chapter 7 that solves $\Pi(cross, cross)$, finding the longest arc-preserving common subsequence, for sequences with bounded arc cutwidth $k$, has the following details.

*Step 1.* Partitioning the arcs into $k$ chains, numbered $\{0, \ldots, k-1\}$ :

For all values $i$ and $j$ such that $1 \leq i \leq n$ and $1 \leq j \leq m$, compute

$$x_1[i] = \begin{cases} 1 & \text{if } \exists i' \text{ such that } (i, i') \in P_1 \\ -1 & \text{if } \exists i' \text{ such that } (i', i) \in P_1 \\ 0 & \text{otherwise } (i \text{ is not an arc endpoint}) \end{cases}$$

$$x_2[j] = \begin{cases} 1 & \text{if } \exists j' \text{ such that } (j, j') \in P_2 \\ -1 & \text{if } \exists j' \text{ such that } (j', j) \in P_1 \\ 0 & \text{otherwise } (j \text{ is not an arc endpoint}) \end{cases}$$

Since the maximum number of active arcs is $k$, have $k$ arc numbers, $\{0, 1, \ldots, k-1\}$. Process each string from left to right to compute $n_1[i]$ for all $i$ and $n_2[j]$ for all $j$, as follows.

If $x_1[i] = 1$, let $n_1[i] =$ lowest unused arc number.

mark this number as used.

If $x_1[i] = -1$, let $n_1[i] = n_1[i']$ where $(i', i) \in P_1$.

mark this number $n_1[i']$ as unused.

If $x_1[i] = 0$, let $n_1[i] = 0$.

Compute $n_2[j]$ from $x_2[j]$ and $P_2$ in the same way.

*Step 2.* Make copies of the sequences with certain initial endpoints removed, creating the sets of sequences $\mathcal{S}_1$ and $\mathcal{S}_2$ :

Compute the cumulative marker for each position by

$$cum_1[i] = \sum_{i'=1}^{i} 2^{n_1[i']} \cdot x_1[i']$$

$$cum_2[i] = \sum_{j'=1}^{j} 2^{n_2[j']} \cdot x_2[j']$$

Note that $\forall i, cum_1[i] \leq 2^k$ and $\forall j, cum_2[j] \leq 2^k$.

Let $S_1^{(h)}$ be the result of processing $S_1$ and replacing all symbols $S_1[i]$ with the null symbol $\lambda$ where $x_1[i] = 1$ and $h \bmod 2^{n_1[i]+1} \geq 2^{n_1[i]}$. Similarly, let $S_2^{(h)}$ be the result of processing $S_2$ and replacing all symbols $S_2[j]$ with the null symbol $\lambda$ where $x_2[j] = 1$ and $h \bmod 2^{n_2[j]+1} \geq 2^{n_2[j]}$. Thus $S_1^{(h)}$ and $S_2^{(h)}$, for all $h$ such that $0 \leq h \leq 2^k - 1$, have null symbols at all the starting endpoints of arcs whose arc number corresponds to a 1 in the binary representation of $h$.

*Step 3.* Creating the tables of values (one for each sequence pair) with the trees of initial matched endpoints :

Let $h_1$ and $h_2$ be two numbers between 0 and $2^k - 1$. The corresponding two strings $S_1^{(h_1)}$ and $S_2^{(h_2)}$ define a two-dimensional table, $T^{(h1,h2)}$, with value entries

$T^{(h1,h2)}[i,j]$, $\forall 1 \leq i \leq n$ and $\forall 1 \leq j \leq m$. The numbers $h_1$ and $h_2$ can be viewed as bit strings of length $k$ that have a 1 at those positions corresponding to active arc numbers.

*Step 4.* Calculating the values in the tables, and manipulating the trees :

These tables can be used to compute the length of the longest common arc-preserving subsequence in a manner similar to the established algorithm without arcs. The multiple tables include all possible combinations of deleted starting endpoints of arcs (by arc number), and are used to search through those possibilities.

To enable matched final endpoints to be aligned with matched starting endpoints, the algorithm must keep track of not just the length of the longest common subsequence for each pair of prefixes, but also all combinations of starting endpoint matches that lie on a path that produces this maximum value. So each table position has both a value entry $T^{(h_1,h_2)}[i,j]$ and a tree entry $M^{(h_1,h_2)}[i,j]$ where each path from leaf to root in $M^{(h_1,h_2)}[i,j]$ is a sequence of starting endpoint matches along a computation path that produces the value entry $T^{(h_1,h_2)}[i,j]$.

The tree data structure has the following operations. Note that since the trees can be edited at any node within the tree, the trees must be copied to produce different copies for different positions in the tables. Thus all operations have time complexity $\in O(|M^{(h_1,h_2)}[i,j]|)$, the size of the tree at that position, unless otherwise noted.

- $extend(M^{(h_1,h_2)}[i-1,j-1],(i,j))$ : add $(i,j)$ as new root of tree with $M^{(h_1,h_2)}[i-1,j-1]$ as its child, returning the resulting tree
  - used if two starting endpoints are matched
  - time complexity $\in O(1)$ since it does not copy the tree

