2015

The Longest Common Subsequence via Generalized Suffix Trees

Tazin Afrin

The Longest Common Subsequence via Generalized Suffix Trees

Tazin Afrin

Thesis submitted to the
College of Engineering and Mineral Resources
at West Virginia University
in partial fulfillment of the requirements for the degree of

Master of Science in
Computer Science

Donald Adjeroh, Ph.D., Chair
Elaine Eschen, Ph.D.
Katerina Goseva Popstojanova, Ph.D.

Lane Department of Computer Science and Electrical Engineering
Morgantown, West Virginia
2015
Keywords: longest common subsequence, LCS, generalized suffix tree, compression

ABSTRACT

The Longest Common Subsequence via Generalized Suffix Trees

by
Tazin Afrin

Given two strings S_1 and S_2, finding the longest common subsequence (LCS) is a classical problem in computer science. Many algorithms have been proposed to find the longest common subsequence between two strings. The most common and widely used method is the dynamic programming approach, which runs in quadratic time and takes quadratic space. Other algorithms have been introduced later to solve the LCS problem in less time and space. In this work, we present a new algorithm to find the longest common subsequence using the generalized suffix tree and directed acyclic graph.

The Generalized suffix tree (GST) is the combined suffix tree for a set of strings $\{S_1, S_2, ..., S_n\}$. Both the suffix tree and the generalized suffix tree can be calculated in linear time and linear space. One application for generalized suffix tree is to find the longest common substring between two strings. But finding the longest common subsequence is not straight forward using the generalized suffix tree. Here we describe how we can use the GST to find the common substrings between two strings and introduce a new approach to calculate the longest common subsequence (LCS) from the common substrings. This method takes a different view at the LCS problem, shading more light at novel applications of the LCS. We also show how this method can motivate the development of new compression techniques for genome resequencing data.

Acknowledgements

I would like to express deepest gratitude to my MS thesis supervisor, Dr. Donald A. Adjeroh. I am grateful for the seed of excitement and passion for research he sowed in me. I thank him for his guidance and support and I appreciate his knowledge and wit. I want to thank my committee members, Dr. Elaine M. Eschen and Dr. Katerina Goseva Popstojanova, for their suggestions and support throughout the coursework and my entire research period. The knowledge I obtained from their courses have been fundamental for my thesis. I also want to thank Dr. Richard Beal for helping me in every aspect of this thesis. My deepest gratitude to my father Shahidul Alam Akanda, and my mother Rokeya Alam for their unconditional love and support. I want to thank my husband Syed Ashiqur Rahman, for I would not be who I am today without his care and love, and for always being there for me. I express my obedience and gratitude to my creator and my sustainer Allah, for I believe His plans are better than my dreams.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem and Motivation

Measuring similarity between sequences, be it DNA, RNA, or protein sequences, is at the core of various problems in molecular biology. An important approach to this problem is computing the longest common subsequence between two strings. Given a string S of length n, a subsequence is a string $S[i_1]S[i_2]\dots S[i_k]$ such that $1 \leq i_1 < i_2 < \dots < i_k \leq n$ for some $k \leq n$. A substring is a subset of characters from S that are located contiguously, but in a subsequence the characters are not necessarily contiguous, just in the same order from left to right. Given S_1 and S_2, the *longest common subsequence* (LCS) problem is to find one or all the longest common subsequences between them. As an example, there are two LCS's for the pair of strings (`abba`, `abab`), which are `abb` and `aba`. Since `aba` can be derived from `abba` in two different ways, this gives three distinct solutions. Finding the LCS between two strings is relatively easy, the real challenge is to do this in a time- and space-efficient manner.

A generalization of the LCS problem is to find the LCS for a set of three or more sequences. This is the multiple longest common subsequence problem, which is known to be NP-hard for an arbitrary number of sequences [24]. Comparing biological sequences using the LCS can be done

in various ways, for instance, by using the length of the LCS directly, or with some normalization [23], using the number of distinct LCS's between the sequences, analyzing the LCS itself, counting the number of all distinct common subsequences between the sequences [12, 36], etc.

The LCS problem is a classical problem in computer science (see [2,16]), and has been used to study applications in various areas, such as text analysis, pattern recognition, file comparison, efficient tree matching [23], etc. Biological applications of the LCS, and similarity measurement in general, are varied and numerous, from sequence alignment [30] in comparative genomics [1], to phylogenetic construction and analysis, to rapid search in huge biological sequences [33], to compression and efficient storage of the rapidly expanding genomic data sets [14, 34], to re-sequencing a set of strings given a target string [21], which is important in efficient genome assembly.

1.2 Thesis Contribution

In this work we introduce an innovative and fundamentally different approach to the LCS problem. We make the use of suffix data structures, and graph algorithms. The generalized suffix tree (GST) is the core data structure used to identify the common substrings (CSSs) between two strings S_1 and S_2. Our main contribution is in exploiting the CSSs from the GST to determine the LCS. First, we show how we can build a backbone structure to represent all the CSSs between two strings. Then, based on the backbone structure we construct a directed acyclic graph (DAG) to capture the partial ordering inherent in the CSSs. Using the DAG, we compute the LCS as the longest path in the DAG. Our main algorithm to construct directed acyclic graph, DAG from CSSs runs in $O(max\{\eta^2, \eta \times (n+m)\})$ time in the worst case and in $O(max\{\eta^2, \eta \times |\Sigma|\})$ on average. The worst case is as good as other current algorithms. Our significant contribution is the average case complexity, which is better than all the current methods for smaller size alphabet like DNA, RNA, Protein alphabet. Another major contribution is in the presentation of initial ideas towards the use of variable-length CSSs for improved LCS computation. The final contribution is in the use of the LCS in one specific application of the LCS: genomic data compression. Specifically, we introduce an LCS-motivated reference-based approach to

compress genome re-sequencing data. We compared our results with those from recently published genome re-sequencing data compression algorithms.

1.3 Thesis Outline

Before jumping into our original thesis contribution, in Chapter 2 we introduce some basic concepts in the area of string algorithms. We describe a powerful data structure, the suffix tree and its variation the generalized suffix tree. We also discuss some related algorithms introduced by other researchers to solve the longest common subsequence problem.

Chapter 3 is our main contribution in the report. We introduce the new algorithm to calculate the longest common subsequence using generalized suffix tree and analyze the worst case and average case complexity. At the end of this chapter, we also give some ideas on how to improve our current algorithm that is based on length-1 common substrings, and how to improve it to use substrings for length more than or equal to 2. We also show how LCS influences the compression of genomic resequencing data. Two new compression methods are described here and we give compression results using one of the methods. Chapter 4 summerizes our contribution and concludes the report with a proposal for future work.

Chapter 2

Background and Related Work

In this Chapter, we describe some basic concepts and terminology of string algorithms. First, the suffix tree data structure is described and we show it's important application in pattern matching. Then the generalized suffix tree (GST) and its applications are discussed. The GST is the basis for our new algorithm introduced in Chapter 3. The basic idea of compressing genomic resequencing data is described here, because our algorithm inspires some new technique for the compression. We also describe other efforts related to the thesis.

2.1 Suffix Trees and Pattern Matching

The suffix tree is a special and interesting data structure. It is well known for its simplicity, compactness and ease of computation. Pattern matching, maximal repetitions, sequence alignments are among several important applications of the suffix tree. In this section, we describe the suffix tree data structure, its variations, and applications.

2.1.1 Suffix Tree

Suffix tree is a compressed trie data structure that is used to represent all the suffixes of a given string S. The nodes of the tree contains the given text as their keys and the starting position of the suffixes in the text as their values. The suffix tree provides more compact representation of the suffixes of a string and exposes the internal structure of a string in a deeper way [16]. In fact, the suffix tree is better than the trie in terms of space. While trie could be quadratic, the suffix tree always takes linear space with respect to the length of the string. Here is the definition of the suffix tree from Dan Gusfield's book [16]:

Definition. A suffix tree T for an n-character string S is a rooted directed tree with exactly n leaves numbered 1 to n . Each internal node, other than the root, has at least two children and each edge is labeled with a nonempty substring of S. No two edges out of a node can have edge-labels beginning with the same character. The key feature of the suffix tree is that for any leaf i , the concatenation of the edge-labels on the path from the root to leaf i exactly spells out the suffix of S that starts at position i. That is, it spells out $S[i...n]$ [16].

For simplicity in constructing the suffix tree, it is ensured that no suffix is a proper prefix of the string. This can be done by adding a sentinel character, for example \$, where $\$ \notin \Sigma, \$ < \sigma, \forall \sigma \in \Sigma$, at the end of the string S. This makes sure each suffix has its unique branch till the leaf node. Thus the given string $S = S[1..n]\$$ of length n becomes of length $n + 1$ with the \$ symbol concatenated at the end. Here we represent some properties of the suffix tree T of S from [2]:

- The suffix tree T has exactly $n + 1$ leaf nodes;
- T has at most n internal nodes, considering the root node as an internal node
- No two edges out of a given internal node can have edge-labels that start with the same symbol.

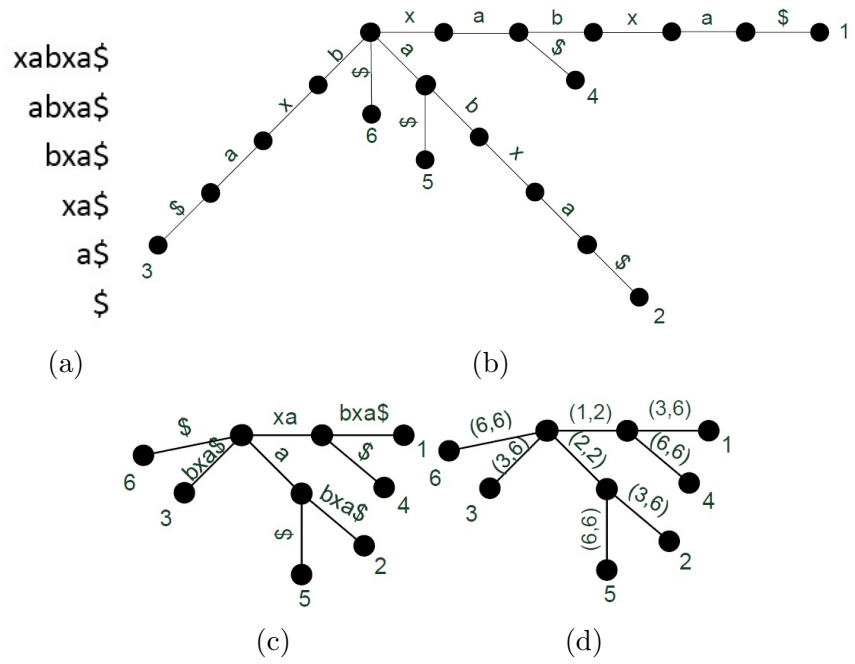

Figure 2.1: Suffix structures for the string $S = xabxa\$$: (a) list of suffixes, (b)suffix trie, (c)suffix tree, (d)ST with edge-lebel compression.(from [16])

- Every internal node has at least 2 outgoing edges and at most $|\Sigma| + 1$ outgoing edges.
- Every distinct substring of T is encoded exactly once in the suffix tree and is spelled out by traveling from the root node to the leaf node.
- The *edge label* is the corresponding substring in S. It is denoted with the numbers of the starting and ending points of the substring.
- Pointers to the suffix of the original string is used for *edge-label* compression.
- The path from the root to the i-th leaf represents the i-th suffix $S[i....n-1]$.

