5-4-2018

Novel Algorithms for Some Fundamental Big Data Problems

Abdullah-Al Mamun

University of Connecticut, abdullah-al.mamun@uconn.edu

Follow this and additional works at: https://opencommons.uconn.edu/dissertations

Recommended Citation
https://opencommons.uconn.edu/dissertations/1812
In this digital era data sets are growing rapidly. Storing, processing, and analyzing large volume of data require efficient techniques. These techniques deal with big data problems by providing time efficient methods, effective external memory algorithms, parallel and high performance solutions, and so on. This thesis studies three important areas of big data problems and presents state of the art approaches to address them.

The first part of this thesis discusses the $k$-mer counting problem. A massive number of bioinformatics applications require counting of $k$-length substrings in genetically important long strings. Genome assembly, repeat detection, multiple sequence alignment, error detection, and many other related applications use a $k$-mer counter as a building block. Very fast and efficient algorithms are necessary to count $k$-mers in large data sets to be useful in such applications. We propose a novel trie-based algorithm for this $k$-mer counting problem.

In the second part, we present algorithms for the record linkage problems. Integrating data from multiple sources is a crucial and challenging problem. Here we have come up with efficient sequential and parallel algorithms for this problem which can handle any number of datasets. Our methods employ single linkage as well as complete linkage hierarchical clustering to address this problem.

The last part explains three problems with algorithmic challenges. The first one is the minimum spanning tree problem. Finding minimum spanning trees (MST) in various types of networks is a well-studied problem in theory and practical applications. We have devised a very efficient algorithm which combines ideas from randomized selection, Kruskal’s algorithm and Prim’s algorithm. The second problem is higher order spectra analysis of nonlinear time series. It has applications in biomedical signal processing, communications, geophysics, speech processing, etc. We address this problem by providing space and time
efficient sequential and parallel algorithms. The third problem is the closest $l$-mers problem. Algorithms for finding the closest $l$-mers have been used in solving the $(l, d)$-motif search problem. We describe exact as well as very fast approximate algorithms for computing a group of three $l$-mers having the minimum combined distance among all possible such combinations.

Source codes related to this thesis are available at [https://github.com/abdullah009](https://github.com/abdullah009).
Novel Algorithms for Some Fundamental Big Data Problems

Abdullah-Al Mamun

B.Sc., Bangladesh University of Engineering and Technology, 2009

A Dissertation
Submitted in Partial Fulfillment of the
Requirements for the Degree of
Doctor of Philosophy
at the
University of Connecticut

2018
Copyright by
Abdullah-Al Mamun

2018
Novel Algorithms for Some Fundamental Big Data Problems

Presented by
Abdullah-Al Mamun

Major Advisor
Sanguthevar Rajasekaran

Associate Advisor
Ion Mandoiu

Associate Advisor
Reda Ammar

University of Connecticut
2018
Acknowledgement

All praise is due to Allah, the Lord of the Worlds, the Beneficent, the Merciful.

I would like to express my sincerest gratitude to my major advisor Professor Sanguthervar Rajasekaran. He contributed in my research by coming up with magnificent ideas and providing insightful suggestions and feedback. I am grateful to him for his continuous support and encouragement.

I would like to thank Professor Ion Mandoiu and Professor Reda Ammar for advising me throughout my PhD. Their support, helpful comments, and research directions helped me to complete my thesis.

I would like to acknowledge all of my teachers, colleagues, and friends who were beside me with their wisdom, assistance, and confidence.

I would like to show my love to my elder brother Abu Ali Md Shazzad Hossain and my sisters. I recognize my brother's contributions in each and every achievement of my academic path. My parents deserve my deep feeling of respect for their unconditional love and endless support in every step of my life.

This thesis includes [90, 89, 88, 87, 91, 13] publications.
## Contents

### 1 k-mer Counting Problem

1. **KCMBT: A k-mer counter based on multiple burst trees**
   - 1.1 Introduction .................................................. 2
   - 1.2 Related Works ............................................... 3
   - 1.3 Methods ...................................................... 4
     - 1.3.1 Burst tries ................................................. 5
     - 1.3.2 Compact k-mers ............................................. 6
     - 1.3.3 \((k + x)\)-mers ........................................... 6
     - 1.3.4 Our Algorithm ............................................. 7
   - 1.4 Results ..................................................... 9
   - 1.5 Discussion ................................................ 19

### II Record Linkage Problem

2. **Efficient sequential and parallel algorithms for record linkage**
   - 2.1 Introduction .................................................. 22
   - 2.2 Background and Significance ................................ 23
   - 2.3 Methods ...................................................... 24
     - 2.3.1 Previous Methods ........................................ 24
     - 2.3.2 Our Approaches ......................................... 26
   - 2.4 Results ..................................................... 31
     - 2.4.1 Results on Simulated Data for The Sequential Algorithm ............................................. 32
     - 2.4.2 Results on Real Data for The Sequential Algorithm ..................................................... 36
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.4.3</td>
<td>Results on Simulated Data for The Parallel Algorithm</td>
<td>37</td>
</tr>
<tr>
<td>2.5</td>
<td>Discussion</td>
<td>39</td>
</tr>
<tr>
<td>3</td>
<td>Efficient record linkage algorithms using complete linkage clustering</td>
<td>41</td>
</tr>
<tr>
<td>3.1</td>
<td>Introduction</td>
<td>41</td>
</tr>
<tr>
<td>3.2</td>
<td>Background and Significance</td>
<td>42</td>
</tr>
<tr>
<td>3.3</td>
<td>Related Works</td>
<td>43</td>
</tr>
<tr>
<td>3.4</td>
<td>Methods</td>
<td>44</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Sequential Algorithm</td>
<td>45</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Analysis</td>
<td>50</td>
</tr>
<tr>
<td>3.4.3</td>
<td>Parallel Algorithm</td>
<td>51</td>
</tr>
<tr>
<td>3.4.4</td>
<td>Analysis</td>
<td>53</td>
</tr>
<tr>
<td>3.5</td>
<td>Results</td>
<td>54</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Generation of Simulated Data Sets</td>
<td>55</td>
</tr>
<tr>
<td>3.5.2</td>
<td>Sequential Algorithm</td>
<td>55</td>
</tr>
<tr>
<td>3.5.3</td>
<td>Parallel Algorithm</td>
<td>62</td>
</tr>
<tr>
<td>3.6</td>
<td>Discussion</td>
<td>64</td>
</tr>
<tr>
<td>4</td>
<td>RLT-S: A web system for record linkage</td>
<td>66</td>
</tr>
<tr>
<td>4.1</td>
<td>Introduction</td>
<td>66</td>
</tr>
<tr>
<td>4.2</td>
<td>Implementation</td>
<td>67</td>
</tr>
<tr>
<td>4.3</td>
<td>Results and Discussion</td>
<td>68</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Input Data Sets and Configurations</td>
<td>69</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Linkage Parameters</td>
<td>71</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Output</td>
<td>73</td>
</tr>
<tr>
<td>4.3.4</td>
<td>Submissions History</td>
<td>74</td>
</tr>
<tr>
<td>4.3.5</td>
<td>Feedback</td>
<td>74</td>
</tr>
<tr>
<td>4.4</td>
<td>Comparisons</td>
<td>75</td>
</tr>
</tbody>
</table>
III Problems with Algorithmic Challenges

5 An efficient minimum spanning tree algorithm

5.1 Introduction .................................................. 78
5.2 Related Studies .............................................. 79
5.3 Methods ....................................................... 80
  5.3.1 Filter-Kruskal Minimum Spanning Tree Algorithm ......... 80
  5.3.2 Edge Pruned Minimum Spanning Tree (EPMST) Algorithm .. 81
  5.3.3 Analysis of Algorithm EPMST ............................. 82
  5.3.4 Parallel EPMST algorithm ............................... 83
5.4 Experiments .................................................. 83

6 Efficient sequential and parallel algorithms for estimating higher order spectra

6.1 Introduction .................................................. 89
6.2 A Better Algorithm for the Direct Method ...................... 93
  6.2.1 Computing window sums .................................. 93
  6.2.2 Direct Method for Bispectrum ............................ 93
6.3 Parallel Models and Preliminaries ................................ 94
  6.3.1 Window sums on the PRAM ............................... 95
  6.3.2 Direct method for bispectrum on a PRAM .................. 96
6.4 Experimental Results ........................................... 98
  6.4.1 Test Platform ............................................. 98
  6.4.2 Run Time and Memory Comparisons ....................... 99
  6.4.3 Multi-core Parallel Approach Evaluation ................. 100
  6.4.4 Summary of Experiments ................................ 101

7 Efficient algorithms for finding the closest l-mers in biological data

7.1 Introduction .................................................. 103
7.2 Background Knowledge ........................................ 106
  7.2.1 The O(n²) Time Algorithm of [114] ..................... 106
2.5 Results on synthetic data using proportional threshold (t=0.1, Y axis denotes time in seconds; X axis corresponds to number of records in thousands) .................................................. 36

2.6 Analysis of results on synthetic data using PRLA (Y axis denotes time in seconds; X axis corresponds to number of processors) .................................................. 38

2.7 Results on simulated data (for 6 million and 9 million records, Y axis denotes time in seconds; X axis corresponds to number of processors) ................. 38

2.8 Speed up (for 6 million and 9 million records, Y axis denotes speed up; X axis corresponds to number of processors) ................................. 39

3.1 A flow chart describing all steps involved in RLA-CL ....................................... 45

3.2 A comparison of running times of TPA(FCED), RLA-SL and RLA-CL on simulated data sets (generated with a low error rate) ........................................... 57

3.3 A comparison of Type I accuracies of TPA(FCED), RLA-SL and RLA-CL on simulated data sets (generated with a low error rate) ..................... 58

3.4 A comparison of accuracies of TPA(FCED), RLA-SL and RLA-CL on simulated data sets (generated with a low error rate) ........................................... 59

3.5 A comparison of running time for variations of blocking attributes ..................... 61

3.6 A comparison of Type I accuracy for variations of blocking attributes ................. 61

3.7 Running time distribution of RLA-CL .......................................................... 62

3.8 Running time of PRLA-CL for 3.2M and 6.4M records ...................................... 63

3.9 Speedup of PRLA-CL for 3.2M and 6.4M records .......................................... 64

4.1 Web-based user interface. (A) shows the first and main page of the website, where users select data files, choose configurations and submit them. (B) is the instruction page. Users can view their submission history through login (C). (D) shows a sample submission history page. ........................................ 69