- *merge* : is applied to a finite list of trees $\langle M^{(h_1,h_2)}[i,j]\rangle$ with corresponding values $\langle T^{(h_1,h_2)}[i,j]\rangle$ and returns the merge of those trees based on the values, as follows.
  - compares the values and finds their maxima
  - if only one maximum, *merge* returns a copy of the corresponding tree
  - if more than one maxima, copies of the corresponding trees are merged by joining the copies as children of an empty root, then simplifying the tree from the root down by uniting identical children of the same parent
  - note that the trees and values may not always be taken exactly from other table positions; they can be modified first using other functions
- $test(M^{(h_1,h_2)}[i,j],(i',j'))$ : returns $true$ if $(i',j')$ in $M^{(h_1,h_2)}[i,j]$, $false$ otherwise
  - used if a pair of final endpoints are being attempted to match, to determine if the corresponding pair of starting endpoints were matched
- $prune(M^{(h_1,h_2)}[i,j],(i',j'))$ : remove all paths that do not contain $(i',j')$, and remove $(i',j')$, returning the resulting tree
  - used if a pair of final endpoints are being matched, to remove paths where they cannot match
- $trim(M^{(h_1,h_2)}[i,j], flag, k')$ : if $flag = 0$, remove all entries $(i',j')$ where $n_1[i'] = k'$; if $flag = 1$, remove all entries $(i',j')$ where $n_2[j'] = k'$. Simplify and return the resulting tree
  - used to remove no longer active arcs from tree

– time complexity $O(|M^{(h_1,h_2)}[i,j]|)$

The algorithm initializes the tables by considering $T^{(h_1,h_2)}[i,j] = 0$ and $M^{(h_1,h_2)}[i,j]$ to be a null tree if either $i = 0$ or $j = 0$. Let $H = \{(h_1, h_2)\}$ represent the currently active set of tables. The algorithm processes each active table row by row, from left to right. Start with $H = \{(0,0)\}$.

In general, the algorithm follows the execution of the fundamental LCS algorithm, but it has many parallel tables at each position instead of only one. The basic longest common subsequence formula, adjusted for removed endpoints,

$$T[i,j] = \max(T[i-1,j], T[i,j-1], T[i-1,j-1] + w(S_1[i], S_2[j]))$$

where

$$w(x,y) = \begin{cases} 1 & \text{if } x = y \text{ or } x = y = \lambda \\ 0 & \text{otherwise} \end{cases}$$

is used. At each step, the $T$ values are computed from maxima of $T$ values, and the $M$ trees are computed by merging the trees that correspond to those maxima. The parallel tables and trees are managed as follows:

- when a starting endpoint is encountered, split the tables
- when a final endpoint is encountered, join the tables and trim the trees
- if both symbols are starting endpoints and the symbols match, add that arc assignment to the tree
- if both symbols are final endpoints and the symbols match, look for paths in the tree that include that arc assignment; if such paths exist, we can match the endpoints, so prune all other branches and remove the arc assignment

**Detailed Pseudocode of the Algorithm:**

For each $j$ from 1 to $m$

For each $i$ from 1 to $n$

There are nine different cases, based on the $x_1[i]$ and $x_2[j]$ values.

$x_1[i] = 0 = x_2[j]$: neither symbol is an endpoint.

$T^{(h_1,h_2)}[i,j] = max$ of $T^{(h_1,h_2)}[i-1,j]$, $T^{(h_1,h_2)}[i,j-1]$,

and $T^{(h_1,h_2)}[i-1,j-1] + w(S_1^{(h_1)}[i], S_2^{(h_2)}[j])$

$M^{(h_1,h_2)}[i,j] = merge$ of $M^{(h_1,h_2)}[i-1,j-1]$,

$M^{(h_1,h_2)}[i-1,j]$, and $M^{(h_1,h_2)}[i,j-1]$

$x_1[i] = 1$ and $x_2[j] = 0$: one starting endpoint (on $S_1$).

$\forall(h_1,h_2) \in H$, activate tables $(h_1 + 2^{n_1[i]}, h_2)$ and compute

$T^{(h_1+2^{n_1[i]},h_2)}[i,j] = max$ of $T^{(h_1,h_2)}[i-1,j]$ and $T^{(h_1+2^{n_1[i]},h_2)}[i,j-1]$

$M^{(h_1+2^{n_1[i]},h_2)}[i,j] = merge$ of $M^{(h_1,h_2)}[i-1,j]$ and $M^{(h_1+2^{n_1[i]},h_2)}[i,j-1]$

For the previously active tables $(h_1,h_2)$, compute as if $x_1[i] = 0$.

$x_1[i] = 0$ and $x_2[j] = 1$: one starting endpoint (on $S_2$).

if $i = 1$, $\forall(h_1,h_2) \in H$, activate tables $(h_1, h_2 + 2^{n_2[j]})$

and copy the previous row:

$\forall l$, $1 \leq l \leq n$,

$T^{(h_1,h_2+2^{n_2[j]})}[l,j-1] = T^{(h_1,h_2)}[l,j-1]$

$M^{(h_1,h_2+2^{n_2[j]})}[l,j-1] = M^{(h_1,h_2)}[l,j-1]$

Add $(h_1, h_2 + 2^{n_2[j]})$ to $H$ and for all tables in $H$,

compute as if $x_2[j] = 0$.

$x_1[i] = 1 = x_2[j]$: both symbols are starting endpoints.

if $i = 1$, $\forall (h_1, h_2) \in H$, activate tables $(h_1, h_2 + 2^{n_2[j]})$

and copy the previous row:

$\forall l$, $1 \leq l \leq n$,

$T^{(h_1, h_2 + 2^{n_2[j]})}[l, j-1] = T^{(h_1,h_2)}[l, j-1]$

$M^{(h_1, h_2 + 2^{n_2[j]})}[l, j-1] = M^{(h_1,h_2)}[l, j-1]$

Add $(h_1, h_2 + 2^{n_2[j]})$ to $H$ and for all tables $(h_1, h_2) \in H$,

activate tables $(h_1 + 2^{n_1[i]}, h_2)$, and compute as if $x_2[j] = 0$.