The list of the suffixes and the suffix tree for the string $S = xabxa\$$ is shown in Figure 2.1. On the figure, if we spell out the path from root to leaf number 2, it spells $abxa\$$. This is equal to the suffix starting at position

2 of the string S. From the figure we can understand the key point of suffix tree, that it captures the repitition in the text. The numbers on the leaf nodes give all the occurrence position of the suffixes of the string S.

History of suffix tree

The suffix tree was first introduced by Weiner in 1973. He called it a bi-tree or position tree associated with a string. He presented a linear time algorithm for the construction of the compacted version of bi-tree and also showed how to solve some interesting pattern matching problem using the then new data structure [37]. Few years later in 1976, McCreight gave a more space efficient algorithm to construct the suffix tree in linear time [25]. Then in 1995 Ukkonen developed a conceptually different algorithm for building the suffix tree in linear time [31]. This is the most popular algorithm to build the suffix tree because of its simplicity. It is easy to understand and implement. Ukkonen's algorithm takes relatively smaller memory than the other suffix tree algorithms. Also it is an online algorithm to contruct the suffix tree, that means one can build the suffix tree for streaming data.

Complexity

Suffix trees are very space efficient and give solutions to a wide range of complex string problems. If the length of the given string $|S| = n$, the average depth of a suffix tree is $\log n$ and the upper bound on the number of nodes in a suffix tree is $2n - 1$. Suffix tree will only store the original text S, node labels for both branching and leaf nodes, edge-labels, the space to indicate the parent for each node and the suffix links [2]. So it requires n bytes for ASCII text. The edge labels are represented by a pair of indices that specify the beginning and end positions of a substring of the text, so it takes $2n$ integer pointers for edge labels. It is the same for the node parents, $2n$ integer pointers, $2n$ integer pointers for node labels for both branching and leaf nodes and $2n$ integer pointers for suffix links. Thus to store the suffix tree we need $33n$ bytes in total. A pointer is represented using an integer which is 4 bytes long and 1 byte is used to represent each character of the text. Hence if there are $O(n)$ pointers and $\log n$ bits are required to

encode each pointer, then the space consumed by a suffix tree is $O(n \log n)$ bits [2].

2.1.2 Repeating Substrings

The longest repeated substring problem is a classical string matching problem in computer science. It is to find the longest substring of a string that occurs repeatedly, or at least twice in the string. The suffix tree data structure is used to solve this problem in linear time and space. This can be done by building a suffix tree for the string and finding the deepest internal node in the tree. The path from the root to that deepest internal node is the desired repeating substring for the given string. Also the suffix tree can be used to find a substring with at least k-occurrences [16].

Repetitive Structure of Biological Sequences

There are various kinds of repititive structures in a string. The principle interest of finding these structures lies in the repetitive structure of biological sequences - DNA, RNA, and protein. The extent of repeated substrings occurrence in DNA (or other genome) is surprisingly high. Most of huge and long genome sequences consists of comparatively shorter repeating genome strings. The repetition structure contains various biological functions that are very interesting for many researchers. Another advent of the repetitive structure is to find out how to store the biological sequence in smaller space, that is, the problem of compression of the biological sequences [2]. Here we give some definition of the repititive structures:

Palindrome A palindrome is a sequence of characters that reads the same backward or forward [16].

$A - T$ are complements and $C - G$ are complements in DNA sequence. Also in RNA sequence $A - U$ and $C - G$ are complements. Complemented Palindrome is determined based on this characteristics of DNA and RNA.

Complemented Palindrome If each character in one half of the DNA or RNA string is changed to its complement character, then the sequence becomes a complimented palindrome. For example, AGCTCGCGAGCT is a complemented palindrome [16].

Tandem Repeat A substring α contained in string S is called a tandem array of β if α consists of more than one consecutive copy of β. For example, if $S = aaatactactacggga$, then $\alpha = tactactac$ is a tandem array of $\beta = tac$.

When a pattern of one or more nucleotides is repeated in the DNA and the repetitions are directly adjacent to each other, then it is called the tandem repeats in the genomic sequences. Variable number of tandem repeats have become an important marker of individuals [16]. Sometimes tandem repeats consists of very short substrings, and they have become preffered marker for many genetic marking applications.

Maximal Pair A maximal pair in a string S is a pair of identical substrings α and β in S such that extending α and β in either direction would destroy the equality of the two strings [16].

A maximal pair is represented by a triple (starting position of α, starting position of β, length of match). For a string of length n, it is possible to find all the maximal repeats in $O(n)$ time using the suffix tree. Also if there are η_{occ}-maximal pairs, then finding all the maximal pairs takes $O(n + \eta_{occ})$ time [16].

2.1.3 Generalized Suffix Tree

When applications involve mutiple strings, suffix tree could be implemented for each string seperately. But it will take more time and space. Hence the notion of generalized sufffix tree has been introduced. It is a better approach than creating suffix trees for each string. A generalized suffix tree (GST) is a suffix tree for a set of strings $\{S_1, S_2, ..., S_n\}$. It contains all the suffixes of all the strings. It can also be built in linear time and space and can be used to find all occurrences of the common substrings between the given strings.

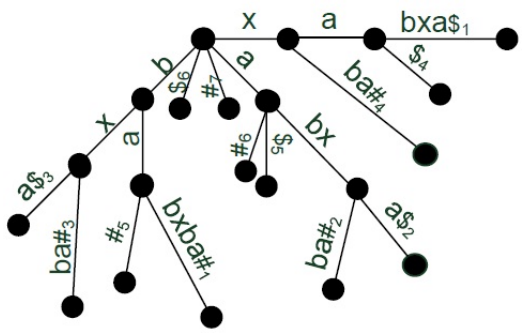

Figure 2.2: Generalized suffix tree for the concatenated string $S = S_1\$S_2\# = xabxa\$babxba\#$ for $S_1 = xabxa$ and $S_2 = babxba$(from [16]). For brevity, the labels for each leaf node is cut at the position of the special symbol.

Constructing the generalized suffix tree is very simple and it uses the same approach as normal suffix tree. Each string in the set is padded with a unique end of string marker (different marker for each string). The concatenated string will look like, $S = S_1\$_1S_2\$_2...S_n\$_n$. Then a suffix tree is built for S, which will be the generalized suffix tree for $\{S_1, S_2, ..., S_n\}$. In the generalized suffix tree, a path label should have substring from only one input string. So if there are path labels having substrings from multiple input strings, we can keep only the initial portion corresponding to one string and remove all the later portion. Because each end of string marker is distinct and is not in any of the original strings. The label on any path from the root to an internal node must be a substring of both original strings. Hence by reducing the second index of the label on leaf edges, without changing any other parts of the tree, all the unwanted synthetic suffixes are removed. Hence, GST guarantees that each suffix is represented by a unique path on the tree [2, 16].

An example of the generalized suffix tree is shown in Figure 2.2. Given $S_1 = xabxa$ and $S_2 = babxba$, the strings are padded with terminal symbols \$ and #. So the final string becomes, $S = S_1\$S_2\# = xabxa\$babxba\#$. Then we build the suffix tree for S, and that is the generalized suffix tree for S_1 and S_2. The well established traditional suffix tree algorithms can be used to build the GST. Ukkonen's and McCreight's [25, 31] algorithms can be used to construct the generalized suffix tree. GST is still linear with respect to the length of the string.

2.2 Longest Common Subsequence

Given a set of strings or sequences, the longest common subsequence (LCS) problem is to find the longest consecutive subsequences common to all the strings. This problem is different from that of finding substrings between strings, since in longest common sequence problem the subsequences need not to be adjacent to each other in each string. Rather we can say the longest common subsequence is a combination of common substrings. The longest common sequence is often calculated between two strings S_1 and S_2. There are several methods to compute the LCS between two strings. In this section we describe some of the approaches.

2.2.1 LCS Algorithms

The LCS problem is a special case of the general edit distance problem, which is also used to measure similarity between strings. Depending on the edit cost for the three basic edit operations, the edit distance is related to the LCS using the simple relation: $d = m + n - 2l$, where d is the edit distance, and l is the length of the LCS. For instance, the edit distance between *abba* and *abab* is 2 ($= 4 + 4 - 2 \times 3$). However, a direct computation of the LCS using dynamic programming is more efficient than going through the edit distance computation first, although the asymptotic complexity of $O(mn)$ remains the same.

The basic approach to compute the LCS, between the n-length S_1 and m-length S_2, is via dynamic programming (DP). The formulation is given as follows:

$$LCS(0,0) = 0,\ LCS(i,0) = 0,\ LCS(0,j) = 0$$
$$LCS(i,j) = 1 + LCS(i-1, j-1) \text{ if } S_1[i] = S_2[j]$$
$$LCS(i,j) = \max\{LCS(i,j-1), LCS(i-1,j)\} \text{ if } S_1[i] \neq S_2[j]$$

The above computes the length of the LCS in the last position of the table ($LCS(n,m)$). As with the edit distance, the actual string forming the LCS can be obtained by using a trace back on the DP table. This

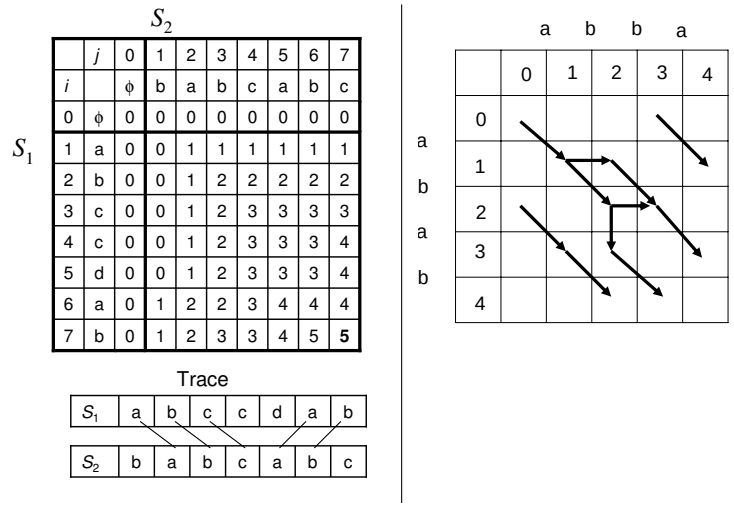

Figure 2.3: Computing the LCS: (a): dynamic programming table and a trace; (b): edit graph.(figures taken from [2])

requires $O(mn)$ time and $O(mn)$ space. See Figure 2.3(a) for an example. The LCS matrix has some interesting properties: the entries in any row or in any column are monotonically increasing, and between any two consecutive entries in any row or column, the difference is either 0 or 1.