4.2 Screenshot of input parameter selection for our 3 example files. ............................ 71

4.3 Screenshot of linkage criteria for our 3 example files. ...................................... 73

5.1 Time per edge for random graphs with $2^{10}$ nodes and random edge weights. ............... 84

5.2 Time per edge for random graphs with $2^{16}$ nodes and random edge weights. ............... 85
List of Tables

1.1 Details of the input data set ......................................................... 10
1.2 Results of $k$-mer counters for $F. vesca$ ($k = 28$) ......................... 13
1.3 Results of $k$-mer counters for $G. gallus$ ($k = 28$) ......................... 13
1.4 Results of $k$-mers counters for $M. balbisiana$ ($k = 28$) ................. 14
1.5 Results of $k$-mer counters for $H. sapiens 1$ ($k = 28$) .................... 14
1.6 Results of $k$-mer counters for $H. sapiens 2$ ($k = 28$) .................... 15
1.7 Distribution of consumed time of KCMBT in the three phases (in seconds) ($k = 28$, 1 thread) ......................................................... 15
1.8 Generation of ($k + x$)-mers in the first phase of KCMBT ($k = 28$, 1 thread) ......................................................... 16
1.9 Count of $(k+x)$-mers after traversal in the second phase of KCMBT ($k = 28, 1$ thread) .................................................. 17
1.10 Insertion of $k$-mers in the first two phases of KCMBT ($k = 28, 1$ thread) .................................................................. 17
1.11 Distribution of consumed time of KCMBT for different number of burst trees ($k = 28, 1$ thread) .................................................. 18
1.12 Performances for different values of $x$ used for $(k + x)$-mers in KCMBT ($k = 28, 4$ threads) .................................................. 18

2.1 Comparison of results on simulated data .............................................................................................................. 33
2.2 Analysis of results on simulated data (RLA-SL) ................................................................................................. 34
2.3 Results on real datasets (1,083,878 records) ...................................................................................................... 37
2.4 Distribution of running time for 6,000,000 records ............................................................................................... 37

3.1 Comparison among TPA(FCED), RLA-SL and RLA-CL on simulated data sets (generated with a low error rate) ................................. 56
3.2 Computation of accuracy of TPA(FCED), RLA-SL and RLA-CL on simulated data sets (generated with a low error rate) ................................. 58
3.3 A comparison among TPA(FCED), RLA-SL and RLA-CL on simulated data sets (generated with a very high error rate) ................................. 59
3.4 A comparison of runtime and accuracy using SSN-LN, SSN and LN as blocking fields .............................................................. 60
3.5 Distribution of running time when blocking on SSN field .......................................................................................... 62
3.6 Distribution of running time on multiple cores .................................................................................................. 63

4.1 Records for 5 people having 9 attributes ................................................................................................................ 69
4.2 Records for 5 people having 4 attributes ................................................................................................................ 70
4.3 Records for 5 people having 8 attributes ................................................................................................................ 70
4.4 Generated output for our example data sets .................................................................................................. 74
4.5 Time comparison of RLT-S with FEBRL, FRIL, and TPA (FCED) .................................................................................. 75

5.1 Analysis of Different Phases of EPMST for Random Graphs with $2^{10}$ Nodes and Random Edge Weights ................................. 85
List of Algorithms

1. KCMBT (*k*-mer Counter based on Multiple Burst Trees) ........................................ 9
2. RLA-SL (Record Linkage Algorithm using Single Linkage Clustering) ................. 27
3. PRLA-SL (Parallel Record Linkage Algorithm using Single Linkage Clustering) .... 29
4. Exact Cluster Finding Algorithm ........................................................................... 46
5. Single Linkage Clustering Algorithm .................................................................... 48
6. RLA-CL (Record Linkage Algorithm using Complete Linkage Clustering) ......... 49
7. PRLA-CL Parallel Record Linkage Algorithm using Complete Linkage Clustering ... 52
8. EPMST (Edge Pruned Minimum Spanning Tree) .................................................. 81
9. Exact-1 Algorithm with $O(n)$ Memory ................................................................. 110
10. Approx algorithm .................................................................................................. 112
Part I

$k$-mer Counting Problem
Chapter 1

KCMBT: A $k$-mer counter based on multiple burst trees

1.1 Introduction

String algorithms have been frequently used in bioinformatics as genomic sequences can be represented by strings from an alphabet of distinct characters. A substring of length $k$ in a string is defined as a $k$-mer, where $k$ is a positive integer. $k$-mers in genomic sequences have been utilized to perform various analyses on the sequences. Numerous applications require counting the occurrences of particular $k$-mers. A $k$-mer counter computes the abundance of every unique $k$-mer in a string or a set of strings. It has become an elementary building block for various bioinformatics applications. Frequencies of $k$-mers along with the coverage information have been used in assembling genomic sequences \cite{54, 104, 115, 145}, correcting errors in sequencing reads to improve the assembly quality \cite{68, 85, 98}, finding repetitions, and solving many other bioinformatics problems \cite{94}.

There exist several efficient $k$-mer counting algorithms. If the available memory is very large, arrays with a straightforward $k$-mer indexing can be used for counting. As memory is limited, fast access hashing has become an alternate solution. Most of the currently available solutions engage this approach. Many existing algorithms also exploit the fact that the usage of unsigned integers instead of character strings simplifies the problem and
facilitates the solution. Spurious $k$-mers can be easily removed by employing a bloom filter. A bloom filter is a space-efficient probabilistic data structure that can be used to search for the existence of an item in a set. Suffix tree based solutions are also popular.

We introduce a novel internal memory technique called KCMBT ($k$-mer Counter based on Multiple Burst Trees), which uses cache efficient burst tries to solve this problem. A burst trie is a trie in which a full node is split into multiple nodes to make space for insertion of new elements. This algorithm combines a number of powerful ideas to enable faster output. These ideas include utilization of burst tries to store $k$-mers, consideration of $(k + x)$-mers, and unifying a $k$-mer and its count in a single unit. Currently KCMBT is the fastest $k$-mer counting algorithm. Experimental results in conjunction with theoretical analysis establish our statement.

1.2 Related Works

An obvious approach to $k$-mer counting is to use a hash table, where any $k$-mer is used as key and its count as value. If the available memory is very large, a simple array can be used to realize this hashing. On the other hand, simple hashing-based methods with limited memory suffer from large run times. Most of the available algorithms take advantage of multi-threading, out-of-core, and locking optimizations to design efficient hash-based solutions.

Tallymer [75] engages enhanced suffix arrays to store $k$-mers. But it is computationally expensive, and its memory requirement is highly dependent on the genome size and its coverage.

Jellyfish [94] introduces a multi-threaded, lock-free hash-table for $k$-mer counting. It exploits CAS assembly instruction to access and update a memory location in a multi-threaded environment.

Single occurrence $k$-mers may be produced due to sequencing errors. Bloom filter [7] is a good way to avoid counting most of these $k$-mers. Single-threaded BFCounter [100] uses this probabilistic data structure to store more frequent $k$-mers in reduced memory. It employs hash tables to store $k$-mers occurring at least two times. Jellyfish is faster than...
BFCounter in experimental comparisons.

Sort and compact instead of hashing is another technique to count $k$-mers. Turtle \cite{122} gathers $k$-mers in an array up to a certain point. Then it sorts and compacts them. This is an in-memory algorithm and the array should be large enough to hold all unique $k$-mers. Some versions utilize a bloom filter to remove spurious single occurrence $k$-mers.

Large memory requirements have been further reduced by disk-based methods. DSK \cite{121} counts $k$-mers with very low memory usage. It partitions $k$-mers according to hash values computed by some hash functions and stores them in the disk. Later it loads one partition at a time and counts $k$-mers using a hash table. It computes $k$-mer frequencies of human genome engaging only 4GB of memory.

Another disk-based parallel $k$-mer counting algorithm is KMC \cite{30}. It works in two phases: distribution phase and sort phase. Distribution phases collects $k$-mers into buffers. When buffers are full, it sends them to different disk files determined by prefixes of $k$-mers. After storing all the compact $k$-mers into disks, the sort phase brings back one file at a time, expands $k$-mers, sorts and merges them to count unique $k$-mers, and writes them back to the disk. This algorithm is very efficient in terms of time and space.

MSPKmerCounter \cite{83} is another efficient disk-based $k$-mer counter. It offers a new technique called Minimum Substring Partitioning (MSP) to partition reads in a more effective way. Another two phase external memory algorithm is KAnalyze \cite{4}. In the first phase, it accumulates $k$-mers until the allocated memory is full. Then it sorts and merges them, and writes them in the disk. In the second phase, all the disk files are merged in multiple steps.

The most efficient among all of these available $k$-mer counting algorithms is KMC2 \cite{31}. It is an extension of the KMC algorithm. It improves by the idea of minimizers to reduce memory and disk space, and the idea of $(k + x)$-mers.

1.3 Methods

We have devised a trie-based in-memory algorithm, KCMBT ($k$-mer Counter based on Multiple Burst Trees), for this $k$-mer counting problem. For large datasets internal $k$-mer
counting algorithms suffer from huge cache misses. We have found this issue for both hash-based and tree-based solutions. KCMBT shows greater improvement by using cache efficient modified burst tries for storing compact $k$-mers as well as proper usage of $(k + x)$-mers.

1.3.1 Burst tries

A burst trie \[47, 127\] is a trie that can be used to store a set of strings efficiently in almost sorted order. The overall trie data structure consists of three components: a set of strings to be placed, an access trie, and containers to contain those input strings. Containers can be thought of as leaves of this trie. Each string is matched against branches of the trie until it finds a container or it reaches past the input string. For the later case, there exists a container to store the string or just a terminal symbol or a counter to count such strings. For every internal node the number of children is at most the size of the alphabet. All the strings stored in a container have the same prefix. So the depth of a container is at most the size of the common prefix. Therefore strings without these common prefixes can be placed safely to reduce space requirement.

Insertion of a string into the trie compares the leftmost symbol of the string against branches of the root to find the next level node. Then it chooses the next symbols to move to descendent nodes until it associates with a container. When a container is full, strings of that container are sorted and the container is burst. Then a node with descendent containers is created in place of that parent container and strings of this parent container are distributed among the newly created children containers. The depth of the current container is one plus the depth of its parent container.

Inorder traversal passes containers in sorted order. But strings within a container may not be sorted yet. As the container size is small, a fast radix sort can easily be employed to sort these strings. After a full traversal, we get a set of sorted strings.

We can get a better insight of bursting of a container by following a simple example. Let the container size be 4, and there be a container $C$ having $\{CGCC, 1\}$, $\{ATGG, 1\}$, $\{GTGA, 1\}$, $\{CGCC, 1\}$. Note that the length of any substring in any container will be in general $\leq k$ (since we do not have to store the prefix corresponding to the container). Let
the length of the substrings in \( C \) be \( k' \). In our algorithm any \( k \)-mer with its count is stored in a single 64-bit unsigned integer. Now consider the insertion of another \( k \)-mer \( Q \) that belongs to the container \( C \) whose suffix (of length \( k' \)) is \( CAGG \). As this container is full, we sort and merge \( k' \)-mers of the container. As a result, we get \( \{ATGG, 1\}, \{CGCC, 2\}, \{GTGA, 1\} \), respectively. Then we split this container into 3 new containers by taking prefix 1-symbol of every \( k' \)-mer for branching. So the newly generated containers contain \( \{\{TGG, 1\}\}, \{\{GCC, 2\}\}, \{\{TGA, 1\}\} \), respectively. Consider now the insertion of the \( k \)-mer \( Q \). Since this starts with a \( C \), it will be stored in the second container, and the container has now \( \{\{GCC, 2\}\}, \{\{AGG, 1\}\} \). We see that there is no container for branch \( T \) yet. If we get such a \( k \)-mer, then we will create a container and store the \( k \)-mer in it.