For the previously active tables $(h_1, h_2)$, similar to $x_1[i] = 0$:

if either $S_1^{(h_1)}[i]$ or $S_2^{(h_2)}[j]$ is $\lambda$,

$T^{(h_1,h_2)}[i,j] = max$ of $T^{(h_1,h_2)}[i-1,j]$ and $T^{(h_1,h_2)}[i,j-1]$

$M^{(h_1,h_2)}[i,j] = merge$ of $M^{(h_1,h_2)}[i-1,j]$ and $M^{(h_1,h_2)}[i,j-1]$

if $S_1^{(h_1)}[i] \neq \lambda \neq S_2^{(h_2)}[j]$,

add the new matched starting endpoints to the tree, computing

$T^{(h_1,h_2)}[i,j] = max$ of $T^{(h_1,h_2)}[i-1,j]$, $T^{(h_1,h_2)}[i,j-1]$,

and $T^{(h_1,h_2)}[i-1,j-1] + w(S_1^{(h_1)}[i], S_2^{(h_2)}[j])$

$M^{(h_1,h_2)}[i,j] = merge$ of $M^{(h_1,h_2)}[i-1,j]$ and $M^{(h_1,h_2)}[i,j-1]$

with $extend(M^{(h_1,h_2)}[i-1,j-1], (i,j))$

$x_1[i] = -1$ and $x_2[j] = 0$: one final endpoint (on $S_1$).

Find all $(h_1', h_2) \in H$ where $h_1' \bmod 2^{n_1[i]+1} \geq 2^{n_1[i]}$. Let $H' = \{(h_1', h_2)\}$.

for each such $h_1'$, compute $T^{(h_1',h_2)}[i,j]$ and $M^{(h_1',h_2)}[i,j]$ as if $x_1[i] = 0$.

To compute $T^{(h_1' - 2^{n_1[i]}, h_2)}[i,j]$ and $M^{(h_1' - 2^{n_1[i]}, h_2)}[i,j]$ :

$T^{(h_1' - 2^{n_1[i]}, h_2)}[i,j] = max$ of $T^{(h_1',h_2)}[i,j]$,

$T^{(h_1' - 2^{n_1[i]}, h_2)}[i-1,j]$, and $T^{(h_1' - 2^{n_1[i]}, h_2)}[i,j-1]$

$M^{(h_1'-2^{n_1[i]},h_2)}[i,j]$ = apply $trim$ to $merge$ of $M^{(h_1',h_2)}[i,j]$,

$M^{(h_1'-2^{n_1[i]},h_2)}[i-1,j]$, and $M^{(h_1'-2^{n_1[i]},h_2)}[i,j-1]$

with $flag = 0$ and $k' = n_1[i]$ as the arguments of $trim$

(to remove all assignments of $n_1[i]$).

Deactivate tables $H'$ and let active set $H$ be $H - H'$.

$x_1[i] = 0$ and $x_2[j] = -1$: one final endpoint (on $S_2$).

Find all $(h_1, h_2') \in H$ where $h_2' \bmod 2^{n_2[j]+1} \geq 2^{n_2[j]}$. Let $H' = \{(h_1, h_2')\}$.

for each such $h_2'$, compute $T^{(h_1,h_2')}[i,j]$ and $M^{(h_1,h_2')}[i,j]$ as if $x_2[j] = 0$.

To compute $T^{(h_1,h_2'-2^{n_2[j]})}[i,j]$ and $M^{(h_1,h_2'-2^{n_2[j]})}[i,j]$ :

$T^{(h_1,h_2'-2^{n_2[j]})}[i,j] = max$ of $T^{(h_1,h_2')}[i,j]$,

$T^{(h_1,h_2'-2^{n_2[j]})}[i-1,j]$, and $T^{(h_1,h_2'-2^{n_2[j]})}[i,j-1]$

$M^{(h_1'-2^{n_1[i]},h_2)}[i,j]$ =apply $trim$ to the $merge$ of $M^{(h_1,h_2')}[i,j]$,

$M^{(h_1,h_2'-2^{n_2[j]})}[i-1,j]$, and $M^{(h_1,h_2'-2^{n_2[j]})}[i,j-1]$

with $flag = 1$ and $k' = n_2[j]$ as the arguments of $trim$

(to remove all assignments of $n_2[j]$).

If $i = n$, deactivate tables $H'$ and let active set $H$ be $H - H'$.

$x_1[i] = -1$ and $x_2[j] = 1$: one starting and one final endpoint.

if $i = 1$, $\forall (h_1, h_2) \in H$, activate tables $(h_1, h_2 + 2^{n_2[j]})$

and copy the previous row:

$\forall l$, $1 \leq l \leq n$,

$T^{(h_1,h_2+2^{n_2[j]})}[l,j-1] = T^{(h_1,h_2)}[l,j-1]$

$M^{(h_1,h_2+2^{n_2[j]})}[l,j-1] = M^{(h_1,h_2)}[l,j-1]$

Add $(h_1, h_2 + 2^{n_2[j]})$ to $H$ and for all tables in $H$,

follow the case where $x_1[i] = -1$ and $x_2[j] = 0$ to compute the corresponding $T$ values and $M$ trees,

and deactivate the tables $H' = (h'_1, h_2) \subset H$ where

$h'_1 \bmod 2^{n_1[i]+1} \geq 2^{n_1[i]}$, setting $H = H - H'$.

$x_1[i] = 1$ and $x_2[j] = -1$: one final and one starting endpoint.

$\forall (h_1, h_2) \in H$, activate tables $(h_1 + 2^{n_1[i]}, h_2)$

$T^{(h_1+2^{n_1[i]},h_2)}[i,j] = max$ of $T^{(h_1,h_2)}[i-1,j]$ and $T^{(h_1+2^{n_1[i]},h_2)}[i,j-1]$

$M^{(h_1+2^{n_1[i]},h_2)}[i,j] =$ apply *trim* to the *merge* of $M^{(h_1,h_2)}[i-1,j]$ and

$M^{(h_1+2^{n_1[i]},h_2)}[i,j-1]$

with $flag = 1$ and $k' = n_2[j]$ as the arguments of *trim*

(to remove all assigments of $n_2[j]$).