An improved approach is to formulate the problem on a two dimensional grid, where the goal is to find the minimal cost (or maximal cost, depending on the formulation) path, from the start position on the grid (typically, (0,0)), to the end position (m, n). Myers et al. [26] and Ukkonen [31] used this idea to propose a minimum cost path determination problem on the grid, where the path takes a diagonal line from $(i-1, j-1)$ to (i, j) if $S_1[i] = S_2[j]$ with cost 0, and takes a horizontal or vertical line with a cost of 1, corresponding respectively to insert or delete operations. Hunt and Szymanski [19] earlier used an essentially similar approach to solve the LCS problem in $(r+n)\log n$ time, with $m < n$, where r is the number of pairwise symbol matches ($S_1[i] = S_2[j]$). When two non-similar files are compared, we will have $r \ll mn$, or r in $O(n)$, leading to a practical $O(n \log n)$ time algorithm. However, for very similar files, we have $r \approx mn$, or an $O(mn \log n)$ algorithm. This worst case occurs, for instance, when $S_1 = a^n$ and $S_2 = a^m$. The grid-based approaches can be easily visualized with the aid of an edit graph as shown in Figure 2.3(b) for the sequences S_1=`abba` and S_2=`abab`.

The end points of the diagonals then define an LCS. Hirschberg [18] proposed space-efficient approaches to compute the LCS using DP in $O(mn)$ time and $O(n+m)$ space, rather than $O(nm)$. More recently, Yang et al. [38] used the observation on monotonically increasing values in the LCS table to identify the "corner points", where the values on the diagonals change from one row to the next. The corners define a more sparse 2D grid, based on which they determine the LCS.

Another interesting view of the LCS problem is in terms of the longest increasing subsequence (LIS) problem, suggested earlier in [4,20,28], and described in detail in [16]. The LIS approach also solves the LCS problem in $O(r \log n)$ time (where $m \leq n$), the same time complexity as the grid based methods. In most practical scenarios, $r < mn$.

2.3 Directed Acyclic Graph

A graph G is a set of vertices V and edges E, where vertices are a finite set and edges are a binary relation on vertices. A directed graph is a graph where the edges are directional, going in one direction from one vertex to another. A cycle in graph is when we can start from one vertex and come back to that after visiting other vertices. An acyclic graph does not have any graph cycles. A directed acyclic graph (DAG) is a directed graph with no cycles. So a DAG is a graph $G(V, E)$, where it is not possible to start at some vertex $u \in V$ and looping back to u again. A good example of a directed acyclic graph is a tree.

The DAG is a very useful data structure to model various types of information. For example, partial ordering may be represented by a DAG using reachability. DAGs are also used in many applications to indicate precedence among events by running a topological sort to order all of its vertices linearly. Directed acyclic graphs can be used as a space-efficient representation of a collection of sequences with overlapping subsequences. The edges of a DAG can be weighted or unweighted.

Longest Path in a DAG

The longest path problem in a graph is to find the longest path from a starting vertex to an ending vertex. This is an NP-hard problem for general graph. But for a directed acyclic graph there is solution for given starting and ending points. Infact for DAG it can be done in linear time. We need the edges to be weighted to calculate the longest path from one node to another.

To find the longest path in a weighted DAG, we need to run a topological sort on the graph. It produces a linear ordering of vertices such that for every directed edge (u, v), vertex u comes before v in the ordering. We initialize distances to all vertices as minus infinite and distance to source as 0, then we find a topological sorting of the graph. Topological sorting of a graph represents a linear ordering of the graph. Once we have topological order (or linear representation), we process all vertices one by one in topological order. As we process the vertices, for each one, we update distances to the adjacent vertices using distance of current vertex. The running time of this algorithm is linear with respect to the number of vertices and edges. If the total number of vertices are $|V|$ and total number of edges are $|E|$, then the algorithm runs in $O(|V| + |E|)$ time [8].

This is the time to compute one longest path. Finding all the possible longest paths from start to end will take more time. If all the path length is L, then the algorithm will run in $O(|V| + |L|)$ time [8].

2.4 Genomic Sequence Compression

Compression of biological sequences is an important but difficult problem, which has been studied for decades by various authors [3,9,27]. See [14,15,32] for recent surveys. Most of the earlier studies focused on lossless compression, as it was believed that biological sequences should not admit any data loss, as that could impact later use of the compressed data. The earlier methods also generally exploited self-contained redundancies, without using a reference sequence. The advent of high-throughput next generation sequencing, with massive datasets that are easily generated for one experiment, have

challenged both compression paradigms. Thus, lossy compression of high throughput sequences admitting limited errors have been proposed in [13,17] for significant compression. Further, with the compilation of several reference genomes for different species, more recent methods have considered lossless compression of re-sequencing data by exploiting the significant redundancies between the genomes from related species. This observation is the basis of various recently proposed methods for reference-based lossless compression [22, 29, 35], whereby some available standard reference genome is used as the dictionary. Compression ratios in the order of 80 to 18,000 without loss have been reported [29, 35]. Here we describe the GRS and GReEN compression techniques that used LCS.

2.4.1 GRS Compression Technique

Wang et.al. [35] introduced a novel approach to compression and efficient storage of genomic resequencing data. They called their tool Genomic Re-Sequencing (GRS) data compression. Other available tools for compressing genomic sequence data has the limitation of requiring the single neucleotide polymorphism (SNP) map for the reference genome. But GRS does not need the SNP map or other variation information of the sequence.

The main module of GRS takes the genome chromosome files as input and ouputs the compressed target file and decompression command file. In Figure 2.4 we showed the GRS architechture. GRS first evaluated the varied sequence percentage (δ) of the target chromosome based on the reference chromosome. GRS can only compress the file if the δ value is ≤ 0.03. If $\delta \geq 0.1$ GRS cannot compress the file, otherwise the method modifies the input chromosome file and then run the compression. At first GRS filters the longest common sequence between the target and the reference files. This can be done effectively using the matrix graph [26]. GRS finds the minimal changes between two files using a modified UNIX **diff** program. They modify the UNIX **diff** program to remove redundant information and to reduce the file size. Then Huffman coding is used to encode the processed individual sequence data.

GRS was run on Korean personal genome sequence data (KO-

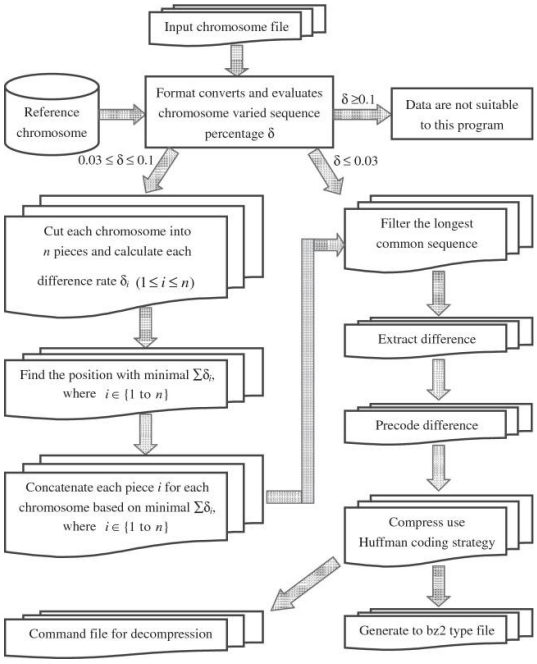

Figure 2.4: The GRS architechture (from [35]).

REF_20090131, KOREF_20090224), data from rice (TIGR5 and TIGR6) and *Arabidopsis Thaliana* (TAIR8 and TAIR9). GRS achieved 159 fold compression on the Korean genome data, 82 on rice genome and 18132 on the *A. Thaliana* genome [35]. They used LCS and modified UNIX **diff** function in their compression technique. Their compression methods and results inspired us to focus on longest common subsequence and use for compression.

2.4.2 GReEn Compression Technique

GReEN (Genome Resequencing Encoding) is also a tool for reference base genomic data compression. *GReEN* have been proposed to overcome some of the drawbacks of the *GRS* method. GRS cannot compress data when the target sequence is very different from the reference sequence. *GReEN* proposed a probabilistic copy model and have gained faster and better compression for some sequences. It performs compression when the target sequence is not very similar to the reference sequence, even when the species is different.In

general, the reference is only slightly different from the target sequence, but it is not a mandatory rule that it has to be from the same species. In fact target from different species is possible to use because genomic sequences are usually very similar to each other and the alphabet is same for most cases. For example, DNA will always have A, T, C, G in it. Though the reference need to be same when decompressing the sequence. The GReEN code is able to handle even arbitrary alphabets.

GReEN compression tool is based on arithmetic coding. Each character of target sequence is encoded. It is necessary to provide probability ditribution of each character to the arithmetic encoder. But the encoder can adjust the probability as it proceeds. This method estimates the probability of a character while encoding, which changes along the encoding process. *GReEN* uses the adaptive copy model for probability estimation first, then gradually it fixes itselt to the static probability distribution model. *GReEN* runs in linear time with respect to the size of the sequences. They have used the same dataset used in *GRS*. *GReEN* have achieved over 100-fold compression for some datasets compared to *GRS* [29].

Chapter 3

Methodology

In this chapter, we propose a new algorithm to find the longest common subsequence (LCS) between two strings S_1 and S_2. The new algorithm uses the generalized suffix tree (GST) to compute all the common substrings (CSSs) between the two strings. Then we sort the CSSs based on the starting positions on S_1 and S_2 and build a table named $ConflictTable$ that stores if one CSS conflicts with another. Then from all common substrings we construct a directed acyclic graph (DAG) with starting and ending points, with the help of the $ConflictTable$. From the directed acyclic graph we directly find the longest path for the given starting to ending position, which is our desired longest common subsequence LCS. The overall algorithm is summerized as follows: (i) From GST to CSS, (ii) From CSS to DAG, and (iii) From DAG to LCS. Our current algorithm is based on length-1 common substrings extracted from the generalized suffix tree. In the second part of this chapter, we proposed how to increase the length of common substrings (CSSs) and how it can help us to gain a faster algorithm to calculate LCS in practice.

3.1 The New LCS Algorithm

First, we give definition of CSS between two given strings S_1 and S_2 and then the definition LCS in terms of the common substrings.