1.3.2 Compact \( k \)-mers

To achieve faster access and computation, \( k \)-mers can be considered as binary streams. For this problem, we have only four symbols to keep track of. Therefore two bits are necessary to represent a symbol. We use one 64 bit unsigned integer to hold a full \( k \)-mer and its count. Sometimes the number of bits used for the counter may not be enough for counting the frequencies of some very frequent \( k \)-mers. We handle this case in a special way as described in the algorithm section. We use a few thousands of burst trees to ensure that the depths of these trees do not become too large. Having these many burst trees also helps us to compact \( k \)-mers more. If \( k = 28 \), we need 56 bits to store a \( k \)-mer. If we use a prefix length of 5 for indexing these tress, then we have \( 4^5 \) or 1024 trees. So we only use \((28 - 5) * 2 = 46\) bits for storing any \( k \)-mer. We get another 18 bits for storing the \( k \)-mer’s count. In this manner we are able to store a \( k \)-mer and its count together in a 64-bit unsigned integer.

1.3.3 \((k + x)\)-mers

We employ extended \( k \)-mers for better performance. An extended \( k \)-mer is nothing but a substring of length more than \( k \). We can identify these extended \( k \)-mers as follows. While processing any input string find the canonical \( k \)-mer or lexicographically minimum of current \( k \)-mer and its reverse complement. Then we move to next \( k \)-mer by sliding one
symbol. We also compute the minimum of this $k$-mer and its reverse complement. If both of these are from either forward or reverse direction, we express these two canonical $k$-mers by one $(k + 1)$-mer. If consecutive $(x + 1)$ canonical $k$-mers are from same direction, a $(k + x)$-mer can be generated by considering this $(k + x)$-length substring as an extended $k$-mer. Note that two successive $k$-mers share a $(k - 1)$-mer, and $l$-successive $k$-mers share a $(k - l + 1)$-mer. For $k' \geq k$, if a $k'$-mer has $c$ occurrences, then all of its constituent $k$-mers will have a count of at least $c$. If we combine successive $k$-mers in this way, the total required memory and time will be reduced. As an example, consider a portion of a read sequence $AAGCATA$. If $k = 4$, then the $k$-mers and their reverse complements in this portion are: \{AAGC, GCTT\}, \{AGCA, TGCT\}, \{GCAT, ATGC\}, and \{CATA, TATG\}. From each pair we take the canonical one, which is the minimum in lexicographic order. In this example, AAGC, AGCA, ATGC, and CATA are canonical. AAGC and AGCA are from the same direction. So we combine them into a $(k + 1)$-mer AAGCA. ATGC is the reverse complement of a 4-mer of the original portion of the sequence. This will be kept as a single $k$-mer. The remaining 4-mer CATA is also a single $k$-mer.

1.3.4 Our Algorithm

Our algorithm KCMBT works in three phases. The first phase starts with the insertion of $(k + x)$-mers, the second phase counts $(k + x)$-mers and converts them to $k$-mers and inserts them into $k$-mer specific trees, and the last phase employs a final traversal of these $k$-mer trees to identify all the unique $k$-mers with their counts. Our algorithm uses burst tries as canonical $(k + x)$-mer containers. Both strands of DNA may be present equally. We do not differentiate between those strands as the input read direction is unknown to us. We consider the lexicographically smaller $k$-mer between an input $k$-mer and its reverse complement as a canonical $k$-mer. A burst trie is a trie in which the prefix of a string is matched against branches of the tree corresponding to symbols of a specific alphabet. The suffix of the string is stored in a bucket, which is considered as a leaf of that trie. For this problem we consider only \{A, C, G, T\} as the alphabet. We remove all symbols other than A, C, G, and T from the input strings. Then we represent those strings by bit arrays for faster data manipulation. Currently we are using 64 bit unsigned integers to hold 32
symbols, where \( A \) is represented by 00, \( C \) by 01, \( G \) by 10, and \( T \) by 11. We do that because currently all available 64-bit machines can operate on 64-bit data. At first the tree has only one bucket, which is populated by \((k + x)\)-mers. When the bucket is full, one node is created along with at most 4 branches, each branch ends with a bucket as a leaf. MSB 2 bits (prefix 1 symbol) of \((k + x)\)-mers from the parent bucket are matched against edges of that tree to choose the proper bucket. These children buckets only contain suffixes of those \((k + x)\)-mers as all the \((k + x)\)-mers within a bucket share the same prefixes. In this way, whenever a bucket is full, we split it into at most 4 buckets under a node. The size of the buckets are dynamically allocated according to the size of the number of \((k + x)\)-mers they get from their parent bucket. In our algorithm, when we split a full bucket of the burst tree, we sort all the \((k + x)\)-mers, merge and compact them, and split them into at most 4 different buckets.

Our first phase is insertion of \((k + x)\)-mers, where \( x \) is a pre-defined value (0 \( \leq \) \( x \) \( \leq \) 3). It reads the input sequences, generates \( k \)-mers and their reverse complements, and takes the lexicographically smaller ones. If the current smaller \( k \)-mer and the previous smaller \( k \)-mer have the same directions, we increment the value of \( x' = x \), where \( x' \) starts with 0. If the direction changes or \( x' = x \), we insert the \((k + x')\)-mer into the corresponding tree. To make the process more cache efficient, KCMBT stores \((k + x')\)-mers of the same tree into a buffer. When that buffer is full, it inserts all the \((k + x')\)-mers of that buffer at a time. The rationale behind using \((k + x')\)-mers is that if a \((k + x')\)-mer is present \( c \) times, then each of its \( k \)-mers will be present at least \( c \) times. In this way we reduce a huge amount of computation required to insert all those \( x' + 1 \) \( k \)-mers separately.

There are two steps in phase two. The first step traverses every \((k + x')\)-mer tree, and counts all the unique \((k + x')\)-mers. The second step splits them into \( k \)-mers having the same count values, and inserts them into trees of similar prefix \( k \)-mer trees. So this phase combines the steps of splitting \((k + x')\)-mers and inserting \( k \)-mers. After this phase, we have only trees of \( k \)-mers.

At the last phase, we traverse all the available trees, and write unique \( k \)-mers with their counts to the disk. Some of the \( k \)-mers may have counts larger than can be stored in the available bits. We do not merge them at the time of traversal, instead we treat them
Algorithm 1 KCMBT (k-mer Counter based on Multiple Burst Trees)

**Input:** A set of sequencing reads and $k$ (the desired substring length)

**Output:** A list of unique $k$-mers in the input with their counts

1: procedure KCMBT
2: for each read sequence do
3:     Generate $(k + x)$-mers
4:     Insert generated $(k + x)$-mers into corresponding trees
5: end for
6: for each $(k + x)$-mer tree do
7:     Traverse the $(k + x)$-mer tree
8:     Split the $(k + x)$-mers into $k$-mers
9:     Insert each $k$-mer into a $k$-mer tree with the same prefix
10: end for
11: for each $k$-mer tree do
12:     Traverse the $k$-mer tree
13:     Write the $k$-mers and their counts in the disk
14: end for
15: end procedure

The multi-threaded implementation virtually splits each file into $t$ (the number of threads) portions. Each thread reads its assigned part of the sequences, and follows the first two phases of the above algorithm. If every thread constructs $n$ trees, then there will be a total of $nt$ trees. So every prefix has $t$ trees. In the last phase, the trees are shared among the threads. Trees with the same prefixes will be processed by the same thread. Specifically, every thread traverses $\frac{n}{t}$ prefix indexed trees, and accumulates the $k$-mers in them with their counts.

1.4 Results

We have compared our KCMBT implementation with previous best-known algorithms. KCMBT has been implemented in C++, and compiled with g++ along with optimization level 3. We ran all the algorithms on a 16 core machine (Dual Intel Xeon Processor E5-2667) with 512 GB DDR4 RAM, 12 TB HDD, 256 GB SATA SSD and Red Hat Linux Enterprise 7.0).

We have collected statistics for Jellyfish-2.2.4, KMC2-2.3.0, Turtle-0.3.1, DSK-2.0.7, and our KCMBT-1.0. We have chosen the latest working implementations of these algo-
Most of these algorithms perform much better than their original versions published in their respective publications.

As KMC2 is currently the fastest \( k \)-mer counting algorithm, we have attempted to compare with it more thoroughly. For this purpose, we have used the same input datasets (Table 1.1) that KMC2 used for experiments. We have received all the information from their publication. Our input data sets consist of five genomes with varied genome lengths. \( F. \) vesca is the smallest data set among these five, and \( H. \) sapiens 2 is the largest one. All of these genomes have multiple compressed fastq files. We have decompressed and concatenated them into one file, so that every tool can handle them easily.

Table 1.1: Details of the input data set

<table>
<thead>
<tr>
<th>Organism</th>
<th>Genome Length</th>
<th>No. of Bases</th>
<th>Input File Size</th>
<th>No. of Files</th>
<th>Avg. Read Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>( F. ) vesca</td>
<td>210</td>
<td>4.5</td>
<td>10.3</td>
<td>11</td>
<td>353</td>
</tr>
<tr>
<td>( G. ) gallus</td>
<td>1,040</td>
<td>34.7</td>
<td>115.9</td>
<td>15</td>
<td>100</td>
</tr>
<tr>
<td>( M. ) balbisiana</td>
<td>472</td>
<td>56.9</td>
<td>197.1</td>
<td>2</td>
<td>101</td>
</tr>
<tr>
<td>( H. ) sapiens 1</td>
<td>3,093</td>
<td>86.0</td>
<td>223.3</td>
<td>6</td>
<td>100</td>
</tr>
<tr>
<td>( H. ) sapiens 2</td>
<td>3,093</td>
<td>135.3</td>
<td>312.9</td>
<td>48</td>
<td>101</td>
</tr>
</tbody>
</table>

Genome length in Mbases, No. of bases in Gbases, files size in Gbytes

Jellyfish2 requires approximate numbers of \( k \)-mers stored in hash table as a parameter. If the number is less than actual number, then it may use some disk space to store \( k \)-mers and merge them later. From our experiments we have not seen any temporary file creation by Jellyfish except output file. We supplied around 10% more value than the exact unique \( k \)-mers count. For example we used 7000M hash size for \( H. \) sapiens 1 where we knew that the total number of unique \( k \)-mers is around 6339 millions. Some tools do not count \( k \)-mers with single occurrences. We ran all of these implementations in such a way that they output all the \( k \)-mers. Turtle is another internal memory algorithm. It has a necessary parameter for the expected number of unique \( k \)-mers to select the array size it uses for sorting and counting. We have also used the same number as we have used for Jellyfish2. Turtle comes with 3 different tools: scTurtle, cTurtle and aTurtle. Each one has two versions to support for maximum 32-mer and 64-mer computations. scTurtle counts \( k \)-mers with frequency > 1, cTurtle only reports \( k \)-mers with frequency > 1 without showing their counts, and aTurtle
gives $k$-mers with all frequencies along with their counts. scTurtle and cTurtle support multiple threads, but aTurtle is single threaded. We have used aTurtle for our experiments as it counts $k$-mers with all frequencies. KMC2 is mainly an external memory or disk-based algorithm, but it has an option to make it an internal memory-based algorithm. Original publication shows results for using 6 GB memory and 12 GB memory limit options. Later option performs better, although we see that it uses less than allocated memory. We have only included results for the 12 GB memory limit option. DSK is another memory frugal disk-based algorithm. It can complete $k$-mer counting of human genome using only 4 GB of memory. We chose a fixed 6 GB of memory for all the tests although it did not use all of it.