For the previously active tables $(h_1, h_2)$, follow the case

where $x_1[i] = 0$ and $x_2[j] = -1$ to compute the corresponding

$T$ values and $M$ trees.

Add activated tables $(h_1 + 2^{n_1[i]}, h_2)$ to $H$, and if $i = n$,

deactivate the tables $H' = (h_1, h'_2) \subset H$ where

$h'_2 \bmod 2^{n_2[j]+1} \geq 2^{n_2[j]}$, setting $H = H - H'$.

$x_1[i] = 1 = x_2[j]$: both symbols are final endpoints.

$\forall (h_1, h_2) \in H$, find $H_1 = \{h_1 \mid h_1 \bmod 2^{n_1[i]+1} \geq 2^{n_1[i]}\}$

and $H_2 = \{h_2 \mid h_2 \bmod 2^{n_2[j]+1} \geq 2^{n_2[j]}\}$.

For each $(h_1, h_2) \in H$ where $h_1 \in H_1$ and $h_2 \in H_2$,

follow the case where $x_1[i] = 0$ and $x_2[j] = 0$.

For each $(h_1, h_2) \in H$ where $h_1 \in H_1$ and $h_2 \notin H_2$,

follow the case where $x_1[i] = 0$ and $x_2[j] = -1$.

For each $(h_1, h_2) \in H$ where $h_1 \notin H_1$ and $h_2 \in H_2$, follow the case where $x_1[i] = -1$ and $x_2[j] = 0$.

For each $(h_1, h_2) \in H$ where $h_1 \notin H_1$ and $h_2 \notin H_2$,

we need to merge results from four different tables.

Compute preliminary value $T'$ and tree $M'$ by

$T' = max$ of $T^{(h_1,h_2)}[i-1,j]$, $T^{(h_1,h_2)}[i,j-1]$,

$T^{(h_1+2^{n_1[i]},h_2)}[i,j]$,$T^{(h_1,h_2+2^{n_2[j]})}[i,j]$,

and $T^{(h_1+2^{n_1[i]},h_2+2^{n_2[j]})}[i,j]$)

$M' = merge$ the corresponding trees

$M^{(h_1,h_2)}[i-1,j]$, $M^{(h_1,h_2)}[i,j-1]$), $M^{(h_1+2^{n_1[i]},h_2)}[i,j]$,

$M^{(h_1,h_2+2^{n_2[j]})}[i,j]$, and $M^{(h_1+2^{n_1[i]},h_2+2^{n_2[j]})}[i,j]$

and apply $trim$ twice:

once with $flag = 0$ and $k' = n_1[i]$ to remove $n_1[i]$

and once with $flag = 1$ and $k' = n_2[j]$ to remove $n_2[j]$.

To match $S_1^{(h_1)}[i]$ with $S_2^{(h_2)}[j]$, find $i'$ and $j'$ where

$(i', i) \in P_1$ and $(j', j) \in P_2$.

if $test(M^{(h_1,h_2)}[i-1, j-1], (i', j'))$,

$T^{(h_1,h_2)}[i,j] = max$ of $T'$ and

$T^{(h_1,h_2)}[i-1,j-1] + w(S_1^{(h_1)}[i], S_2^{(h_2)}[j])$

$M^{(h_1,h_2)}[i,j] = merge$ of $M'$ and the result of

$prune(M^{(h_1,h_2)}[i-1,j-1],(i',j'))$

otherwise $T^{(h_1,h_2)}[i,j] = T'$ and $M^{(h_1,h_2)}[i,j] = M'$.

Deactivate tables $(h_1, h_2) \in H$ where $h_1 \in H_1$.

If $i = n$, also deactivate tables $(h_1, h_2) \in H$ where $h_2 \in H_2$.

Update $H$ to remove deactivated tables.

After the table computation, the decision algorithm returns *true* if and only if the length of the longest common arc-preserving subsequence, stored in $T^{(0,0)}[n,m]$, is at least $l$, and returns *false* otherwise.

## A.2 Details for Modified Algorithm

Chapter 8 discusses several different variations that can be done to algorithm 7.11 so that it can use weights for arcs, labels for arcs, and arc mismatch penalties. These variations produce changes in the details of the algorithm for some of the cases.

The preprocessing of the sequences to set up the sequence families $\mathcal{S}_1$ and $\mathcal{S}_2$ and the $4^k$ different tables is unchanged, as is the tree data structure and its operations. Any symbol weighting scheme $w : \Sigma \times \Sigma \rightarrow \mathbf{N}$ can be used. A multiplicative arc weight with arc labels $w_m : C \times C \rightarrow \mathbf{R}^+$ is incorporated, along with the symbol weight function $w$, the induced arc penalty $w_p$, and the offset $f$ resulting from *test*, into a composite arc weight function:

$$w_c : \{1, \ldots, n\} \times \{1, \ldots, m\} \rightarrow \mathbf{R}$$

defined as:

if $x_1[i] \neq x_2[j]$,

$$w_c(i, j) = -w_p + w(S_1[i], S_2[j])$$

if $x_1[i] = x_2[j] = -1$ and where $(i', i, c_1) \in P_1$, $(j', j, c_2) \in P_2$, and $test(M^{(h_1,h_2)}[i-1, j-1], (i', j')) = \mathit{false}$,

$$w_c(i, j) = -2w_p + w(S_1[i], S_2[j])$$

if $x_1[i] = x_2[j] = -1$ and where $(i', i, c_1) \in P_1$, $(j', j, c_2) \in P_2$, and $test(M^{(h_1,h_2)}[i-1, j-1], (i', j')) = f \neq \mathit{false}$,

$$w_c(i,j) = w_m(c_1,c_2) \cdot (w(S_1[i'], S_2[j']) + w(S_1[i], S_2[j])) - f$$

Note that this definition of $w_c$ uses $x_1$ and $x_2$, the arc endpoint indicators, and thus must use altered $x_1$ and $x_2$ when specified in the algorithm. Note also that labeled arcs are used, so $P_1 \subset \{1,\ldots,n\} \times \{1,\ldots,n\} \times C$ and $P_2 \subset \{1,\ldots,m\} \times \{1,\ldots,m\} \times C$.