Definition 1. ***Common substrings*** *(CSS): Given two strings, n-length S_1 and m-length S_2, $CSSs$ between S_1 and S_2 is the set of strings $\mathcal{V} = \{v_1, ..., v_V\}$, where $v_i = S_1[p_{11} \dots p_{12}] = S_2[p_{21} \dots p_{22}]$ for some $p_{11} \leq p_{12} \leq n$ and $p_{21} \leq p_{22} \leq m$.*

Definition 2. ***Longest common subsequence*** *(LCS): For S_1 and S_2, the LCS between S_1 and S_2 is the longest sequence of string pairs $\mathcal{M} = \{m_1, ..., m_M\}$, where $\mathcal{M} \subset \mathcal{V}$ and $m_i = S_1[p_{11} \dots p_{12}] = S_2[p_{21} \dots p_{22}]$ for $1 \leq i \leq M$ and $m_i.p_1 < m_{i+1}.p_1 \ \wedge \ m_i.p_2 < m_{i+1}.p_2$ for $\forall(p_1, p_2)$.*

3.1.1 From GST to CSS

Here we describe how to compute the common substrings from the generalized suffix tree. From Section 2.1.3 we know that, a generalized suffix tree is used when more than one string is involved. For string S_1 and S_2 we can build a generalized suffix tree GST using the same construction method for suffix tree. There are several well known algorithms that constructs the suffix tree in linear time w.r.t. the length of the string [25, 31]. We can use Ukkonen's algorithm [31] to build the GST. Then on the GST we traverse to find the $CSSs$ and their positions in S_1 and S_2.

Here we use the same example as in Section 2.1.3. Given, $S_1 = xabxa$ and $S_2 = babxba$. We concatenate terminal strings \$ and # at the end of S_1 and S_2 respectively. Then concatenate S_1 and S_2 and get the string $S = S_1\$ \circ S_2\#$. We construct the generalized suffix tree using Ukkonen's Algorithm [31]. The GST is shown in Figure 3.1 with the suffix numbers for the strings S_1 and S_2. We do a pre-order traversal of the generalized suffix tree to get the CSSs. At each internal node N of the GST, we find all the suffix indices that starts with the node. The leaf nodes descending from the node-N represent the suffix numbers and terminal symbols \$ or # to indicate whether it is from S_1 or S_2. We can get variable number of suffix indices from

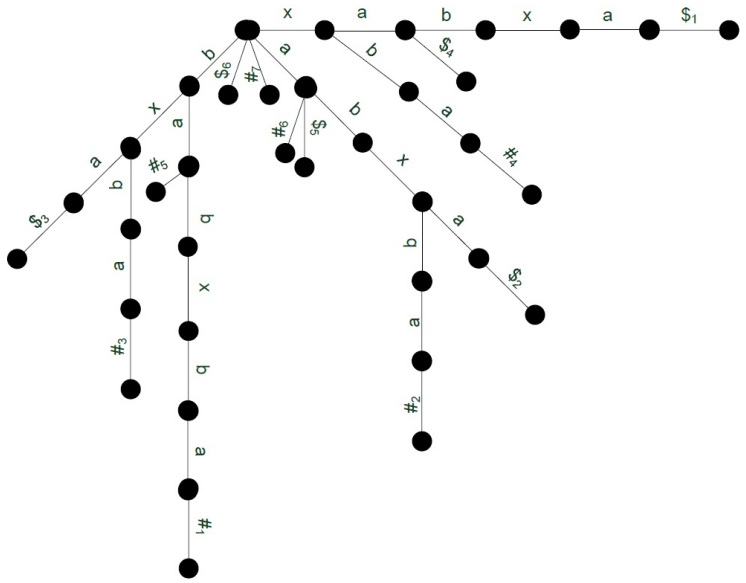

Figure 3.1: Generalized suffix trie for the concatenated string $S = S_1\$S_2\# = xabxa\$babxba\#$ for $S_1 = xabxa$ and $S_2 = babxba$ with suffix numbers.

S_1 and S_2. The string from root to node-N is the substring shared between the suffixes from S_1 and S_2.

In Figure 3.1 we can see that there is a branch with string b at the root of the tree. We go to depth-1 node along that path, we start collecting the suffixes descending that node. At this point, we have the suffix indices $\{3\}$ from S_1 and $\{1, 3, 5\}$ from S_2. We can see that the suffixes at these positions in $S_1 = xabxa$ and $S_2 = babxba$ starts with b. Let p_1 denote the suffix position in S_1 and p_2 denote the suffix position in S_2. So for previous example for the suffixes starting with b in depth-1 node we get three suffixes $\{(3, 1), (3, 3), (3, 5)\}$. So we have found three matches between S_1 and S_2, in other words we have found three CSSs with length-1.

We declare an empty set $\mathcal{A} = \emptyset$, that will store all the CSSs we collect. In $\mathcal{A}$ we store all the CSS with the suffix positions in S_1 and S_2. By traversing all the depth-1 nodes we can collect all the length-1 CSS matches between S_1 and S_2 and add them in $\mathcal{A}$.

If we want length-k CSSs, we need to traverse the nodes at depth-k

in the GST. Then we can collect all the suffixes shared between S_1 and S_2. Any length CSS list can be calculated by traversing all the nodes in the generalized suffix tree. However if a node-N contains suffixes from only one string, then the string from root to that node-N is not a shared or common string between S_1 and S_2. Hence it will not be included in the common substrings list in $\mathcal{A}$.

Algorithm 1 Compute the common substrings list from GST

```
1: Compute CSS List(S1,S2)
2: G ←Construct GST(S1,S2) /* using Ukkonen's Algorithm */
3: A = ∅
4: while G do
5:     Identify suffix indices descending at each node at depth-1
6:     while indices form a matching CSS do
7:         c ← CSS
8:         A ← A ∪ c
9:     end while
10: end while
11: return A
```

From the above description and Algorithm 1, it is clear that the running time is the maximum of the generalized suffix tree construction and the number of length-1 CSSs. The GST is linear w.r.t. the length of S_1 and S_2. If the length of S_1 is n and S_2 is m, then the running time for GST is $O(n+m)$. Let the total number of CSSs be η. Then Algorithm 1 running time is the maximum of $(n+m)$ and η.

Lemma 3. *For n-length S_1 and m-length S_2, computing the η number of CSSs requires $O(\max\{n+m, \eta\})$ time.*

3.1.2 From CSS to DAG

From the previous step, all the CSS matches are given in $\mathcal{A}$. Now we can calculate the directed acyclic graph from the set of CSSs. In our DAG we need a starting and ending point to calculate the longest path. The starting

point $\mathcal{S}$ will precede all the CSSs in $\mathcal{A}$ and ending point $\mathcal{E}$ will succeed all matching CSSs. The paths from $\mathcal{S}$ to $\mathcal{E}$ in the DAG will represent all the maximal common sequences between S_1 and S_2. From the paths we can find the longest one and that will be the longest common sequence between S_1 and S_2.

In the DAG each node will represent a length-1 CSS string. According to Definition 1, each $v \in \mathcal{V}$ will be a node in the DAG. A link from node v_1 to v_2 presents that, $S_1[v_1.p_1] = S_2[v_1.p_2]$ is substring followed by $S_1[v_2.p_1] = S_2[v_2.p_2]$. And these substrings are chosen for those maximal common subsequences, that contains the link $v_1 \rightarrow v_2$. And obviously according to Definition 2, $v_1.p_1 \leq v_2.p_1$ and $v_1.p_2 \leq v_2.p_2$. Because LCS is an ordered list of substrings and we cannot choose a $v_i \in \mathcal{V}$ and then choose $v_j \in \mathcal{V}$ with $j < i$. Hence we can say that, there exists no cycle in the DAG. The main task in constructing the DAG is to set the links between the CSSs.

For the clarity of understanding how we get a directed acyclic graph DAG from the CSSs, here we represent the CSSs as a backbone structure on S_1 and S_2. Then using the previous example we explain how to construct the DAG. The backbone structure is purely conceptual, we do not construct the actual structure in our algorithms.

Backbone structure

For $S_1 = xabxa$ and $S_2 = babxba$ we can think of a structure $\mathcal{B}$ shown in Figure 3.2. On $\mathcal{B}$, S_1 and S_2 are shown as two straight lines. We can present the letters on S_1 and S_2 as circles. The links from one circle on S_1 to another circle on S_2 represents that there is a matching CSS from S_1 to S_2. Thus all the links represnts the CSSs in $\mathcal{A}$. We add the starting point $\mathcal{S}$ as a dotted line on the left of the structure and ending point $\mathcal{E}$ to the right of the structure.

The goal in using this backbone $\mathcal{B}$ is to represent and clearly understand which CSSs conflicts with each other and which does not. After choosing one CSS v, we cannot add a link from v to those CSSs that conflicts with v while we construct the directed acyclic graph. We need to find the CSSs that do not conflict with v and put a link on DAG from v to

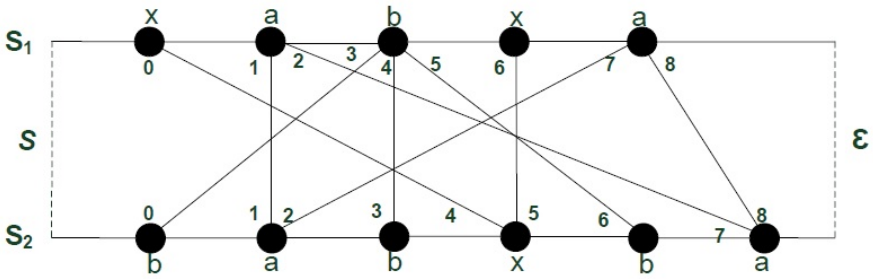

Figure 3.2: The backbone structure for $S_1 = xabxa$ and $S_2 = babxba$ with CSS serial numbers on S_1 and S_2.

those non-conflicting CSSs. By *conflict* we mean that two CSSs v_1 and v_2 crosses each other on the backbone. We can give a clear definition of conflict in terms of the CSS position (p_1, p_2) on S_1 and S_2 respectively.

Definition 4. ***Conflict****: Given two length-*1 *CSS, v_1 and v_2 and their positions in S_1 and S_2, if $v_1.p1 < v_2.p1$ and $v_1.p2 > v_2.p2$, or $v_1.p1 > v_2.p1$ and $v_1.p2 < v_2.p2$, or $v_1.p1 = v_2.p1$ and $v_1.p2 \neq v_2.p2$ or $v_1.p1 \neq v_2.p1$ and $v_1.p2 = v_2.p2$ then v_1 and v_2 conflicts with each other.*

For each matching CSS, we put a serial number on S_1 and S_2 based on their starting sequence, because multiple CSS can start from the same position (see Fig. 3.2). The numbers are shown on the backbone $\mathcal{B}$. We denote each CSS by a serial number pair (sequence in S_1, sequence in S_2)=(r_1, r_2) . Note that this is different from the starting positions of CSSs in the strings which we denoted as p_1 and p_2 before. We show this representation in the directed acyclic graph figure. Now we need to calculate the end block for each CSS.