KCMBT is a cache efficient algorithm. It has several parameters which have substantial effects on the running time. Containers of burst trie contain $k$-mers. If the container size is large, many containers will be partially filled up, and it will leave a huge memory unused. On the other hand, the depth of the trees will be low, and consequently insertion and traversal will take less time. Also there will be less number of sorting, as sorting is called when a container is full. But if the container size is small, memory requirement is low. Sorting and bursting are called frequently, and the height of the tree will be high. Hence insertion and traversal need to compare many branches, and the running time will be increased. Even if the container size is large, consumed time may be high because of numerous cache misses. Another important factor is the number of trees. If we have only one tree, it becomes giant for large genomes. Height of the tree grows fast, and cache misses occur constantly. To keep the depth reasonable, we employ hundreds of trees to store $k$-mers. Index of the tree for a $k$-mer is determined by its prefix. As a tree contains $k$-mers with the same prefix, there is no need to keep that prefix in those $k$-mers. So we can use this spare bits to manage counting. There are some optimal values for the number of trees dependent on the genome size, cache, and memory size. We see in our experiments that $4^5$ or 1024 trees work the best for 2 or 4 threads, but $4^6$ or 4096 is a good number for a single thread. We chose $4^6$ for experiments with a single thread, and $4^5$ for multiple threads. We insert $k$-mers into trees in batches. If we insert one $k$-mer at a time, caches have to be refreshed possibly each time, which is very time consuming. So we fill a buffer
of a fixed size for each tree until it is full. Then we insert these $k$-mers into that tree. We have found that a buffer size of 1024 is a good value for our experiments. Another major impact factor is the value of $x$ for $(k + x)$-mers. Generation of $(k + x)$-mers is time consuming, but it reduces the total number of insertions or traversals. We use $x = 3$ as a default value as we have received good results with this value.

*F. vesca* has a comparatively smaller genome size and coverage. We ran Jellyfish2, Turtle, KMC2, DSK and KCMBT. From Table 1.2 we see that Jellyfish2 is the slowest one among these. Turtle is faster than Jellyfish2, but it has occupied much more memory than Jellyfish2. All of our tests count all the $k$-mers including single occurrence $k$-mers. cTurtle and scTurtle do not provide perfect counting. So we used single threaded aTurtle32 for our purposes. DSK is well-known for its careful memory usage. KMC2 is a popular disk-based $k$-mer counter. It requires almost the same amount of memory as DSK, but it is much faster than DSK. The internal memory version of KMC2 consumes the same amount of time as the external memory one. Our KCMBT implementation shows a remarkable improvement. It is around 50% faster than KMC2 for single thread and two threads, and around 30% faster for four threads. We see that KCMBT is around 6 times faster than Jellyfish2 for 1 thread and 2 threads, and 4 times for 4 threads. It uses 14 GB memory for 1 thread, where 8 GB for 2 threads and 11 GB for 4 threads.

We have noticed that KMC2 is more than two times faster when running with 4 threads than when running with 2 threads. KMC2 has 2 phases. It uses 1 thread for reading sequences in first phase. This reading thread does not employ CPU all the time. Therefore it wastes some time here. So when we run with 1 thread or 2 threads, the first phase generally uses 2 threads. One thread is for reading sequences and another one is for computational works. So the speed up is not that much good for 2 threads. But for 4 threads there are 3 threads for computation compared to 1 in 2 threads.

Jellyfish2 takes more than 90 minutes to complete the counting of $k$-mers in *G. gallus* for 1 thread and 25 minutes for 4 threads. KCMBT has also performed excellently for this data set, which took 22%-30% less time than KMC2. KMC2 needed 445 seconds for 4 threads, whereas KCMBT takes 347 seconds. KMC2(RAM) spends the same amount of time as KMC2 for this genome. DSK is much slower than KMC2. Table 1.3 compares
### Table 1.2: Results of k-mer counters for *F. vesca* (*k* = 28)

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>1 Thread</th>
<th>2 Threads</th>
<th>4 Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RAM</td>
<td>Disk</td>
<td>Time</td>
</tr>
<tr>
<td>Jellyfish2</td>
<td>6</td>
<td>0</td>
<td>932</td>
</tr>
<tr>
<td>Turtle</td>
<td>14</td>
<td>0</td>
<td>652</td>
</tr>
<tr>
<td>KMC2</td>
<td>12</td>
<td>4</td>
<td>298</td>
</tr>
<tr>
<td>KMC2RAM</td>
<td>8</td>
<td>0</td>
<td>309</td>
</tr>
<tr>
<td>DSK</td>
<td>6</td>
<td>9</td>
<td>392</td>
</tr>
<tr>
<td>KCMBT</td>
<td>14</td>
<td>0</td>
<td>160</td>
</tr>
</tbody>
</table>

RAM and Disk in GB, Time in Sec

These tools are for *G. gallus*.

### Table 1.3: Results of k-mer counters for *G. gallus* (*k* = 28)

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>1 Thread</th>
<th>2 Threads</th>
<th>4 Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RAM</td>
<td>Disk</td>
<td>Time</td>
</tr>
<tr>
<td>Jellyfish2</td>
<td>21</td>
<td>0</td>
<td>5536</td>
</tr>
<tr>
<td>Turtle</td>
<td>56</td>
<td>0</td>
<td>4707</td>
</tr>
<tr>
<td>KMC2</td>
<td>12</td>
<td>26</td>
<td>1633</td>
</tr>
<tr>
<td>KMC2RAM</td>
<td>31</td>
<td>0</td>
<td>1657</td>
</tr>
<tr>
<td>DSK</td>
<td>6</td>
<td>48</td>
<td>2871</td>
</tr>
<tr>
<td>KCMBT</td>
<td>42</td>
<td>0</td>
<td>1110</td>
</tr>
</tbody>
</table>

RAM and Disk in GB, Time in Sec

aTurtle32 was run for *M. balbisiana*, but we waited for several hours without observing any noticeable update, and then we killed the process. So we did not include Turtle for all of our other tests. [31] also showed that Turtle was an underperformer than Jellyfish2 although they included results for scTurtle, which reports k-mers with frequency above 1. Table 1.4 displays results for *M. balbisiana*. Jellyfish2 uses only 11 GB of memory to count k-mers for *M. balbisiana*, but it is more than 3 times slower than KMC2. KCMBT is also around 30% faster than KMC2, and DSK takes more than double the time taken by KCMBT. We see from Table 1.2, Table 1.3, and Table 1.4 that KCMBT for 2 threads and 4 threads takes much less memory than for 1 thread. The memory usage depends on several factors such as k-mer distribution of that input genome, percentage of buckets filled up in burst trees, and so on.

Human genome is a massive as well as vital data set. We have compared these tools for *H. sapiens 1* and *H. sapiens 2* and arranged the results in Table 1.5 and Table 1.6.
Table 1.4: Results of \(k\)-mers counters for \(M.\) balbisiana \((k = 28)\)

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>1 Thread</th>
<th>2 Threads</th>
<th>4 Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RAM</td>
<td>Disk</td>
<td>Time</td>
</tr>
<tr>
<td>Jellyfish2</td>
<td>11</td>
<td>0</td>
<td>8133</td>
</tr>
<tr>
<td>KMC2</td>
<td>12</td>
<td>41</td>
<td>2279</td>
</tr>
<tr>
<td>KMC2RAM</td>
<td>47</td>
<td>0</td>
<td>2450</td>
</tr>
<tr>
<td>DSK</td>
<td>6</td>
<td>42</td>
<td>4504</td>
</tr>
<tr>
<td>KCMBT</td>
<td>37</td>
<td>0</td>
<td>1731</td>
</tr>
</tbody>
</table>

RAM and Disk in GB, Time in Sec

\(H.\) sapiens 1 has around 62.7 billion 28-mers and 6.3 billion unique 28-mers. Jellyfish2 is still a very memory efficient internal memory algorithm, as it has used only 41 GB of memory to hold these huge number of unique \(k\)-mers and their counts in memory. But it is very slow compared to other tools in Table 1.5. KMC2 is five times faster than this one. KMC2(RAM) achieves a similar speed up. DSK seems to be slow compared to KMC2 and KCMBT. KMC2 is slower than KCMBT by around 35% for 1 thread, 20% for 2 threads, and 17% for 4 threads.

Table 1.5: Results of \(k\)-mer counters for \(H.\) sapiens 1 \((k = 28)\)

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>1 Thread</th>
<th>2 Threads</th>
<th>4 Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RAM</td>
<td>Disk</td>
<td>Time</td>
</tr>
<tr>
<td>Jellyfish2</td>
<td>41</td>
<td>0</td>
<td>16433</td>
</tr>
<tr>
<td>KMC2</td>
<td>12</td>
<td>64</td>
<td>3657</td>
</tr>
<tr>
<td>KMC2RAM</td>
<td>69</td>
<td>0</td>
<td>3647</td>
</tr>
<tr>
<td>DSK</td>
<td>6</td>
<td>64</td>
<td>8233</td>
</tr>
<tr>
<td>KCMBT</td>
<td>70</td>
<td>0</td>
<td>2703</td>
</tr>
</tbody>
</table>

RAM and Disk in GB, Time in Sec

\(H.\) sapiens 2 has more coverage than \(H.\) sapiens 1. The comparative analysis among these tools remains the same from \(H.\) sapiens 1 to \(H.\) sapiens 2. In this case, KMC2 is around 45% slower than KCMBT for 1 thread and 30% slower for 4 threads. KCMBT consumes 82 GB memory for a single thread, whereas KMC2(RAM) uses 107 GB. But KCMBT takes 147 GB memory for 4 threads as every prefix has 4 trees and similar \(k\)-mers exist in 4 trees.