Since arc weights are used, the altered *merge* and *test* functions described in section 8.3 must be used instead of the originals. These altered functions keep non-maximum paths and determine the minimum offset (from the maximum) of an arc assignment pair.

The tables entries are computed in the same order as before, with the following changes to the cases.

**Modified Detailed Pseudocode**:

$x_1[i] = 0 = x_2[j]$: neither symbol is an endpoint.

no change

$x_1[i] = 1$ and $x_2[j] = 0$: one starting endpoint (on $S_1$).

no change

$x_1[i] = 0$ and $x_2[j] = 1$: one starting endpoint (on $S_2$).

no change

$x_1[i] = 1 = x_2[j]$: both symbols are starting endpoints.

no change

$x_1[i] = -1$ and $x_2[j] = 0$: one final endpoint (on $S_1$).

Find all $(h_1', h_2) \in H$ where $h_1' \bmod 2^{n_1[i]+1} \geq 2^{n_1[i]}$. Let $H' = \{(h_1', h_2)\}$.

for each such $h_1'$, compute $T^{(h_1',h_2)}[i,j]$ and $M^{(h_1',h_2)}[i,j]$ as if $x_1[i] = 0$.

To compute $T^{(h_1'-2^{n_1[i]},h_2)}[i,j]$ and $M^{(h_1'-2^{n_1[i]},h_2)}[i,j]$ :

$T^{(h_1'-2^{n_1[i]},h_2)}[i,j] = max$ of $T^{(h_1',h_2)}[i,j]$,

$T^{(h_1'-2^{n_1[i]},h_2)}[i-1,j]$, $T^{(h_1'-2^{n_1[i]},h_2)}[i,j-1]$,

and $T^{(h_1'-2^{n_1[i]},h_2)}[i-1,j-1])) + w_c(i,j)$

$M^{(h_1'-2^{n_1[i]},h_2)}[i,j] =$ apply *trim* to *merge* of $M^{(h_1',h_2)}[i,j]$,

$M^{(h_1'-2^{n_1[i]},h_2)}[i-1,j]$, $M^{(h_1'-2^{n_1[i]},h_2)}[i,j-1]$,

and $M^{(h_1'-2^{n_1[i]},h_2)}[i-1,j-1]$

with $flag = 0$ and $k' = n_1[i]$ as the arguments of *trim*

(to remove all assignments of $n_1[i]$).

Deactivate tables $H'$ and let active set $H$ be $H - H'$.

$x_1[i] = 0$ and $x_2[j] = -1$: one final endpoint (on $S_2$).

Find all $(h_1, h_2') \in H$ where $h_2' \bmod 2^{n_2[j]+1} \geq 2^{n_2[j]}$. Let $H' = \{(h_1, h_2')\}$.

for each such $h_2'$, compute $T^{(h_1,h_2')}[i,j]$ and $M^{(h_1,h_2')}[i,j]$ as if $x_2[j] = 0$.

To compute $T^{(h_1,h_2'-2^{n_2[j]})}[i,j]$ and $M^{(h_1,h_2'-2^{n_2[j]})}[i,j]$ :

$T^{(h_1,h_2'-2^{n_2[j]})}[i,j] = max$ of $T^{(h_1,h_2')}[i,j]$,

$T^{(h_1,h_2'-2^{n_2[j]})}[i-1,j]$, $T^{(h_1,h_2'-2^{n_2[j]})}[i,j-1]$,

and $T^{(h_1,h_2'-2^{n_2[j]})}[i-1,j-1] + w_c(i,j)$

$M^{(h_1'-2^{n_1[i]},h_2)}[i,j] =$apply *trim* to the *merge* of $M^{(h_1,h_2')}[i,j]$,

$M^{(h_1,h_2'-2^{n_2[j]})}[i-1,j]$, $M^{(h_1,h_2'-2^{n_2[j]})}[i,j-1]$,

and $M^{(h_1,h_2'-2^{n_2[j]})}[i-1,j-1]$

with $flag = 1$ and $k' = n_2[j]$ as the arguments of $trim$

(to remove all assignments of $n_2[j]$).

If $i = n$, deactivate tables $H'$ and let active set $H$ be $H - H'$.

$x_1[i] = -1$ and $x_2[j] = 1$: one starting and one final endpoint.

no change (note, however, that it references a changed case)

$x_1[i] = 1$ and $x_2[j] = -1$: one final and one starting endpoint.

no change (note, however, that it references a changed case)

$x_1[i] = 1 = x_2[j]$: both symbols are final endpoints.

$\forall (h_1, h_2) \in H$, find $H_1 = \{h_1 \mid h_1 \bmod 2^{n_1[i]+1} \geq 2^{n_1[i]}\}$

and $H_2 = \{h_2 \mid h_2 \bmod 2^{n_2[j]+1} \geq 2^{n_2[j]}\}$.

For each $(h_1, h_2) \in H$ where $h_1 \in H_1$ and $h_2 \in H_2$,

follow the case where $x_1[i] = 0$ and $x_2[j] = 0$.

For each $(h_1, h_2) \in H$ where $h_1 \in H_1$ and $h_2 \notin H_2$,

follow the case where $x_1[i] = 0$ and $x_2[j] = -1$.

For each $(h_1, h_2) \in H$ where $h_1 \notin H_1$ and $h_2 \in H_2$,

follow the case where $x_1[i] = -1$ and $x_2[j] = 0$.