End block

For each common substring, CSS on $\mathcal{B}$ we need to calculate the end block to find out the outgoing links on the directed acyclic graph (DAG). For a CSS v, the end block boundary of v consists of the closest CSS y_1 of v on S_1 that does not conflict with v and the closest CSS y_2 on S_2 that does not conflict

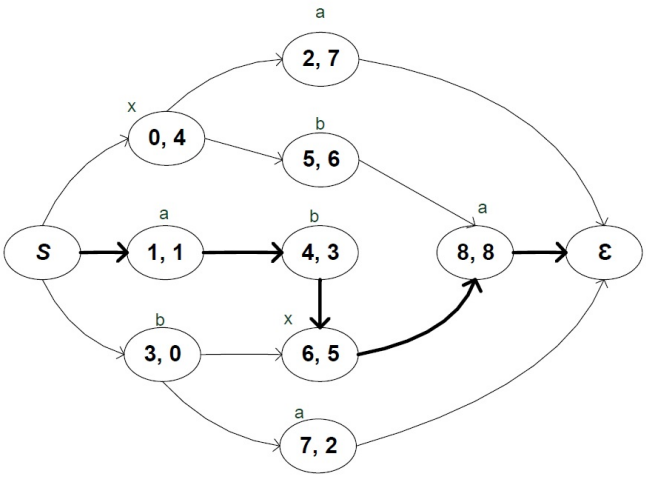

Figure 3.3: The directed acyclic graph from the backbone $\mathcal{B}$.

with v. y_1 and y_2 should conflict with each other, otherwise they are the same. y_1 and y_2 will have a link from v on the directed acyclic graph DAG. Now in the boundary of y_1 and y_2, which is bounded by $y_1.p1$ to $y_2.p1$ on S_1 and $y_2.p2$ to $y_1.p2$ on S_2, may contain other CSSs z that does not conflict with v, but conflicts with y_1 and y_2. Those z's will also have links from v on DAG. All these CSSs having a link from v on DAG build up the end block of v.

But we need to be very careful when choosing z's, because some z's will be parallel to each other on $\mathcal{B}$ but both conflicting y_1 and y_2. In those cases, we only need to consider the left z on $\mathcal{B}$, because the right one may have an incoming link from the left z. In that case, putting another link from v to the right z will be redundant. Other matching substrings outside the endblock need not to be considered because they will have link from the substrings in the end block.

In Figure 3.2 the starting CSS $\mathcal{S}$'s end block boundary consists of the CSSs $(0, 4)$ and $(3, 0)$ (represented with (r_1, r_2)). These have a link from $\mathcal{S}$ on the DAG. In the boundary, we find $(1, 1)$ which also has a link from $\mathcal{S}$. So the end block of $\mathcal{S}$ consists of $(0, 4), (3, 0)$, and $(1, 1)$. $\mathcal{S}$ has an outgoing link to all these three CSSs.

A1\A2	$\mathcal{S}$	0	1	2	3	4	5	6	7	8	$\mathcal{E}$
$\mathcal{S}$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
0	✓	×	×	×	×	✓	×	✓	✓	✓	✓
1	✓	×	✓	×	✓	×	✓	✓	×	✓	✓
2	✓	×	×	×	×	✓	×	×	✓	×	✓
3	✓	✓	×	✓	×	×	✓	×	×	✓	✓
4	✓	×	✓	×	✓	×	✓	×	×	✓	✓
5	✓	×	✓	×	×	✓	×	✓	×	✓	✓
6	✓	✓	✓	×	✓	×	✓	×	×	✓	✓
7	✓	✓	×	✓	×	×	×	×	×	×	✓
8	✓	✓	×	✓	✓	✓	✓	✓	×	✓	✓
$\mathcal{E}$	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Figure 3.4: The conflict table for the CSS from S_1 and S_2 with serial numbers on A_1 and A_2.

In Figure 3.3 the directed acyclic graph from the backbone $\mathcal{B}$ for S_1 and S_2 is shown. The longest path from $\mathcal{S}$ to $\mathcal{E}$ is shown with dark lines. The longest path is $[\mathcal{S}, (1,1), (4,3), (6,5), (8,8), \mathcal{E}]$. If we spell out the letters in the path, it is $abxa$, which is the longest common subsequence LCS between S_1 and S_2.

Computing the conflict table

In the DAG construction process, first we sort the list of CSSs in $\mathcal{A}$ w.r.t. the starting position in S_1 and number them serially as on backbone and store them in A_1. Then we do the same sorting on S_2 and number the CSSs based on their starting position in S_2 and store them in A_2. We have shown the numbering in Figure 3.2. Using A_1 and A_2 we calculate the conflict table $\mathcal{T}$.

Conflict table $\mathcal{T}$ contains information on whether a substring starting at r_1 in A_1 conflicts with a substing starting in r_2 in A_2. In Figure 3.4 we

show the conflict table for the CSSs from Figure 3.2. Each row represents the CSS serial number on A_1 or on S_1 on $\mathcal{B}$ and each column represents the CSS serial number on A_2 or on S_2 on $\mathcal{B}$. A cross mark (✗) represents that those CSSs conflict with each other, and a check mark (✓) represents that they do not conflict. The green boxes represent that those CSSs from S_1 and S_2 that are the same indentity. Hence they do not conflict with themselves. A conflict table is not a symmetric, because for a CSS the starting position in S_1 may not be same in S_2.

In Algorithm 2 we show how to calculate the conflict table. We have calculated the CSS list A_1 from n-length S_1 and A_2 from m-length S_2. So the conflict table can be constructed in $O(|A_1| \times |A_2|)$ time. A_1 and A_2 contains the sequentially numbered CSSs and are of same length, $|CSS| = \eta$.

Theorem 5. *For the n-length S_1 and m-length S_2, and the sorted set of CSSs A_1 and A_2, the conflict table $\mathcal{T}$ on A_1 and A_2 is constructed in $O(\eta^2)$ time.*

Algorithm 2 Compute the conflict table values from A_1 and A_2

```
1: Compute ConflictTable(A1, A2)
2: for each a1 in A1 do
3:     for each a2 in A2 do
4:         if a1 and a2 conflict then T[a1][a2] = ✗
5:         elseT[a1][a2] = ✓
6:         end if
7:     end for
8: end for
9: return T
```

Calculate end block

In section 3.1.2, we have described the end block. Here we give an algorithm to calculate the end block. This algorithm uses the help of the conflict table $\mathcal{T}$. For a CSS v, we can find the next CSS with which v is not conflicting, from the conflict table. First we find the sequence position of v on S_1 from A_1 and sequence position of v on S_2 from A_2. Then we go to that block on

$\mathcal{T}$. These blocks are marked with green on $\mathcal{T}$. Let y_1 on S_1 and y_2 on S_2 be the next CSSs that does not conflict with v. On $\mathcal{T}$, we go to the right of v, the first CSS that v does not conflict with is the y_2, if we go down, the first CSS that v does not conflict with is the y_1. Once we find y_1 and y_2, we store them in set EB.

To calculate the next non-conflicting CSS we use the function End Block Boundary. Here we give the CSS v and a boundary on A_1 or A_2. This boundary is defined by y_1 and y_2. Once we find y_1 and y_2 we only try to find next non-conflicting CSS in between $y_1.p_1$ to $y_2.p_1$ on S_1 and $y_2.p_2$ to $y_1.p_2$ on S_2. We try to find a new y_1 and a new y_2 in this boundary. If we find one we add it to the EB set. We search until $y_1 = y_2$ or y_1 and y_2 is null. We show the algorithms below. Algorithm 5 shows the DAG construction process. it uses Algorithms 2 and 4, Compute ConflictTable and Compute EndBlock. Algorithm 4 uses the Algorithm 3, Compute End Block Boundary.

Algorithm 3 Compute the end block boundary from v

```
1: Compute End Block Boundary(v, p1, p2, flag)
2: Map p1 to r1 and p2 to r2 using A1 and A2
3: if flag == false then
4:     i ← v's position in A1
5:     if r2 ≤ r1 then
6:         y ← null
7:     end if
8:     for r = r1 to r2 do
9:         if T[i][r] == ✓ then
10:            y ← get rth pos CSS from A2
11:            break;
12:        end if
13:    end for
14: else
15:    i ← v's position in A2
16:    if r2 ≤ r1 then
17:        y ← null
18:    end if
19:    for r = r1 to r2 do
20:        if T[r][i] == ✓ then
21:            y ← get rth pos CSS from A1
22:            break;
23:        end if
24:    end for
25: end if
26: return y
```

Algorithm 4 Compute end block of a CSS

Compute EndBlock(v)
$y_1 \leftarrow$ Compute End Block Boundary(v, $v.p_1 + 1$, n, true)
$y_2 \leftarrow$ Compute End Block Boundary(v, $v.p_2 + 1$, m, true)
EB $= \emptyset$
if y_1==y_2 **then** EB $\leftarrow EB \cup y_1$
else if y_1==null || y_2==null **then** break;
else
    EB $\leftarrow EB \cup y_1 \cup y_2$
    while (1) **do**
        $y_1 \leftarrow$ Compute End Block Boundary(v, $y_1.p_1 + 1$, $y_2.p_1 - 1$, true)
        $y_2 \leftarrow$ Compute End Block Boundary(v, $y_1.p_2 + 1$, $y_2.p_2 - 1$, true)
        if y_1==y_2 **then** EB $\leftarrow EB \cup y_1$
        else if y_1=null || y_2 == null **then** break;
        else
            EB $\leftarrow EB \cup y_1 \cup y_2$
        end if
    end while
end if
return EB

Algorithm 5 Constructing the directed acyclic graph

Construct DAG($\mathcal{A}$)
$A_1 \leftarrow$ Sort $\mathcal{A}$ according to starting position in S_1
$A_2 \leftarrow$ Sort $\mathcal{A}$ according to starting position in S_2
$\mathcal{T} \leftarrow$ Compute ConflictTable(A_1, A_2)
$DAG \leftarrow \emptyset$
for each a in $\mathcal{A}$ **do**
    EB = Compute EndBlock(a)
    for each e in EB **do**
        DAG.add($a \rightarrow e$)
    end for
end for
return DAG

3.1.3 From DAG to LCS

Each node in the DAG is a substring occuring in the given strings. Since our DAG has a single source $\mathcal{S}$, we start from the $\mathcal{S}$ and take different paths to reach the end $\mathcal{E}$. While traversing the graph we concatenate the substrings of the nodes we are traversing. Thus the length from start to end will be the length of the subsequence. So the longest path from start to end will give us our desired longest common subsequence between S_1 and S_2. This is equivalent to the reknowned longest path problem in a directed acyclic graph. In a general graph, the longest path problem in NP-hard. However there is linear time solution for directed acyclic graph via topological sorting [8].

3.1.4 Complexity Analysis

As indicated before, our LCS algorithm has three main steps. (i) From GST to CSS, (ii) From CSS to DAG, and (iii) From DAG to LCS. Here we present the overall time complexity of the algorithm.

(i) By Lemma 3, computing the η number of length-1 CSSs requires $O(\max\{n+m, \eta\})$ time.

(ii) Then we construct the DAG from the list of CSSs $a \in \mathcal{A}$ using `Construct DAG`. In the `Construct DAG` function, we initially sort the CSSs based on their starting positions in S_1 or S_2 and the we compute the conflict $\mathcal{T}$ in $O(\eta^2)$ time by Theorem 5. This is shown in Algorithm 2.