KCMBT works in three phases. The most time consuming phase is the generation of
Table 1.6: Results of $k$-mer counters for *H. sapiens* 2 ($k = 28$)

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>1 Thread</th>
<th>1 Thread</th>
<th>2 Threads</th>
<th>2 Threads</th>
<th>4 Threads</th>
<th>4 Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RAM</td>
<td>Disk</td>
<td>Time</td>
<td>RAM</td>
<td>Disk</td>
<td>Time</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>RAM</td>
<td>Disk</td>
<td>Time</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>RAM</td>
<td>Disk</td>
<td>Time</td>
</tr>
<tr>
<td>Jellyfish2</td>
<td>41</td>
<td>0</td>
<td>23651</td>
<td>41</td>
<td>0</td>
<td>11938</td>
</tr>
<tr>
<td>KMC2</td>
<td>12</td>
<td>101</td>
<td>5813</td>
<td>12</td>
<td>101</td>
<td>3737</td>
</tr>
<tr>
<td>KMC2RAM</td>
<td>107</td>
<td>0</td>
<td>5584</td>
<td>106</td>
<td>0</td>
<td>3714</td>
</tr>
<tr>
<td>DSK</td>
<td>6</td>
<td>71</td>
<td>15746</td>
<td>6</td>
<td>70</td>
<td>8753</td>
</tr>
<tr>
<td>KCMBT</td>
<td>82</td>
<td>0</td>
<td>4021</td>
<td>100</td>
<td>0</td>
<td>2433</td>
</tr>
</tbody>
</table>

(RAM and Disk in GB, Time in Sec)

$(k + x)$-mers and insertion of them. We have chosen $0 \leq x \leq 3$ for all of our tests. The first phase generates $k$-mers and their reverse complements, and tracks the minimum or canonical ones. Then it slides one symbol, computes new $k$-mer value and a new reverse complement. It again finds the minimum one of these two. If this minimum and the previous minimum are from the same direction (given a $k$-mer direction or its reverse complement direction), it gets a $(k + 1)$-mer. Then it shifts symbol again to extend till $x = 3$ or directions of minimum and previous minimum are different. We maintain fixed length buffers for each tree. Whenever a buffer is full, all the $(k + x)$-mers are inserted into the tree. Insertion is also time consuming. It incorporates traversal to find the proper bucket, and if the bucket is full, a new node and buckets are created and the distribution of $(k + x)$-mers takes place. So we see from Table 1.7 that $59\%$ of the total time for *F. vesca* was needed for this first phase. This value is $78\%$ for *G. gallus*, $90\%$ for *M. balbisiana*, $81\%$ for *H. sapiens 1*, and $86\%$ for *H. sapiens 2*. This phase takes a major portion of the total time, and the ratio increases according to the size of the total number of $k$-mers and unique $k$-mers.

Table 1.7: Distribution of consumed time of KCMBT in the three phases (in seconds) ($k = 28$, 1 thread)

<table>
<thead>
<tr>
<th>Genome</th>
<th>Insertion</th>
<th>Traversal Insertion</th>
<th>Traversal</th>
<th>Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>F. vesca</em></td>
<td>95</td>
<td>39</td>
<td>26</td>
<td>160</td>
</tr>
<tr>
<td><em>G. gallus</em></td>
<td>871</td>
<td>173</td>
<td>66</td>
<td>1110</td>
</tr>
<tr>
<td><em>M. balbisiana</em></td>
<td>1570</td>
<td>124</td>
<td>37</td>
<td>1731</td>
</tr>
<tr>
<td><em>H. sapiens 1</em></td>
<td>2205</td>
<td>363</td>
<td>135</td>
<td>2703</td>
</tr>
<tr>
<td><em>H. sapiens 2</em></td>
<td>3460</td>
<td>420</td>
<td>141</td>
<td>4021</td>
</tr>
</tbody>
</table>
We show the importance of generation and insertion of \((k + x)\)-mers in the first phase in Table 1.8. The last column indicates how many \(k\)-mers will have to be inserted if we only insert \(k\)-mers instead of \((k + x)\)-mers, where \(x > 0\). For each data set, we had to call insertion less than 50% of the times because of the benefits of generation of \((k + x)\)-mers. We also see that this phase produces a majority number of \((k + 3)\)-mers after \(k\)-mers for every considered genome except \(M.\) \textit{balbisiana}. This point is beneficial for the second and the third phases. This reduction in the number of insertions comes at some expense on \((k + x)\)-mers generation.

Table 1.8: Generation of \((k + x)\)-mers in the first phase of KCMBT \((k = 28, 1\) thread)

<table>
<thead>
<tr>
<th>Genome</th>
<th>((k + 0))-mers</th>
<th>((k + 1))-mers</th>
<th>((k + 2))-mers</th>
<th>((k + 3))-mers</th>
<th>((k + x))-mers total</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{F. vesca}</td>
<td>785</td>
<td>418</td>
<td>258</td>
<td>435</td>
<td>1896</td>
</tr>
<tr>
<td>\textit{G. gallus}</td>
<td>4688</td>
<td>2520</td>
<td>1590</td>
<td>2710</td>
<td>11508</td>
</tr>
<tr>
<td>\textit{M. balbisiana}</td>
<td>8067</td>
<td>4322</td>
<td>2607</td>
<td>4133</td>
<td>19129</td>
</tr>
<tr>
<td>\textit{H. sapiens 1}</td>
<td>11743</td>
<td>6303</td>
<td>3945</td>
<td>6639</td>
<td>28630</td>
</tr>
<tr>
<td>\textit{H. sapiens 2}</td>
<td>18565</td>
<td>9807</td>
<td>6226</td>
<td>10508</td>
<td>45106</td>
</tr>
<tr>
<td>\textit{H. sapiens 2}</td>
<td>18565</td>
<td>9807</td>
<td>6226</td>
<td>10508</td>
<td>45106</td>
</tr>
</tbody>
</table>

Number of \((k + x)\)-mers in millions

The first phase completes a considerable part of the work. The second phase traverses all the \((k + 1)\)-mer, \((k + 2)\)-mer, and \((k + 3)\)-mer trees. After the traversal of a \((k + 1)\)-mer tree, we get the \((k + 1)\)-mer count of that specific \((k + 1)\)-mer. If this count value is \(c\), then the constituent two \(k\)-mers (starting at position 0 and position 1, respectively) will have a count of \(c\) from this \((k + 1)\)-mer. \((k + 2)\)-mer trees contain \((k + 2)\)-mers, each of which has 3 \(k\)-mers. For a \((k + 3)\)-mer, this value is 4. So when \(x\) is large, our gain is large as well. Table 1.9 shows how many unique \((k + x)\)-mers exist after the traversal in phase 2, where \(1 \leq x \leq 3\). Let \(c_1\), \(c_2\), and \(c_3\) be the counts of \((k + 1)\)-mers, \((k + 2)\)-mers, and \((k + 3)\)-mers found in phase 2, respectively. We know that a \((k + x)\)-mer covers \((x + 1)\) \(k\)-mers. So we get \(2c_1 + 3c_2 + 4c_3\) \(k\)-mers from only \(c_1 + c_2 + c_3\) \((k + x)\)-mers, where \(1 \leq x \leq 3\). For \textit{H. sapiens 2}, we observe that the average counts of \((k + 1)\)-mers, \((k + 2)\)-mers, and \((k + 3)\)-mers is around a billion. This average is more than 65 million for \textit{F. vesca}, 300 million for \textit{G. gallus}, 125 million for \textit{M. balbisiana}, and 760 million for \textit{H. sapiens 1}. This improved achievement is the main reason behind the consideration of \((k + x)\)-mers. From all of these
experiments, we discover that high expenses at first phase are substantially recovered by second and third phases.

Table 1.9: Count of \((k + x)\)-mers after traversal in the second phase of KCMBT \((k = 28, 1\) thread)

<table>
<thead>
<tr>
<th>Genome</th>
<th>((k + 1))-mers</th>
<th>((k + 2))-mers</th>
<th>((k + 3))-mers</th>
</tr>
</thead>
<tbody>
<tr>
<td>(F.\ vesca)</td>
<td>72</td>
<td>46</td>
<td>89</td>
</tr>
<tr>
<td>(G.\ gallus)</td>
<td>364</td>
<td>235</td>
<td>411</td>
</tr>
<tr>
<td>(M.\ balbisiana)</td>
<td>142</td>
<td>90</td>
<td>145</td>
</tr>
<tr>
<td>(H.\ sapiens) 1</td>
<td>814</td>
<td>521</td>
<td>955</td>
</tr>
<tr>
<td>(H.\ sapiens) 2</td>
<td>1007</td>
<td>645</td>
<td>1076</td>
</tr>
</tbody>
</table>

Number of \((k + x)\)-mers in millions

Table 1.10 displays the total numbers of \(k\)-mers in \(k\)-mer trees after the first two phases. If the number of \(k\)-mers in a tree increases, the height of the tree might increase. The other effects are more insertion time and traversal time. We have thousands of trees to reduce the height of these trees. The idea of generation, insertion, and traversal of \((k + x)\)-mers also facilitates our intention of keeping the trees within a reasonable height. The total number of \(k\)-mers in \(k\)-mer trees is 18,754 millions instead of 62,739 millions for \(H.\ sapiens\) 1, 26,818 millions instead of 98,893 millions for \(H.\ sapiens\) 2. These values are 4-5 times smaller than originally required insertions.

Table 1.10: Insertion of \(k\)-mers in the first two phases of KCMBT \((k = 28, 1\) thread)

<table>
<thead>
<tr>
<th>Genome</th>
<th>phase 1 ((k + 0)) -mers</th>
<th>from ((k + 1)) -mers</th>
<th>from ((k + 2)) -mers</th>
<th>from ((k + 3)) -mers</th>
<th>insertion total (k)-mers</th>
<th>total (k)-mers</th>
</tr>
</thead>
<tbody>
<tr>
<td>(F.\ vesca)</td>
<td>785</td>
<td>144</td>
<td>138</td>
<td>356</td>
<td>1423</td>
<td>4134</td>
</tr>
<tr>
<td>(G.\ gallus)</td>
<td>4688</td>
<td>728</td>
<td>705</td>
<td>1644</td>
<td>7765</td>
<td>25338</td>
</tr>
<tr>
<td>(M.\ balbisiana)</td>
<td>8067</td>
<td>284</td>
<td>270</td>
<td>580</td>
<td>9201</td>
<td>41063</td>
</tr>
<tr>
<td>(H.\ sapiens) 1</td>
<td>11743</td>
<td>1628</td>
<td>1563</td>
<td>3820</td>
<td>18754</td>
<td>62739</td>
</tr>
<tr>
<td>(H.\ sapiens) 2</td>
<td>18565</td>
<td>2014</td>
<td>1935</td>
<td>4304</td>
<td>26818</td>
<td>98893</td>
</tr>
</tbody>
</table>

Number of \((k + x)\)-mers in millions

As the inserted numbers of \(k\)-mers have been reduced a lot, the time for traversal in the third phase is dramatically reduced. We observe from Table 1.10 that very little amount of time was spent for this final traversal. Third phase traverses these \(k\)-mer trees, and produces \(k\)-mers with their counts. If we do not use \((k + x)\)-mers for some \(x > 0\), then we
have to traverse larger trees, which is very time taking.

Table 1.11: Distribution of consumed time of KCMBT for different number of burst trees ($k = 28$, 1 thread)

<table>
<thead>
<tr>
<th>Genome</th>
<th>1024</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>$F. \text{vesca}$</td>
<td>153</td>
<td>160</td>
</tr>
<tr>
<td>$G. \text{gallus}$</td>
<td>1328</td>
<td>1110</td>
</tr>
<tr>
<td>$M. \text{balbisiana}$</td>
<td>1585</td>
<td>1731</td>
</tr>
<tr>
<td>$H. \text{sapiens 1}$</td>
<td>3007</td>
<td>2703</td>
</tr>
<tr>
<td>$H. \text{sapiens 2}$</td>
<td>5409</td>
<td>4021</td>
</tr>
</tbody>
</table>

Time in seconds

The number of trees has a huge impact on the running time. If we increase the number of trees, the average height of trees decreases. Therefore insertion and traversal take less time. But after a certain number, the running time starts to increase. Because there are already many trees and cache misses occur frequently at the time of insertion. Our observations from experimental results imply that 1024 ($4^5$) or 4096 ($4^6$) are quite good numbers for the trees for these data sets. We used $4^6$ for all of these experiments. We have included elapsed time for all of these data sets for these two values in Table 1.11.