For each $(h_1, h_2) \in H$ where $h_1 \notin H_1$ and $h_2 \notin H_2$,

we need to merge results from four different tables.

Compute preliminary value $T'$ and tree $M'$ by

$T' = max$ of $T^{(h_1,h_2)}[i-1, j]$, $T^{(h_1,h_2)}[i, j-1]$,

$T^{(h_1+2^{n_1[i]},h_2)}[i, j]$,$T^{(h_1,h_2+2^{n_2[j]})}[i, j]$,

and $T^{(h_1+2^{n_1[i]},h_2+2^{n_2[j]})}[i, j]$

$M' = merge$ the corresponding trees

$M^{(h_1,h_2)}[i-1,j]$, $M^{(h_1,h_2)}[i,j-1])$, $M^{(h_1+2^{n_1[i]},h_2)}[i,j]$,

$M^{(h_1,h_2+2^{n_2[j]})}[i,j]$, and $M^{(h_1+2^{n_1[i]},h_2+2^{n_2[j]})}[i,j]$

and apply *trim* twice:

once with $flag = 0$ and $k' = n_1[i]$ to remove $n_1[i]$

and once with $flag = 1$ and $k' = n_2[j]$ to remove $n_2[j]$.

To match $S_1^{(h_1)}[i]$ with $S_2^{(h_2)}[j]$, find $i'$ and $j'$ where

$(i',i) \in P_1$ and $(j',j) \in P_2$.

$T^{(h_1,h_2)}[i,j] = max$ of $T'$ and $T^{(h_1,h_2)}[i-1,j-1] + w_c(i,j)$

$M^{(h_1,h_2)}[i,j] = merge$ of $M'$ and the result of

$prune(M^{(h_1,h_2)}[i-1,j-1],(i',j'))$

Deactivate tables $(h_1,h_2) \in H$ where $h_1 \in H_1$.

If $i = n$, also deactivate tables $(h_1,h_2) \in H$ where $h_2 \in H_2$.

Update $H$ to remove deactivated tables.

The overall weight produced by this modified algorithm is found in the final calculated value, $T^{(0,0)}[n,m]$.

# References


[AC75] A. Aho and M. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the Association for Computing Machinery* **18** (1975), 333-340.

[ADF93] K. Abramson, R. Downey, and M. Fellows. Fixed-parameter intractability II. *Proceedings of the 10th Symposium on Theoretical Aspects of Computer Science (STACS)* (1993), 374-385.

[ADKF70] V. Arlazarov, E. Dinic, M. Kronod, and I. Faradzev. On economic construction of the transitive closure of a directed graph. Originally in (Russian) *Doklady Akademii nauk SSR* **194** (1970), 487-488. Translated in *Soviet Mathematics – Doklady* **11** (1975), 1209-1210.

[AKOF99] T. Akutsu, K. Kanaya, A, Ohyama, and A. Fujiyama. Matching of spots in 2D electrophoresis images. Point matching under non-uniform distortions. *Proceedings of Combinatorial Pattern Matching '99*, Springer-Verlag, **LNCS 1645** (1999), 212-222.

[AT93] A. Aszodi and W. Taylor. Connection topology of proteins. *Computer Applications in the Biosciences* **9** (1993), 523-529.

[BDFW95] H. Bodlaender, R. Downey, M. Fellows, and H. Wareham. The parameterized complexity of sequence alignment and consensus. *Theoretical Computer Science* **147** (1995), 31-54.

[BDFHW95] H. Bodlaender, R. Downey, M. Fellows, M. Hallett, and H. Wareham. Parameterized complexity analysis in computational biology. *Computer Applications in the Biosciences* **11** (1995), 49-57.

[BE94] S. Bird and T. Ellison. One-level phonology: autosegmental representations and rules as finite automata. *Computational Linguistics* **20** (1994), 55-90.

[BEF96] H. Bodlaender, P. Evans, and M. Fellows. Finite-state computability of annotations of strings and trees. *Proceedings of Combinatorial Pattern Matching '96*, Springer-Verlag, **LNCS 1075** (1996), 384-391.

[BG97] T. Bailey and M. Gribskov. Score distribution for simultaneous matching to multiple motifs. *Journal of Computational Biology* **4** (1997), 45-59.

[BM77] R. Boyer and J. Moore. A fast string searching algorithm. *Communications of the Association for Computing Machinery* **20** (1977), 762-772.

[BMR96] V. Bafna, S. Muthukrishnan, and R. Ravi. Computing similarity between RNA strings. *DIMACS Technical Report* **96-30**, 1996.

[CL94] W. Chang and E. Lawler. Sublinear approximate string matching and biological applications. *Algorithmica* **12** (1994), 327-344.

[CM92] C. Chevalet and B. Minchot. An algorithm for comparing RNA secondary structures and searching for similar substructures. *Computer Applications in the Biosciences* **8** (1992), 215-225.

[CM94] F. Corpet and B. Minchot. RNAlign program: alignment of RNA sequences using both primary and secondary structures. *Computer Applications in the Biosciences* **10** (1994), 389-399.

[Coo71] S. Cook. The complexity of theorem-proving procedures. *Proceedings of the Third ACM Symposium on the Theory of Computing* (1971), 151-158.

[CP94] F. Chin and C. Poon. Performance analysis of some simple heuristics for computing longest common subsequences. *Algorithmica* **12** (1994), 293-311.

[DEF93] R. Downey, P. Evans, and M. Fellows. Parameterized learning complexity. *Proceedings of the Sixth Annual ACM Workshop on Computational Learning Theory* (1993), 51-57.

[DF92] R. Downey and M. Fellows. Fixed-parameter intractability (extended abstract). *Proceedings of the Seventh Annual Conference on Structure in Complexity Theory* (1992), 36-49.

[DF99] R. Downey and M. Fellows. *Parameterized Complexity.* Springer-Verlag, 1999.