After computing $\mathcal{T}$, the `Construct DAG` iterates through every CSS a in $\mathcal{A}$ and calculates the end block for each a. The `Compute EndBlock` function computes finds all the nodes in DAG that has an incoming node from a. End block is bounded by substrings y_1 and y_2 described in Section 3.1.2. So the running time of end block is $O((y_2.p_1 - y_1.p_1) + (y_2.p_2 - y_1.p_2))$. But it might be the case that, $y_1.p_1 = 1$ and $y_2.p_1 = n$ and $y_1.p_2 = 1$ and $y_2.p_2 = m$. In that case, the `Compute EndBlock` function runs in $O(n+m)$ time.

Lemma 6. *For n-length S_1 and m-length S_2, computing the end block for each CSS requires $O(n+m)$ time.*

For a uniformly distributed string, we can more precisely bound the running time to compute the end block. As described in Section 3.1.2, y_1 and y_2 are the end block boundary of a. Since S_1 and S_2 are uniformly distributed, that means for a symbol σ at position i in S_1, we can find every $\sigma \in \Sigma$ in the range of positions $(i-\Delta...i+\Delta)$ in S_2 with $\Delta \in O(|\Sigma|)$, then $(y_2.p_1 - y_1.p_1) \in O(|\Sigma|)$ and $(y_2.p_2 - y_1.p_2) \in O(|\Sigma|)$. Which follows, $n, m \in O(|\Sigma|)$

Lemma 7. *Given S_1 and S_2 with symbols uniformly drawn from alphabet Σ, the* `Compute EndBlock` *function requires $O(|\Sigma|)$ time.*

So, the overall `Construct DAG` time follows.

Theorem 8. *Given $|\mathcal{A}| = \eta$, $|S_1| = n$ and $|S_2| = m$, the* `Construct DAG` *function requires $O(\max\{\eta^2, \eta \times (n+m)\})$ time in the worst case and $O(\max\{\eta^2, \eta \times |\Sigma|\})$ time on average.*

(iii) If the number of vertices in the DAG is V and number of edges E, finding longest path in $DAG = G(V, E)$ is linear to the DAG size, which is $O(|V| + |E|)$.

The computation of CSS from GST in Step (i) and calculating LCS from DAG in Step (iii) does not add to the complexity of constructing DAG from CSSs. So the overall complexity of finding the LCS is the complexity of constructing the DAG.

Theorem 9. *Given n-length S_1 and m-length S_2, the LCS can be computed in $O(\max\{\eta^2, \eta \times (n+m)\})$ time in the worst case and $O(\max\{\eta^2, \eta \times |\Sigma|\})$ time on average.*

3.2 Towards Variable length CSS

In the previous section, we have seen how to compute the LCS for length-1 CSSs from the strings S_1 and S_2. From the generalized suffix tree we have

only extracted length-1 common substrings. But we can get all the variable length common substrings from the GST. In this section, we describe how to approach calculating LCS from variable length CSSs.

3.2.1 Determining CSSs of length ≥ 2

In Section 3.1.1 we have described how to build a generalized suffix tree for the string $S = S_1\$ \circ S_2\#$. We can use the same tree to calculate all the CSSs with $|CSS| \geq 2$ between S_1 and S_2. For length-1 CSS we went to only depth-1 of the tree. Now for length-k CSSs we can go down to depth-k of the tree, and calculate the suffix numbers descending that node. We do a pre-order traversal of the tree and collect the suffix numbers at each internal node. Then we match the common substrings from S_1 and S_2 and add them in $\mathcal{A}$. This process takes the same time as length-1 CSSs (See Lemma 3) $O(\max\{(n+m), \eta\})$.

3.2.2 Handling overlaps

Now that we have $|CSS| \geq 2$, some pairs of common substrings will overlap with each other. When using length-1 CSS, we describe how to handle them when they cross with each other or conflict with each other. We used (p_1, p_2) to represent the position of a CSS v on S_1 and S_2 respectively. Now that our $|CSS| \geq 2$, we need four points to represent them. The starting and ending points on S_1 and the starting and ending position on S_2. We denote them by $(p11, p12, p21, p22)$ respectively. In Figure 3.5 we show an example of overlapping substrings. From the figure we can see that B starts before A ends on S_1.

Definition 10. ***Overlap****: Given two strings, S_1 and S_2, and two CSSs A and B between S_1 and S_2. Assuming A is on left of B on S_1, we say A and B overlaps if the right edge of A conflicts with the left edge of B, or $A.p12 \geq B.p11$ or $A.p22 \geq B.p21$.*

If two CSS overlaps we divide them in a way so that the overlapped portion becomes an independent CSS, and the other parts become separated

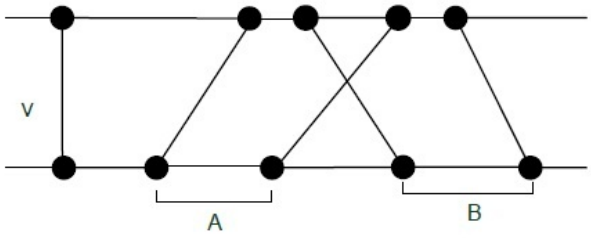

Figure 3.5: Overlapping of substrings A and B.

from the overlapped portion. Based on A and B's conflicting edge we can divide the overlaps in five categories. Here we describe the types and how to divide them into multiple CSS:

Left-overlap

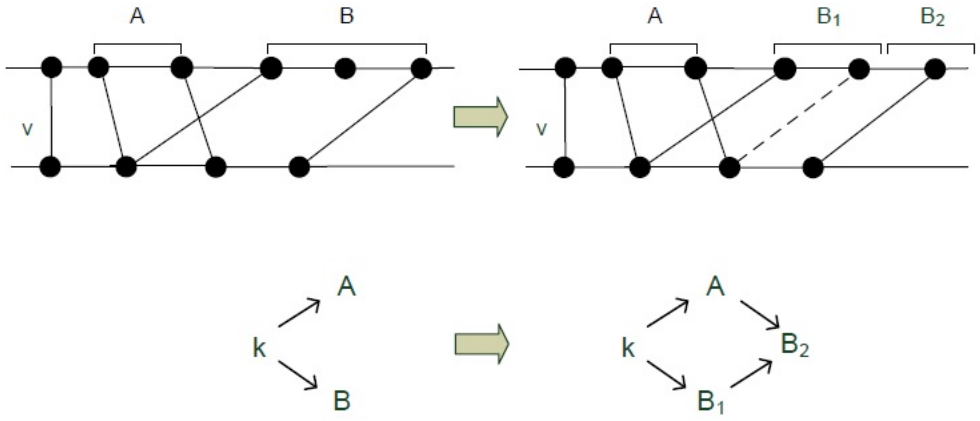

Figure 3.6: Division of left-overlap.

In Figure 3.6 we can see that the left side of B overlaps with A. We call this type of overlap left-overlap when a CSS overlaps with the left side of another CSS. In this case we can divide B into two parts B_1 and B_2 so that $B_1 = A$. So now from 3.6(b) we can see that we have 3 CSSs, A, B_1, B_2. A and B_1 conflict with each other but there is no overlap. Now we know how to solve them for DAG using our LCS algorithm. In 3.6(c) we have shown that part of the DAG.

Right-overlap

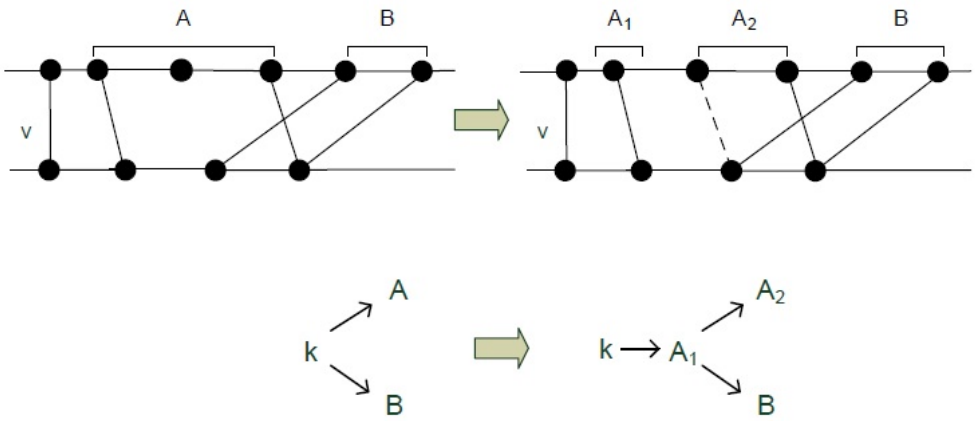

Figure 3.7: Division of right-overlap.

In Figure 3.7 we can see that the right side of A overlaps with B. We call this type of overlap right-overlap when a CSS overlaps with the right side of another CSS. In this case we can divide A into two parts A_1 and A_2 so that $A_2 = B$. So now from Figure 3.7(b) we have 3 CSSs, A_1, A_2, B. A_2 and B conflicts with each other but there is no overlap. The DAG is shown in Figure 3.7(c).

Mid-overlap (no crossing)

In Figure 3.8 we can see that the right side of A overlaps with the left side of B. We call this type of overlap mid-overlap. In this case we can divide A into two parts A_1 and A_2 and B into two parts B_1 and B_2. After dividing we can see from Figure 3.8(b) that, $A_2 = B_1$. So now we have 4 CSSs, A_1, A_2, B_1, B_2. A_2 and B_1 conflicts with each other but there is no overlap. The DAG is shown in Figure 3.8(c).

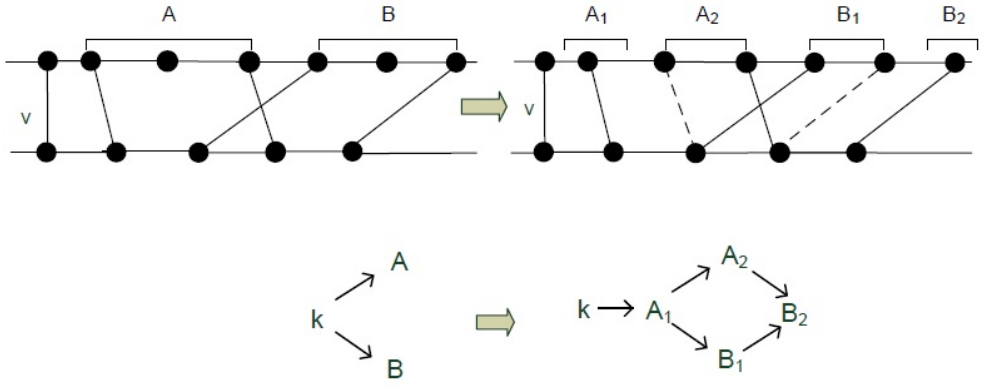

Figure 3.8: Division of mid-overlap.