Table 1.12: Performances for different values of $x$ used for $(k+x)$-mers in KCMBT ($k = 28$, 4 threads)

<table>
<thead>
<tr>
<th>Genome</th>
<th>$x = 1$</th>
<th>$x = 2$</th>
<th>$x = 3$</th>
<th>$x = 4$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RAM</td>
<td>Time</td>
<td>RAM</td>
<td>Time</td>
</tr>
<tr>
<td>----------</td>
<td>---------</td>
<td>---------</td>
<td>---------</td>
<td>---------</td>
</tr>
<tr>
<td>$F. \text{vesca}$</td>
<td>12</td>
<td>69</td>
<td>11</td>
<td>61</td>
</tr>
<tr>
<td>$G. \text{gallus}$</td>
<td>60</td>
<td>424</td>
<td>55</td>
<td>389</td>
</tr>
<tr>
<td>$M. \text{balbisiana}$</td>
<td>34</td>
<td>550</td>
<td>30</td>
<td>479</td>
</tr>
<tr>
<td>$H. \text{sapiens 1}$</td>
<td>114</td>
<td>1017</td>
<td>109</td>
<td>919</td>
</tr>
<tr>
<td>$H. \text{sapiens 2}$</td>
<td>175</td>
<td>1670</td>
<td>149</td>
<td>1467</td>
</tr>
</tbody>
</table>

RAM in GB, Time in Sec

We have chosen $x = 3$ for all of our experiments. We see from Table 1.12 that $x = 3$ is the optimal value for all of our data sets. For $x = 1$ it requires more memory and time than $x = 2$ or $x = 3$. It employs more memory for $x = 5$ than other used values. Large values of $x$ will reduce the number of generated $(k+x)$-mers and the traversal time. But the generation of $(k+x)$-mers will take more time. Also, we have to construct a large number of trees and the cache misses will be frequent after a certain value of $x$. Our experimental results show that 3 is the optimal value of $x$. 

18
1.5 Discussion

KCMBT outperforms every other $k$-mer counter by a large margin. KMC2 is currently the best-performing $k$-mer counter. It requires a low internal memory and some not so much inexpensive disk storage. It is practically a very fast $k$-mer counter with several good options. We have collected data sets of the five genomes used in our experiments from available links in the published paper of KMC2 [31]. In this KMC2 paper the authors also explain how they ran the other programs. We have tried to employ the same values for the underlying parameters except for the number of cores.

We have shown in the previous section how our ideas were fruitful in counting $k$-mers efficiently. Our algorithm chooses burst tries to store $k$-mers. Burst trie is very cache efficient for keeping strings in approximate sorted order. We need that for holding somewhat similar $k$-mers together. For large genomes, there exist enormous numbers of $k$-mers. A single tree is not enough to store all of them efficiently. Insertion and traversal become very time expensive operations because of the large height of the tree. We employed hundreds of trees to resolve this issue. When the number of trees is large, we can remove some prefix bits from each $k$-mer to index its corresponding tree. These extra bits can be used to store counts. Keeping a $k$-mer and its count together in an unsigned 64 bit word is indeed helpful. Since the alphabet size is 4, each symbol needs at least 2 bits. If we want a 5 symbol prefix, there will be $4^5$ or 1024 trees. We ran KCMBT for prefix 0-7 symbols. For 0 to 4 value, some of these tree heights become large. For a value of 7, there are a massive number of trees, and cache misses occur regularly. We have found good results for values 5 and 6. It is not fruitful to insert one $k$-mer at a time. If we choose this option, in some of the cases the running time rises more than 25% because of frequent cache swapping. Therefore we choose a large buffer to cache these $k$-mers and insert them at a time, which is very cache friendly.

Our first attempt was generation and insertion of $k$-mers. There were two phases. The first phase inserted these $k$-mers into trees and the second phase traversed them to accumulate all $k$-mers with their counts. But this idea was not good enough to outperform KMC2 in some cases. We have adapted the very good idea of using of $(k + x)$-mers from
KMC2. We have followed almost the same idea to generate these \((k + x)\)-mers. KMC2 generates them from super \(k\)-mers stored in disk files. We form them from \(k\)-mers in the first phase. Experimental outcomes prove the usefulness of this idea. It eliminates many insertion and traverse counts. We generally use \((k + x)\)-mers, where \(0 \leq x \leq 3\). A value of more than 3 for \(x\) works better if they have enough duplicate occurrences. We have noticed that \(0 \leq x \leq 3\) performs better in our experiments.

In the second phase, we traverse \((k+x)\)-mer trees, and count the number of occurrences of each \((k+x)\)-mer. We then split each \((k+x)\)-mer into \(k\)-mers, and insert them into \(k\)-mer trees. We could have improved the time by not inserting the \(k\)-mers occuring in the first positions of \((k + x)\)-mers. If we did that, we will have to merge at a later stage. The process would be complicated. As the total time spent in the last two phases is not that much, we have avoided this complexity.
Part II

Record Linkage Problem
Chapter 2

Efficient sequential and parallel algorithms for record linkage

2.1 Introduction

Identifying duplicates in voluminous datasets is a crucial problem in many areas of science and engineering. This is especially true for medical records of individuals from different health agencies. Integration of medical records provides a great opportunity to analyze and evaluate disease evolution [22, 35]. Methods [135] exist for linking records across multiple medical data centers to identify disease origin and diversity [86]. Copy detection in digital documents also employs data integration techniques to detect similarities [10, 125]. Data Integration techniques integrate records across different data sources usually in the absence of any global identifier. This is a way to identify individuals who have records in different data sets. If all the records pertaining to the same individual are exactly correct, the problem of identifying duplicates will be straightforward to solve. Unfortunately, records of the same person might look different owing to errors introduced by typing, phonetic similarity, etc. As a result, the record linkage problem is very challenging. Existing algorithms take very long time especially when the data size is large. Thus, it is still an important open problem to discover faster algorithms. In this research work we propose a sequential algorithm that is up to two orders of magnitude faster than one of
the best-known algorithms called TPA (FCED) [102]. We also present a parallel algorithm that achieves almost linear speedup.

A huge number of approaches have been developed in the literature. Most of these algorithms link two datasets at a time. In practice, we have more than two data sets. If we have two data sets $A$ and $B$ and if $n_a$ and $n_b$ are the numbers of records in them respectively, then in the worst case we have to process $n_a \cdot n_b$ record pairs [38]. Some learning algorithms generate comparison vectors and classify them [41], which take a large amount of time to generate the vectors.

We have used hierarchical clustering as the basis for our algorithms [141, 146], which is also widely applied in information theory [136], gene expression [49, 46, 52, 29], data mining [142, 107], health psychology [23], and many other fields to identify distributions of corresponding objects or data. Our algorithms use single linkage method to calculate distances. To reduce load on calculating linkages, we employ radix sort initially on records [124]. Our algorithms also consider different types of errors including typing distance, reversal of the first name and the last name, use of nicknames, truncation of attributes, etc. [102]. We have thoroughly tested our algorithms on a large number of synthetic and real datasets. These tests show that the proposed algorithms outperform previous algorithms in terms of time and space. The parallel algorithm attains nearly linear speedup.

2.2 Background and Significance

Record linkage among multiple datasets typically involves millions of records and hundreds of thousands of individuals. The problem of record linkage can be thought of as one of clustering the records such that each cluster has records pertaining to one and only one individual [18]. Clustering, in general, is the process of partitioning objects so that similar objects are grouped into the same group (i.e., cluster). A number of clustering methods can be found in the literature, including hierarchical clustering, graph-based clustering, statistical clustering, centroid based clustering, etc. Any clustering method employs a metric (known as linkage) for defining the distance between two clusters. Distance between two clusters indicates how similar these two clusters are. In complete linkage, distance
between two clusters (of records) A and B is defined as the maximum distance between a record in A and a record in B, while single linkage uses the minimum distance. The distance between two given records can also be defined in a number of ways. Examples include the Levenshtein distance (also known as the edit distance) and the Hamming distance.

Hierarchical clustering can be done in two different ways: (1) agglomerative approach (bottom-up) starts with $n$ clusters (where $n$ is the number of records or points to be clustered) where each cluster has a single point. From there on clustering happens in iterations where in each iteration the two closest clusters are merged into one. Iterations stop when we have only a single cluster containing all the $n$ points. The sequence of merging steps done in the algorithm can be represented as a tree called a dendrogram. If we have a target number of clusters in mind, we can cut the dendrogram at an appropriate level. The dendrogram can also be cut using a cluster threshold distance. (2) divisive clustering approach (top-down) starts with a single cluster having all the $n$ points. Then this cluster is split hierarchically until we end up with $n$ clusters, each cluster having a single point.

Here we employ agglomerative hierarchical clustering, using single linkage. We treat each record as a string of characters and define the distance between two records based on edit distance. Different kinds of common errors have been taken into account including reversal of first and last names, truncation of attributes, etc. [102].

2.3 Methods

2.3.1 Previous Methods

A simple brute force approach for record linkage is to compute the distance between every pair of records and identify the pair as a match or a non-match. This will take too much time. Some of the previous methods generate comparison vectors and define classification [41]. Cluster-based entity resolution that uses both relational and attribute information has been shown to perform better than attribute-based record linkage [6]. Linking several datasets using record linkage methods [139] and deduplication [21] to merge records and remove repetitions are popular techniques. A wide range of studies on methods for record linkage have been done [138]. Expectation-Maximization (EM) algorithm provides
improved decision rule in the Fellegi-Sunter Model of record linkage by employing probability estimation [140]. Traditional probabilistic linkage models classify pairs of records as matches if they agree on some of their common attributes and non-matches otherwise [36]. Probabilistic linkage system AutoMatch results in better linkage quality than some deterministic ones as shown in a recent study [38]. Many other probabilistic methods also exist [34, 137, 138]. Identity uncertainty and citation matching problems have been solved by relational probability model [113]. Conditional models also cover the problem of identity uncertainty [97]. Conditional random fields have been used to segment and label data [76]. These are also applied in a relational partitioning algorithm [26]. Multi-relational record linkage allows propagation of matches [33]. Personal name matching techniques [16], distance calculation [80], matching methods [58], automated correction of text techniques [74], Longest Common Substring [37], and many other techniques are also available for comparisons.

FEBRL is famous for the linkage of two data sets [20, 17]. IntelliClean is another framework to identify duplicates by computing the transitive closure under uncertainty and anomalies efficiently [78]. Multi-pass approach for merge/purge problem considers alternate key attributes and applies these results to compute the transitive closure [48]. Many of these techniques use blocking phase as a preprocessing step where the records are hashed into buckets (or blocks) based on some of the characters in the records, including Canopy Clustering [96]. Unsupervised and unconstrained partition-based clustering algorithms exist which are different from hierarchical clustering methods [45]. We have improved the TPA (FCED) algorithm, which is one of the fastest known record linkage algorithms, significantly.