[DS94] S. Dong and D. Searls. Gene structure prediction by linguistic methods. *Genomics* **23** (1994), 540-551.

[ECU96] P. Evans, K. Cattell, and C. Upton. MF_Align: a tool to search the SWISS-PROT database colored with PROSITE motif information. *Manuscript*, 1996.

[Eva99] P. Evans. Finding common sequences with arcs and pseudoknots. *Proceedings of Combinatorial Pattern Matching '99*, Springer-Verlag, **LNCS 1645** (1999), 270-280.

[FMKH91] K. Flaherty, D. McKay, W. Kabsch, K. Holmes. Similarity of the three-dimensional structures of actin and the ATPase fragment of a 70-kDa heat shock cognate protein. *Proceedings of the National Academy of Sciences, USA* **88** (1991), 5041-5045.

[GBN94] D. Gusfield, K. Balasubramanian, and D. Naor. Parameterized optimization of sequence alignment. *Algorithmica* **12** (1994), 312-326.

[GJ79] M. Gary and D. Johnson. *Computers and Intractability.* W. H. Freeman and Company, 1979.

[GLE90] M. Gribskov, R. Lüthy, and D. Eisenberg. Profile analysis. *Methods in Enzymology* **183** (1990), 146-159.

[GME87] M. Gribskov, A. McLachlan, and D. Eisenberg. Profile analysis: detection of distantly related proteins. *Proceedings of the National Academy of Sciences, USA* **84** (1987), 4355-4358.

[Gur89] E. Gurari. *An Introduction to the Theory of Computation*, Computer Science Press, 1989.

[Hal96] M. Hallett. *An Integrated Complexity Analysis of Problems from Computational Biology*, Ph.D. Thesis, University of Victoria, 1996.

[Hir75] D. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the Association for Computing Machinery* **18** (1975) 341-3.

[Hir75] D. Hirschberg. Algorithm for the longest common subsequence problem. *Journal of the Association for Computing Machinery* **24** (1977), 664-75.

[Hir83] D. Hirschberg. Recent results on the complexity of common subsequence problems. In D. Sankoff and J. Kruskal (eds.) *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, Addison-Wesley, 1983, 325-330.

[HK93] K. Han and H. Kim. Prediction of common folding structures of homologous RNAs. *Nucleic Acids Res.* **21** (1993), 1251-1257.

[IF92] R. Irving and C. Fraser. Two algorithms for the longest common subsequence of three (or more) strings. *Proceedings of Combinatorial Pattern Matching '92*, Springer-Verlag, **LNCS 644** (1992), 214-229.

[JT95] T. Jiang and V. Timkovsky. Shortest consistent superstrings computable in polynomial time. *Theoretical Computer Science* **143** (1995), 113-122.

[JTT92] D. Jones, W. Taylor, and J. Thornton. A new approach to protein fold recognition. *Nature* **358** (1992), 86-89.

[Kar93] R. Karp. Mapping the genome: some combinatorial problems arising in molecular biology. *Proceedings of the Twenty-Fifth ACM Symposium on the Theory of Computing* (1993), 278-285.

[KKS92] I. Koch, F. Kaden, and J, Selbig. Analysis of protein sheet topologies by graph theoretic methods. *PROTEINS: Structure, Function, and Genetics* **12** (1992), 314-323.

[KMP77] D. Knuth, J. Morris, and V. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing* **6** (1977), 323-350.

[LRV98] H. Lenhof, K. Reinert, and M. Vingron. A polyhedral approach to RNA sequence structure alignment. *Proceedings of the Second Annual International Conference on Computational Molecular Biology (RECOMB 98)* (1998), 153-159.

[Mai78] D. Maier. The complexity for some problems on subsequences and supersequences. *Journal of the ACM* **25** (1978), 322-336.

[MARW89] E. Mitchell, P. Artymiuk, D. Rice, and P. Willett. Use of techniques derived from graph theory to compare secondary structure motifs in proteins. *Journal of Molecular Biology* **212** (1989), 151-166.

[MDM95] E. Mayoraz, I. Dubchak, and I Muchnik. Relation between protein structure, sequence homology, and composition of amino acids. *DIMACS Technical Report* **95-6** (1995).

[MJT96] R. Miller, D. Jones, and J. Thornton. Protien fold recognition by sequence threading: tools and assessment techniques. *The FASEB Journal* **10** (1996), 171-178.

[MP80] W. Masek and M. Paterson. A faster algorithm for computing string-edit distances. *Journal of Computer and Systems Sciences* **20** (1980), 18-31.

[MP83] W. Masek and M. Paterson. How to compute string-edit distances quickly. In D. Sankoff and J. Kruskal (eds.) *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, Addison-Wesley, 1983, 337-349.

[MSOM89] H. Margalit, B. Shapiro, A. Oppenheim, and J. Maizel. Detection of common motifs in RNA secondary structures. *Nucleic Acids Research* **17** (1989), 4829-4845.

[NW70] S. Needleman and C. Wunsch. A general method applicable to the search for similarities in the amino-acid sequence of two proteins. *Journal of Molecular Biology* **48** (1970), 443-453.

[PA92] S. Pascarella and P. Argos. A data bank merging related protein structures and sequences. *Protein Engineering* **5** (1992), 121-137.

[PL88] W. Pearson and D. Lipman. Improved tools for biological sequence comparison. *Proceedings of the National Academy of Sciences, USA* **85** (1988), 2444-2448.

[PM92] W. Pearson and W. Miller. Dynamic programming algorithms for biological sequence comparison. *Methods in Enzymology* **210** (1992), 575-601.

[RSDS94] B. Rost, R. Schneider, A. de Daruvar, and C. Sander. The PredictProtein server. *http://www.embl-heidelberg.de/predictprotein/predictprotein.html*

[San72] D. Sankoff. Matching sequences under deletion/insertion constraints. *Proceedings of the National Academy of Sciences, USA* **69** (1972), 4-6.