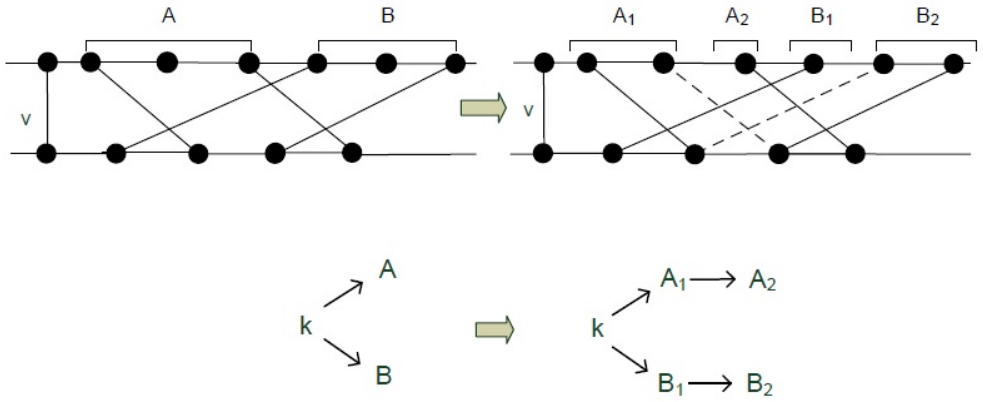

Figure 3.9: Division of mid-overlap with crossing.

Mid-overlap (with crossing)

In Figure 3.9 we can see that the left side of A overlaps with the right side of B. We call this type of overlap mid-overlap with crossing. In this case we can again divide A into two parts A_1 and A_2 and B into two parts B_1 and B_2. After dividing we can see from Figure 3.9(b) that, $A_1 = B_2$. So now we have 4 CSSs, A_1, A_2, B_1, B_2. A_1 and A_2 conflicts with B_1 and B_2 but there is no overlap. The DAG is shown in Figure 3.9(c).

Mid-overlap (with containment)

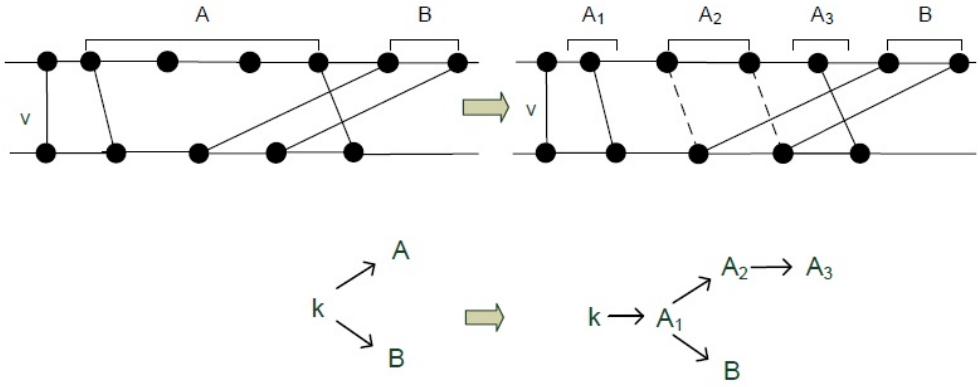

Figure 3.10: Division of mid-overlap with containment.

In Figure 3.10 we can see that B is inside of A on S_2 but outside on S_1. We call this type of overlap mid-overlap with containment. In this case we can divide A into three parts A_1, A_2 and A_3. After dividing we can see from Figure 3.10(b) that, $A_2 = B$. So now we have 4 CSSs, A_1, A_2, A_3, B. A_2 and A_3 conflicts with B but there is no overlap remaining. The DAG is shown in Figure 3.10(c).

Using the approach mentioned above we can remove the overlaps between common substrings, but as mentioned before conflict will exists. After removing all the overlaps from the CSSs. Then our algorithms described in Section 3.1 can be used to compute the longest common subsequence. In that case we can use a weighted directed acyclic graph where the weight will be the length of the CSS.

3.3 Applications of LCS

Comparing strings is a general and basic problem in computer science, because strings are everywhere. In Section 1.1 we have mentioned some applications of longest commmon subsequence between two strings. In general it is used in similarity measurement, in particular file comparison, pattern

recognition in images, etc. LCS is a step in general compression, because it compares two strings and gives us the similarity. So when comparing two files we can get the similarity between them using LCS. So it is clear that if we have a file for reference, we can compare the other file to compress and based on the similarity we can compress and store the dissimilarity. This is why LCS is a hallmark of reference based approaches to compressing genome resequencing data and has been used in some recent algorithms [29, 35]. Volumes of genomic data are being generated each day and we know that human genes have a significant similarity between individuals, so if we can extract the similarity, we can easily handle and store the dissimilarity. In Section 2.4, we have described what is genome data compression and some recent works on genome data compression using LCS [29, 35]. In this section, we describe some new compression techniques using and inspired by LCS.

3.3.1 Compression Method 1

Assume we have two sequences S_1 and S_2. The length of S_1 is n and the length of S_2 is m. Let S_1 be the target sequence and S_2 be the reference sequence. The reference sequence can be stored as it is, and target sequence will be compressed comparing with the reference based on their similarity. This is what is done in most reference based genomic data compression methods. The LCS between S_1 and S_2 can be found using our LCS algorithm. Or in general, any LCS algorithm could be used.

We know that LCS between two strings may not contain consecutive positions. It is the sequence of common substrings CSSs. Let $\mathcal{M} = \{m_1, ..., m_M\}$ be an LCS from S_1 and S_2, where m_i is a common substring. Here we do not limit ourselves to length-1 CSS, rather we can use length-k CSS, $k \leq n$. Denote m_i's starting poistion in S_1 as $m_i.p_1$, and in S_2 as $m_i.p_2$ as before (see Section 3.1.1) and the length as $|m_i| = l$. So each CSS can be represented by a triplet *(pos in target, pos in reference, length)*=(p_1, p_2, l). So the compression result will have two files, one for the *triplets* and another to store the *symbols* not in the LCS. Then Huffman coding or arithmetic coding can be applied to encode the files.

To decompress the target from the *triplets* and *symbols* files, we

need to start with the first target CSS position p_1 from the triplet file. If it starts from the first position, then we copy the CSS from the reference using the reference position p_2 and length l. If the CSS does not start from first position, we copy symbols $[1 \dots p_1]$ from the *symbols* file. Thus we proceed by copying matchings either from reference file or from the symbols file and recreate the original target file.

We describe this method with an example. Let $S_1 = xabxa$ and $S_2 = babxba$. The longest common subsequence between S_1 and S_2 is $abxa$, and the CSSs in this LCS are $\{abx, a\}$. The *triplets* file will be $\{(2, 2, 3), (5, 6, 1)\}$ and the *symbols* file will contain $\{x\}$. So when decompressing we take $\{x\}$ first, then take length-3 CSS from starting position 2 from reference S_2 $\{abx\}$ and add it with $\{x\}$. Then the current target will be $\{xabx\}$. Then take length-1 CSS from starting position 6 from reference S_2 $\{a\}$ and add it with current target. Then the final target file will be $\{xabxa\}$.

3.3.2 Compression Method 2

Here we describe another compression technique which is inspired by the LCS. We do not directly use the longest common subsequence LCS here, but we use the components the of LCS, the CSSs. Also here length-k CSS are considered. Let $Z = S_2 \circ S_1$. Now we can calculate the starting position of CSSs based on the length of Z. Suppose a CSS m starts at i on S_1 (target) part of Z and starts at h on S_2 (reference) part of Z. Obviously we know that $h < i$ and $|S_2| + 1 \leq i \leq |Z|$ and $1 \leq h \leq |S_2|$. Let the length of the CSS at these poisiton $|m| = k$. These k's and positions h's can be computed using the longest previous factor data structure (LPF) and POS in linear time [5, 6, 10]. Now we can compress the CSS starting at position i with reference at position h in the reference using LPF data structure. This matches the dictionary compression [11].

To compress target S_1 with respect to reference S_2, we scan the LPF and POS on Z in a left to right fashion. If $LPF[i] \geq k$, we encode the CSS with triplets *(pos in target, pos in reference, length)*$=(i - |S_2|, POS[i], LPF[i])$. Where $LPF[i] < k$ we simply write out the symbols. We can write the triplets and symbols as bytes. The resulting file will be

a binary file which can be further compressed using standard compression methods. To decompress the target we again scan left to right. We start with the *triplets* file and we decompress with the same method described in Compression Method 1(3.3.1).

3.3.3 Compression Results

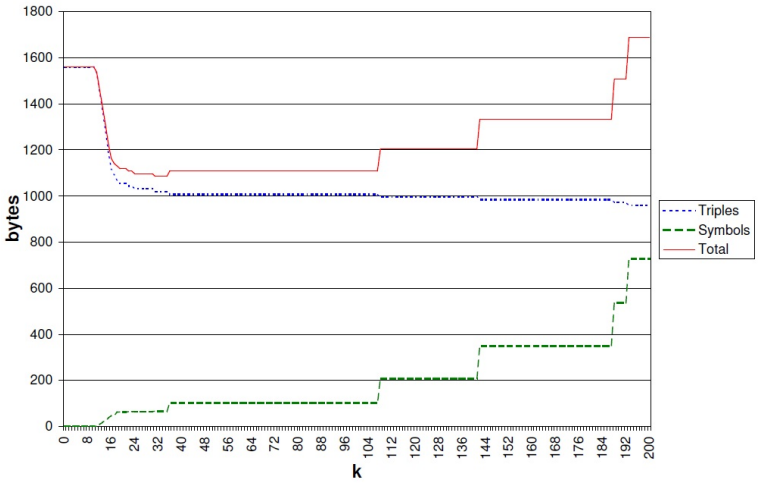

Figure 3.11: Size of TAIR9 after compression vs k.

We implemented Method 1 and tested on the performance on *Arabidopsis thaliana* genome chromosomes, TAIR9 (target) with respect to TAIR8 (reference) like Wang et al. [29,35]. The compression results were eligibly better than those from tradition compression schemes (LZMA2, BZip2, PPMd etc.). But it was not compatible with the recently published compression result from [29,35]. Method 2 have been implemented for the recent paper we submitted to IEEE BIBM, 2015 [7]. We have tested it on the same dataset (chromosome $1-5$). We varied the length k described in section 3.3.2 over $0-200$ to find the best k for chromosome 1, and looked at the compression result closely. On Figure 3.11 the results of variation of k vs size (in bytes) after the compression is plotted. From the plot we can see that for lower k values (close to 0) and for higher k value (close to 200), the compression result is not so good. We get the best k between $31-35$ when the compression

Table 3.1: Results (in bytes) for compressing the TAIR9 chromosome (target), with respect to TAIR8 (reference) with $k = 31$.

Chromosome (C)		Basic Compression			c_1 =Method_2($C, 31$))				GRS	GReEN
#	size	`lzma2`	`ppmd`	`bzip2`	c_1	`lzma2`	`ppmd`	`bzip2`	[35]	[29]
1	30,427,671	7,175,593	7,263,093	7,994,719	1,086	963	1,037	1,227	**715**	1,551
2	19,698,289	4,701,866	4,756,604	5,264,059	504	584	605	720	**385**	937
3	23,459,830	5,554,174	5,660,957	6,242,349	**746**	759	803	947	2,989	1,097
4	18,585,056	4,362,107	4,475,662	4,911,401	4,555	2,507	3,156	3,580	**1,951**	2,356
5	26,975,502	6,409,741	6,485,115	7,140,759	**433**	502	520	613	604	618
Total	119,146,348	28,203,481	28,641,431	31,553,287	7,324	**5,315**	6,121	7,087	6,644	6,559

result is the lowest. So we fixed $k = 31$ and tested the compression method on the chromosome dataset.