Some parallel algorithms for hierarchical clustering have been developed [81, 109, 118, 143]. Parallel methods for record linkage also exist [65, 70, 71, 27]. P-Swoosh uses match and merge processes, and also uses domain knowledge [65]. An algorithm that performs better than P-Swoosh has been given in [70]. This algorithm achieves an almost linear speed-up, e.g., 6.55-7.49 on 8 processors. A different blocking technique in initial data partitioning followed by a matching phase has also been introduced [71, 27]. Algorithms that we propose in this chapter are based on single linkage hierarchical clustering. Single
linkage has been shown to perform better, from a time complexity perspective, over complete linkage and average linkage [109]. An analysis on different linkages in hierarchical clustering can be found in [101].

2.3.2 Our Approaches

Naïve algorithms for record linkage take $O(n^2 L^2)$ time where $n$ is the number of records and $L$ is the maximum length of any record. The length of any record is nothing but the total aggregated length of all the attributes employed in the record linkage analysis. When the data size is very large, these algorithms take a very long time. Thus it was an important open problem to devise faster algorithms. To make record linkage process faster and more reliable, we propose a very fast sequential algorithm and a parallel algorithm.

Sequential Algorithm

The proposed algorithm is independent of the number of data sets. Thus, we are able to integrate data from any number of data sets in an elegant way. It is true that any algorithm that links two data sets can be employed to integrate more than two data sets by invoking the algorithm multiple times each time integrating two. For example, if we have three data sets $A$, $B$, and $C$, we can first merge $A$ and $B$ to get $A'$ and then merge $A'$ and $C$. However, the output and accuracy of this approach will depend on the order in which these pairwise merges are done. In our sequential algorithm called RLA-SL (Record Linkage Algorithm - Single Linkage), we collect all the records from all the data sets and form a collection $X$ and sort $X$ after concatenating some or all of the common attributes (such as first name, last name, gender, address, etc.) in each record. Using this sorted list exact duplicates are eliminated. Two records are treated as identical if they agree on the common attributes. Note that in any record linkage algorithm record distances are calculated using only these common attributes. Let $X'$ be the set of records remaining after the elimination of duplicates. Clustering is performed on $X'$. We use blocking on $X'$ based on $l$ characters of the last names (for some suitable value of $l$). Blocking may be done on last name, first name or any other relevant attribute. In our experiments on real datasets we have realized that the use of last names yields the best accuracy. Each block
consists of records that share an l-mer (i.e., a substring of length l) in the last names. An l-mer is also referred to as an l-gram in the literature. Two records \( r_1 \) and \( r_2 \) will be in the same block if they share at least one l-mer in their last names. Since a record might share an l-mer with many other records, it could be in many different blocks. If \( q \) is the maximum number of blocks that a record is in and if \( n' \) is the number of records in \( X' \), then the expected size of each block is \( \frac{qn'}{26^l} \), assuming the English alphabet. Single linkage and edit distance are used for the clusters and records, respectively. Instead of constructing the entire dendrogram, we utilize a threshold \( \tau \) (an input parameter) to generate a partial dendrogram that has only edges with distances no more than \( \tau \). Then a graph \( G(V, E) \) is generated in which \( V \) is \( X' \). Two nodes in \( V \) have an edge between them if and only if they are in the same cluster of the partial dendrogram from some blocking. Thus, each connected component of \( G \) contains the records pertaining to one individual.

The most time-consuming part of the proposed algorithm is the calculation of linkages between records in blocks to generate the graph \( G(V, E) \). Let \( b \) be the number of blocks in \( X' \), \( b_a \) be the average number of records in a block, \( L \) be the maximum length of a record, \( n' \) be the number of records in \( X' \), and \( \tau \) be the threshold on the distance. The time complexity of Algorithm 2 (steps 3 through 7) is \( O(bb_a^2L\tau) \). In practice we have noted that \( bb_a = O(n') \) and hence it takes \( O(n'baL\tau) \) time for steps 3 through 7. Clearly, the smaller the value of \( n' \) the better will be the run time. Steps 1 and 2 of Algorithm 2 take time that is linear in the size of \( X \). We refer to the average number of (identical) duplicates we have for each record as multiplicity. Another prominent idea we have applied is to cache misses. As the cache memory of each processor is limited and most of the times it is not enough to hold all the records, cache misses occur frequently. We handle this issue by copying frequently needed data into a separate array so that these data will be in contiguous memory locations. TPA (FCED) consumes a considerable amount of time in removing duplication of linkages. We have cut this amount of time by considering a graph-based solution where we find connected components in linear time.
Algorithm 2 RLA-SL (Record Linkage Algorithm using Single Linkage Clustering)

Input: A set of data sets consisting of lists of records

Output: A list of clusters of records

1: procedure RLASL
2: Collect all records from all the data sets and form a single list $X$.
3: Sort the records in $X$ and form groups such that each group consists of identical records. Pick one record from each such group and let $X'$ be the resultant collection of records.
4: Do blocking on $X'$. Specifically, there could be a block for every possible $l$-mer. (Note that there are possible $l$-mers when the alphabet corresponds to English.) Consider one such $l$-mer $y$. If two records have $y$ as an $l$-mer in their last names then these two records will be in the block corresponding to $y$. If there is an $l$-mer $y'$ that does not occur in the last name of any record, then the block corresponding to $y'$ will be empty. Also, the same record could be in many different blocks. So a record is going to be in $(L-l+1)$ blocks where $L$ is the length of this record and the blocking size is $l$.
5: Cluster every block obtained in Step 3. Employ hierarchical clustering with single linkage. Specifically, two records $r_1$ and $r_2$ will belong to the same cluster if the distance between them is no more than $\tau$. We have employed a fast algorithm for computing the edit distance between two records. This algorithm, also used in [102], takes $O(\tau k)$ time where $k$ is the minimum of the two record lengths and $\tau$ is the specified threshold [102].
6: We generate a graph where $V$ is the collection $X'$. Two records have an edge between them if there exists at least one cluster in at least one block in which both of these records belong.
7: Find the connected components of $G(V, E)$.
8: Output each connected component as a cluster. While outputting a connected component, also output records that are identical to records in the component. (Note that information about identical records is available from step 2).
9: end procedure
**Parallel Algorithm**

We have parallelized the sequential algorithm (Parallel Record Linkage Algorithm, or PRLA-SL), which achieves nearly linear speedups. We keep a copy of the input list $X$ with each processor. One of the processors is identified as the master and the other processors are called slaves. Let $p$ be the number of slaves. The steps in the algorithm are enumerated below.

**Algorithm 3 PRLA-SL (Parallel Record Linkage Algorithm using Single Linkage Clustering)**

**Input:** A set of data sets consisting of lists of records, Number of processors  
**Output:** A list of clusters of records

1: **procedure** PRLASL  
2: The master broadcasts all the input records to the slave processors.  
3: Each processor sorts a portion of $X$ in parallel. Specifically, the records of $X$ are grouped based on the first two characters of the last names. Note that there are $26^2$ possible 2-mers of characters and hence there are these many possible groups (some of which could be empty). Each processor sorts $\frac{26^2}{p}$ groups. As a by-product of this sorting, each processor picks a representative from every group of identical records that it sorted. In other words, we form $X'$. The slaves inform the master about their findings.  
4: The master assigns $\frac{|X'|}{p}$ number of records from $X'$ to each processor for the purpose of blocking. Each processor then performs blocking on its records and sends the blocks information to the master.  
5: The master aggregates the blocks. In particular, let $y$ be some possible $l$-mer. Parts of the block corresponding to $y$ could be with multiple processors. The master merges these partial blocks.  
6: Let $B_1, B_2, \ldots, B_t$ be the blocks in $X'$. Note that $t \leq 26^l$, where $l$ is the blocking size. Let $n_i = |B_i|$, for $1 \leq i \leq t$. The master sorts $n_1^2, n_2^2, \ldots, n_t^2$ values in descending order. Let $s = \sum_{i=1}^{t} n_i^2$. The master then distributes the blocks among the processors so that the work assigned to each processor is nearly even. Specifically, the distribution is such that the sum of squares of block sizes assigned to any processor is nearly $\frac{s}{p}$.  
7: The next task is to generate the graph $G(V,E)$. To do this, each processor finds the edges in its blocks along the same lines as in the sequential algorithm. All of these edges from all the processors are sent to the master.  
8: The master finds the connected components in the graph. These connected components together with the initially removed copies of records yield us the clusters of interest.  
9: **end procedure**

Let $n$ be the number of records and $n'$ be the number of distinct records in the input. Let $L$ be the maximum length of any record in the input.

In step 2 the broadcasting takes $O(n)$ time. Grouping in step 3 can be done by sorting
the records based on two characters and hence this sorting step takes $O(n)$ time as well. Once the groups are formed (based on two characters), we can expect each group to have $\frac{n}{26^2}$ records and hence the sorting of groups takes an expected $O(\frac{n}{p})$ steps. The communication of the slaves with the master takes $O(n)$ time.

In step 4, the master sends a subset of $X'$ to each of the slaves. This communication takes $O(n')$ time. If $l$ is the blocking size, then, each processor spends $O(\frac{n'}{p}(L - l + 1))$ time in forming the blocks. Note that there will be a total of $26^l$ blocks. Each slave sends the master information about its blocks. In particular, for every block it sends a list of indices of all the records that belong to this block. As a result, the amount of information sent from each slave to the master is $O(\frac{n'}{p}(L - l + 1))$. Therefore, the total communication time in this step is $O(n'(L - l + 1))$.

In step 5, aggregation of the blocks received from all the slaves in step 4 is done in $O(n'(L - l + 1))$ time by the master. Then a sorting is done on the list of sizes of the blocks. This takes $O(26^l)$ time using radix sort.

In step 6, the blocks are distributed among the slaves such that the value of $s$ is nearly balanced across the slaves. Note that this problem is NP-complete. We use the sum of squares of block sizes to compute $s$ for the following reason. To compute the edges within each block, in the worst case, each record is compared with every other record. As a result, the worst case time spent on each block is proportional to the square of the block size. We have tried several ways of distributing the blocks. In each of these ways, a block might get split between two adjacent processors to ensure a close partitioning. Therefore, each of the techniques we have employed does not guarantee an exactly even partitioning (or an optimal partitioning). One simple partitioning we have used is to use the sorted list $Q = n_1^2, n_2^2, \ldots, n_t^2$. We will identify a minimum prefix of this sequence whose sum equals or exceeds $\frac{s}{p}$. If this prefix sum equals $\frac{s}{p}$, then this prefix sequence of blocks will be assigned to the first processor. If this prefix sum exceeds $\frac{s}{p}$, then the last block in this prefix sequence will be split between the first and the second processors. The splitting will be done to ensure that the work assigned to the first processor is as close to $\frac{s}{p}$ as possible.