[San85] D. Sankoff. Simultaneous solution of the RNA folding, alignment, and protosequence problems. *SIAM Journal of Applied Mathematics* **45** (1985), 810-825.

[Sea95] D. Searls. Formal grammars for intermolecular structure. *Proceedings of the International IEEE Symposium on Intelligence in Neural and Biological Systems* (1995).

[SD93] D. Searls and S. Dong. A syntactic pattern recognition system for DNA sequences. *Proceedings of the Second International Conference on Bioinformatics, Supercomputing, and Complex Genome Analysis* (1993), 89-101.

[SW81] T. Smith and M. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology* **147** (1981), 195-197.

[Ste94] G. Stephen. *String Searching Algorithms.* Lecture Notes Series on Computing, vol. 3. World Scientific, 1994.

[Tay94] W. Taylor. Motif-biased sequence alignment. *Journal of Computational Biology* **1** (1994), 297-310.

[Ull76] J. Ullmann. An algorithm for subgraph isomorphism. *Journal if the Association for Computing Machinery* **23** (1976), 31-42.

[Wat95] M. Waterman. *Introduction to Computational Biology.* Chapman & Hall, 1995.

[WBWRS94] D.S. Wishart, R.F. Boyko, L. Willard, F.M. Richards, and B.D. Sykes. SEQSEE: A comprehensive program suite for protein sequence analysis. *Computer Applications in the Biosciences* **10** (1994), 121-132.

[WC+94] J. Wang, G. Chirn, T. Marr, B. Shapiro, D. Shasha, and K. Zhang. Combinatorial pattern discovery for scientific data: some preliminary results. *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data (SIGMOD)* (1995), 115-125.

[WF74] R. Wagner and M. Fischer. The string-to-string correction problem. *Journal of the Association for Computing Machinery* **21** (1974), 168-173.

[WP84] M. Waterman and M. Perlwitz. Line geometries for sequence comparisons. *Bulletin of Mathematical Biology* **46** (1984), 567-577.

[WS78] M. Waterman and T. Smith. RNA secondary structure: a complete mathematical analysis. *Mathematical Biosciences* **42** (1978), 257-266.

[WZ99] Z. Wang and K. Zhang. Finding common RNA secondary structures from

RNA sequences. *Proceedings of Combinatorial Pattern Matching '99*, Springer-Verlag, **LNCS 1645** (1999), 258-269.

[WZS94] J. Wang, K. Zhang, and D. Shasha. Pattern matching and pattern discovery in scientific, program, and document databases (abstract). *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data (SIGMOD)* (1994), 487.

[YK95] T. Yokomori and S. Kobayashi. DNA evolutionary linguistics and RNA structure modeling: a computational approach. *Proceedings of the International IEEE Symposium on Intelligence in Neural and Biological Systems* (1995).

[Zha98] K. Zhang. Computing similarity between RNA secondary structures. *Proceedings of IEEE International Joint Symposia on Intelligence and Systems* (1998), 126-132.

[ZWM99] K. Zhang, L. Wang, and B. Ma. Computing similarity between RNA structures. *Proceedings of Combinatorial Pattern Matching '99*, Springer-Verlag, **LNCS 1645** (1999), 281-293.

# VITA

| | |
|---|---|
| Surname: **Evans** | Given Names: **Patricia Anne** |
| Place of Birth: **Fergus, Ontario, Canada** | Date of Birth: **Dec. 17, 1969** |

Educational Institutions Attended:

| | |
|---|---|
| University of Victoria | 1991 to 1999 |
| University of Alberta | 1987 to 1991 |

Degrees Awarded:

| | | |
|---|---|---|
| B. Sc. (Honours) | University of Alberta | 1991 |
| M. Sc. | University of Victoria | 1994 |

Honors and Awards:

| | |
|---|---|
| NSERC Postgraduate Scholarship | 1991 to 1993 |
| President's Award | 1991 to 1993 |
| Advanced Systems Institute Scholarship | 1991 to 1992 |
| NSERC Summer Research Scholarship | 1990 |
| Barry J. Mailloux Prize in Computing Science | 1989 |
| Louise McKinney Scholarship | 1988 to 1989 |
| University Women's Club of Edmonton Scholarship | 1987 to 1989 |
| Rutherford University Entrance Scholarship | 1987 |

Publications:

with L. Clark, M. Fellows, and W. Wallis. Algorithms for learning and teaching sets of vertices in graphs. *University of Victoria Technical Report DCS-212-IR* (1993).

with R. Downey and M. Fellows. Parameterized learning complexity. *Proceedings of*

*the Sixth Annual ACM Workshop on Computational Learning Theory* (1993), 51-57.

with J. Culberson. Asymmetry in Binary Search Tree Update Algorithms. *University of Alberta Computing Science Technical Report TR 94-09*, 1994.

with M. Fellows. Bounding the Vapnik-Chervonenkis dimension of DNF via learning reductions. *Proceedings of the Machine Learning Workshop at* (1994), v1-v8.

*The Learnability and Teachability of Vertex Sets.* Master's Thesis, University of Victoria, 1994.

with H. Bodlaender and M. Fellows. Finite-state computability of annotations of strings and trees. *Proceedings of Combinatorial Pattern Matching '96*, Springer-Verlag LNCS 1075 (1996), 384-391.

Finding common subsequences with arcs and pseudoknots. *Proceedings of Combinatorial Pattern Matching '99*, Springer-Verlag LNCS 1645 (1999), 270-280.

# Partial Copyright License



Title of Thesis:

# Algorithms and Complexity for Annotated Sequence Analysis

Author: ________________________________

PATRICIA ANNE EVANS

August 24, 1999