Table 3.1 shows the results for the compression of TAIR9 chromosomes $(1 - 5)$, with $k = 31$. Column 1 shows the chromosome numbers $(1-5)$ from the dataset. Column 2 is the size of the TAIR9 chromosome files. Column 10 and 11 shows the results from the existing best results for compression of these dataset [29,35]. Those methods are described in Section 2.4. Column $3, 4, 5$ shows the results using the basic compression schemes on the dataset. We have used $7zip$ software and $LZMA2$, $PPMd$ and $BZip2$ compression methods. Column 6 shows the result using our compression method 2. We have ran the basic compression methods on top of our compression and those results are shown in column $7, 8$, and 9.

We can see that, all of the $c_1 = Method\_2(C, 31)$ results are competitive with the GRS and GReEN systems, except for chromosome 4. We can explain it in terms of average CSS length. Because if the CSS length is bigger, the reference and the target has more matching portions and our *symbols* and *triplets* files are smaller. Chromosome 4 has the smallest average CSS length of about $326K$, followed by chromosome 3 (≈455K), chromosome 1 (≈458K), chromosome 2 (≈510K), and chromosome 5 (≈1,704K). However, we are able to further compress the files after running our compression method via standard compression schemes (in 7-zip) to make up for chromosome 4. Chromosome 3 and 5 show the best results, we were able to compress them in less bytes than needed by the GRS and GReEN. For each chromosome best results are shown in bold.

Chapter 4

Conclusion

4.1 Summary

We proposed a novel method to compute the longest common subsequence (LCS). The contruction of directed acyclic graph DAG via the generalized suffix tree (GST) is a creative approach to solving the LCS problem. Our algorithm gives solution for common substrings (CSS) of length-1. We have compiled all the CSSs and presented them on a conceptual backbone structure. This backbone concept clears the main idea of our approach. We sort the CSSs according to their starting position in S_1 and S_2 and then calculate the *conflict table*. The pre-computed *conflict table* holds the core information needed during the run-time of the algorithm. The we compute the $end - block$ by ruling out the redundant CSS links on the directed acyclic graph (DAG). The DAG gives our desired solution in linear time. Our overall algorithm runs in $O(\max\{\eta^2, \eta \times (n+m)\})$ time in the worst case and $O(\max\{\eta^2, \eta \times |\Sigma|\})$ time on average. The average case complexity has improved result over other LCS algorithms that are based on dynamic programming.

We have also investigated genomic data compression. It attracted our attention due to the contemporary research results related to LCS and genomic data compression. We have proposed and narrated two new ref-

erenced based methodology for compression of biological data. The first method directly uses the LCS and the second one uses CSS and LPF data structure (see Section 3.3.2). Results of Method 2 on *Arabidopsis thaliana* genome data is presented and compared with other compression techniques.

Before this thesis, longest common subsequence did not have any direct relation with generalized suffix tree, although GST is a widely used data structure for string matching problems. We took the challenge and successfully made the connection between the LCS and the GST via the DAG, which is our biggest achievement in this work.

4.2 Future work

Our main target was to find an algorithm that uses all the common substrings of any length. Now we have fomulated our algorithm for length-1 CSSs. In Section 3.2, we have described how to derive length-k CSSs and a divide and conquer approach to make them usable in our algorithm. One direction for future work would be to identify techniques to improve our algorithm to make use of CSSs of various lengths. If we could use of various length CSS then number of CSSs η will be smaller, because multiple length-1 CSSs will be merged to create a single CSS of length-k. If η becomes reasonably smaller our algorithm should run faster.

Another direction for future work would be to extend the LCS for multiple strings. We know that for arbitrary number of sequences this is NP-hard. But for a fixed number of sequences we could have a solution for longest common subsequence between them. And we can again approach it via generalized suffix tree, because we know how to construct GST for multiple strings. Also we plan to improve our Compression Method 1 for better compression and different genomic resequencing datasets could be used to run the compression algorithm.

Bibliography

[1] John Aach, Martha Bulyk, George Church, Jason Comander, Adnan Derti, and Jay Shendure. Computational comparison of two draft sequences of the human genome. *Nature*, 26(1):5–14, 2001.

[2] Donald Adjeroh, Timothy Bell, and Amar Mukherjee. *The Burrows-Wheeler Transform: Data Compression, Suffix Arrays, and Pattern Matching*. Springer Publishing Company, Incorporated, 1 edition, 2008.

[3] Donald Adjeroh and Fei Nan. On compressibility of protein sequences. In *DCC*, pages 422–434. IEEE Computer Society, 2006.

[4] A. Apostolico and R. Giancarlo. The Boyer-Moore-Galil string searching strategies revisited. *SIAM Journal of Computing*, 15(1):98–105, 1986.

[5] Richard Beal and Donald Adjeroh. Parameterized longest previous factor. *Theoretical Computer Science*, 437:21 – 34, 2012.

[6] Richard Beal and Donald Adjeroh. Variations of the parameterized longest previous factor. *Journal of Discrete Algorithms*, 16:129 – 150, 2012. Selected papers from the 22nd International Workshop on Combinatorial Algorithms (IWOCA 2011).

[7] Richard Beal, Tazin Afrin, Aliya Farheen, and Don Adjeroh. A new algorithm for the *lcs* problem with application in compressing genome resequencing data. In *IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*. IEEE, 2015, [Submitted].

[8] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill, 2nd edition, 2001.

[9] Anthony J. Cox, Markus J. Bauer, Tobias Jakobi, and Giovanna Rosone. Large-scale compression of genomic sequence databases with the burrows-wheeler transform. *Bioinformatics*, 28(11):1415–1419, 2012.

[10] Maxime Crochemore and Lucian Ilie. Computing longest previous factor in linear time and applications. *Information Processing Letters*, 106(2):75 – 80, 2008.

[11] Maxime Crochemore, Lucian Ilie, and W. F. Smyth. A simple algorithm for computing the lempel ziv factorization. In *Proceedings of the Data Compression Conference*, DCC '08, pages 482–488, Washington, DC, USA, 2008. IEEE Computer Society.

[12] Cees Elzinga, Sven Rahmann, and Hui Wang. Algorithms for subsequence combinatorics. *Theor. Comput. Sci.*, 409(3):394–404, Dec 2008.

[13] Cochrane G Fritz M, Leinonen R and Birney E. Efficient storage of high throughput dna sequencing data using reference-based compression. In *Proceedings of the Conference on Data Compression*, Epub 2011, pages 734–40. Cold Spring Harbor Laboratory Press, 2011.

[14] Raffaele Giancarlo, Davide Scaturro, and Filippo Utro. Textual data compression in computational biology: a synopsis. *Bioinformatics*, 25(13):1575–1586, 2009.

[15] Raffaele Giancarlo, Davide Scaturro, and Filippo Utro. Textual data compression in computational biology: Algorithmic techniques. *Computer Science Review*, 6(1):1–25, 2012.

[16] Dan Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge Univ. Press.

[17] Faraz Hach, Ibrahim Numanagic, Can Alkan, and Suleyman Cenk Sahinalp. Scalce: boosting sequence compression algorithms using locally consistent encoding. *Bioinformatics*, 28(23):3051–3057, 2012.

[18] D.S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, 1975.

[19] James W. Hunt and Thomas G. Szymanski. A fast algorithm for computing longest subsequences. *Commun. ACM*, 20(5):350–353, 1977.

[20] Guy Jacobson and Kiem-Phong Vo. Heaviest increasing common subsequence problems. In *Proceedings of the Third Annual Symposium on Combinatorial Pattern Matching*, CPM '92, pages 52–66, London, UK, 1992. Springer-Verlag.

[21] Chih-En Kuo, Yue-Li Wang, Jia-Jie Liu, and Ming-Tat Ko. Resequencing a set of strings based on a target string. *Algorithmica*, 72(2):430–449, June 2015.

[22] Shanika Kuruppu, Simon J. Puglisi, and Justin Zobel. Optimized relative lempel-ziv compression of genomes. In *Thirty-Fourth Australasian Computer Science Conference, ACSC 2011, Perth, Australia, January 2011*, pages 91–98, 2011.

[23] Zhiwei Lin, Hui Wang, and Sally I. McClean. A multidimensional sequence approach to measuring tree similarity. *IEEE Trans. Knowl. Data Eng.*, 24(2):197–208, 2012.

[24] David Maier. The complexity of some problems on subsequences and supersequences. *J. ACM*, 25(2):322–336, April 1978.

[25] Edward M McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM (JACM)*, 23(2):262–272, 1976.

[26] Eugene W. Myers. An $O(ND)$ difference algorithm and its variations. *Algorithmica*, 1(2):251–266, 1986.

[27] Craig G. Nevill-Manning and Ian H. Witten. Protein is incompressible. In *Proceedings of the Conference on Data Compression*, DCC '99, pages 257–. IEEE Computer, 1999.

[28] P. A. Pevzner and M. S. Waterman. A fast filtration algorithm for the substring matching problem. *LNCS 684, Combinatorial Pattern Matching*, pages 197–214, 1993.

[29] Armando J. Pinho, Diogo Pratas, and Sara P. Garcia. Green: a tool for efficient compression of genome resequencing data. *Nucleic Acids Research*, 2011.

[30] Temple F. Smith and Michael S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[31] E. Ukkonen. Algorithms for approximate string matching. *Inform and Control*, 64:100–118, 1985.

[32] Sebastian Wandelt, Marc Bux, and Ulf Leser. Trends in genome compression. *Current Bioinformatics*, 9(3):315 – 326, 2014.

[33] Sebastian Wandelt and Ulf Leser. Fresco: Referential compression of highly similar sequences. *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, 10(5):1275–1288, September 2013.

[34] Sebastian Wandelt, Johannes Starlinger, Marc Bux, and Ulf Leser. Rcsi: Scalable similarity search in thousand(s) of genomes. *Proc. VLDB Endow.*, 6(13):1534–1545, August 2013.

[35] Congmao Wang and Dabing Zhang. A novel compression tool for efficient storage of genome resequencing data. *Nucleic Acids Res.*, 39(4), 2011.

[36] Hui Wang. All common subsequences. In *Proceedings of the 20th International Joint Conference on Artifical Intelligence*, IJCAI'07, pages 635–640, 2007.

[37] Peter Weiner. Linear pattern matching algorithms. In *Switching and Automata Theory, 1973. SWAT'08. IEEE Conference Record of 14th Annual Symposium on*, pages 1–11. IEEE, 1973.

[38] Jiaoyun Yang, Yun Xu, Yi Shang, and Guoliang Chen. A space-bounded anytime algorithm for the multiple longest common subsequence problem. *IEEE Trans. Knowl. Data Eng.*, 26(11):2599–2609, 2014.