By the work assigned to a processor we mean the sum of squares of the blocks assigned to the processor. In the case of the prefix sum exceeding $\frac{s}{p}$, a portion of the last block in
this prefix sequence will be assigned to the second processor. The second processor will also be assigned the next some number of blocks in the sorted sequence \( Q \). This number of blocks will be such that the work assigned to this processor is nearly \( \frac{s}{p} \), and so on. The time taken by the master in step 6 is \( O(t) \) where \( t \) is the number of blocks. If the blocking size is \( l \), then \( t \leq 26^l \). After this, the master creates a list of records for each slave to work on. This takes \( O(n'(L - l + 1)) \) time. Subsequently, the master sends the individual lists to the slaves. This communication also takes \( O(n'(L - l + 1)) \) time.

In step 7 each processor works on its blocks. The time spent in this step is \( O(\frac{s}{p}) \). Note that the expected size of each block is \( \frac{n'(L-l+1)}{26^l} \). Also, the time spent in computing the distance between any two records is \( O(\tau L) \). Thus the expected value of \( s \) is \( \frac{(n')^2(2L-l+1)^2}{26^l} \). Our empirical results indicate that the total number of edges generated across all the processors is \( O(n'(L - l + 1)) \). In this case, the communication time is \( O(n') \). As a result, the connected components in step 8 can also be found in \( O(n'(L - l + 1)) \) time.

In summary, the total expected run time of the algorithm is \( O(n + n'(L - l + 1) + \frac{(n')^2(2L-l+1)^2}{26^l} \tau L) \). It turns out that the last term is the dominating one among the three terms in this time complexity. Table 2.4 explains why we get a speedup that is close to linear. Please note that blocking is quite useful in reducing the run time. For example, even if \( L = 15 \), for a value of \( l = 3 \), the value of is 0.0096\((n')^2\).

Also, the run times of most of the (sequential and parallel) algorithms found in the literature depend on \( n^2 \). Thus the work done by our algorithm is expected to be significantly better than competing algorithms since our run time depends on \( (n')^2 \). In practice the value of \( (n')^2 \) is much smaller than that of \( n^2 \). Although parallel algorithms exist (see e.g., [40]) for finding connected components, we have not used them here since the time needed for this step is very small.

2.4 Results

We have implemented our sequential version for simulated data in C++ to make a better comparison with parallel version, as PRLA-SL has been implemented using MPI with C++. We have also used C++ implementation of the TPA (FCED) algorithm to compare
with our sequential version. As TPA (FCED) was originally implemented in java, we have also implemented our algorithm in java to make a fair comparison with the results in [102]. Our sequential algorithm outperforms TPA (FCED) [102] especially when the multiplicity is large.

We have tested our algorithms on both synthetic and real data. We have collected real data sets from the Connecticut Health Information Network (CHIN). As TPA (FCED) [102] ensures very high accuracy of record linkage but consumes a large amount of time, our main purpose was to provide a much faster solution. So we have developed our algorithms in such a way that the accuracy remains the same, but the algorithms run much faster. In the blocking phase, we have used 4-mer for all the experiments. The value of $l$ in the blocking phase has to be chosen carefully. If $l$ is low, the accuracy will be high. A higher value will result in a reduction in the run time but the accuracy might suffer.

2.4.1 Results on Simulated Data for The Sequential Algorithm

The implementation has been deployed in the HORNET cluster housed in the Booth Engineering Center for Advanced Technology (BECAT), University of Connecticut. This cluster has 64 nodes, each of which has 12 Intel Xeon X5650 Westmere cores, 48 GB of RAM and 500 GB of local storage.

Running time of our algorithms is independent of the number of datasets as we add all the records to a single list and work with only this list. Like in TPA (FCED) [102], we have employed both constant and proportional threshold values in the clustering step. Our algorithm has been tested for each type of distance calculation. Total number of records used for this test ranges from 50,000 to 5,000,000 to reveal the power of our algorithm. Five records have been generated for each individual, in which 4 are error free. So, on the five records of any individual, exact clustering will find 2 clusters.

To compare with TPA (FCED), we employ edit distances of two attributes, namely the first name and the last name. TPA (FCED) spends around 650.49 sec for 1,000,000 data whereas our algorithm takes only 92.99 sec, which is seven times faster for this amount of data. Table 2.1 summarizes the comparison. Figure 2.1 provides a graphical representation of this comparison.
Table 2.1: Comparison of results on simulated data

<table>
<thead>
<tr>
<th>Number of records</th>
<th>Algorithm</th>
<th>Run time in seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>50,000</td>
<td>TPA (FCED)</td>
<td>7.35</td>
</tr>
<tr>
<td></td>
<td>RLA-SL</td>
<td>1.19</td>
</tr>
<tr>
<td>100,000</td>
<td>TPA (FCED)</td>
<td>24.81</td>
</tr>
<tr>
<td></td>
<td>RLA-SL</td>
<td>3.67</td>
</tr>
<tr>
<td>200,000</td>
<td>TPA (FCED)</td>
<td>71.47</td>
</tr>
<tr>
<td></td>
<td>RLA-SL</td>
<td>10.25</td>
</tr>
<tr>
<td>400,000</td>
<td>TPA (FCED)</td>
<td>178.74</td>
</tr>
<tr>
<td></td>
<td>RLA-SL</td>
<td>26.09</td>
</tr>
<tr>
<td>600,000</td>
<td>TPA (FCED)</td>
<td>324.82</td>
</tr>
<tr>
<td></td>
<td>RLA-SL</td>
<td>45.99</td>
</tr>
<tr>
<td>800,000</td>
<td>TPA (FCED)</td>
<td>489.43</td>
</tr>
<tr>
<td></td>
<td>RLA-SL</td>
<td>68.67</td>
</tr>
<tr>
<td>1,000,000</td>
<td>TPA (FCED)</td>
<td>650.49</td>
</tr>
<tr>
<td></td>
<td>RLA-SL</td>
<td>92.99</td>
</tr>
<tr>
<td>2,000,000</td>
<td>TPA (FCED)</td>
<td>1844.52</td>
</tr>
<tr>
<td></td>
<td>RLA-SL</td>
<td>256.51</td>
</tr>
<tr>
<td>3,000,000</td>
<td>TPA (FCED)</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>RLA-SL</td>
<td>490.54</td>
</tr>
<tr>
<td>4,000,000</td>
<td>TPA (FCED)</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>RLA-SL</td>
<td>800.02</td>
</tr>
<tr>
<td>5,000,000</td>
<td>TPA (FCED)</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>RLA-SL</td>
<td>1123.85</td>
</tr>
</tbody>
</table>

Table 2.2: Analysis of results on simulated data (RLA-SL)

<table>
<thead>
<tr>
<th>Number of Records</th>
<th>Number of Exact Clusters</th>
<th>Number of Clusters</th>
<th>Exact Cluster Time</th>
<th>Approx Cluster Time</th>
<th>Merge Time</th>
<th>Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>50000</td>
<td>19582</td>
<td>12130</td>
<td>0.21</td>
<td>0.95</td>
<td>0.03</td>
<td>1.19</td>
</tr>
<tr>
<td>100000</td>
<td>39201</td>
<td>23965</td>
<td>0.48</td>
<td>3.11</td>
<td>0.08</td>
<td>3.67</td>
</tr>
<tr>
<td>200000</td>
<td>78453</td>
<td>46487</td>
<td>1.11</td>
<td>8.97</td>
<td>0.17</td>
<td>10.25</td>
</tr>
<tr>
<td>400000</td>
<td>156934</td>
<td>88725</td>
<td>2.71</td>
<td>23.04</td>
<td>0.34</td>
<td>26.09</td>
</tr>
<tr>
<td>600000</td>
<td>232866</td>
<td>130746</td>
<td>4.67</td>
<td>40.74</td>
<td>0.58</td>
<td>45.99</td>
</tr>
<tr>
<td>800000</td>
<td>309615</td>
<td>173617</td>
<td>6.52</td>
<td>61.42</td>
<td>0.73</td>
<td>68.67</td>
</tr>
<tr>
<td>1000000</td>
<td>387707</td>
<td>214912</td>
<td>8.8</td>
<td>83.21</td>
<td>0.98</td>
<td>92.99</td>
</tr>
<tr>
<td>2000000</td>
<td>771004</td>
<td>427269</td>
<td>27.21</td>
<td>227.31</td>
<td>1.99</td>
<td>256.51</td>
</tr>
<tr>
<td>3000000</td>
<td>1154323</td>
<td>639501</td>
<td>45.07</td>
<td>442.46</td>
<td>3.01</td>
<td>490.54</td>
</tr>
<tr>
<td>4000000</td>
<td>1537531</td>
<td>851729</td>
<td>61.69</td>
<td>732.89</td>
<td>5.44</td>
<td>800.02</td>
</tr>
<tr>
<td>5000000</td>
<td>1920723</td>
<td>1064825</td>
<td>77.14</td>
<td>1041.28</td>
<td>5.43</td>
<td>1123.85</td>
</tr>
</tbody>
</table>

33
When the input data contains a large number of records, TPA (FCED) spends too much time to complete. Table 2.2 displays the time taken by RLA on various steps. When we have 1,000,000 records, finding clusters using exact matching (steps 3-6 in the sequential algorithm) takes only 8.8 sec. The size of $X'$, after removing duplicates, is only 387,707. From Table 2.1 we see that TPA (FCED) takes around 178.74 sec to find clusters for 400,000 records. But RLA clusters 387,707 records by approximate clustering within 83.21 sec. This improvement is because of the graph-based solution and the avoidance of cache misses. So clustering of 1,000,000 records takes only 92.99 sec. Even when the multiplicity is 1, our algorithm runs around two times faster than TPA (FCED). Since in practice the multiplicity of data is more than 1, our algorithms run much faster as shown in Figure 2.1. Our proposed algorithm is more than 20 times faster than the previous algorithm TPA (FCED) on the datasets of records having a multiplicity of 5. Figure 2.2 is the graphical representation of Table 2.2.

A similar experiment, which uses reversal edit distance, also shows superiority of the RLA algorithm. Reversal edit distance takes in two groups of attributes, calculates edit distance in both original direction and reversal direction, and returns the smaller one. In our experiments, we aggregate the edit distance of the first attributes of the two records and the edit distance of the second attributes of them. Again we add the edit distance between the first attribute of the first record and the second attribute of the second record and the edit distance between the second attribute of the first record and the first attribute.
of the second record. We then take the smaller of these two distances, as this is the reversal distance value. Figure 2.3 shows almost the same efficiency for RLA on this distance as well.

But in this case, both the algorithms take more time than for the previous distance calculation as two edit distances are needed to be calculated as per the definition of reversal distance.

We have performed another experiment using edit distance as the distance method but adding a parameter namely truncation count. We have used a truncation count of 2, which means that we only employ the first 2 characters of any attribute concerned. Both the algorithms produce more clusters in this case. The process is slow since more linkages will have to be dealt with. Figure 2.4 shows the comparison.
In the above cases, we have used constant threshold to find clusters. The next test shows results for using proportional threshold, which is dependent on the length of the considered attributes. Results are shown in Figure 2.5.

Proportional threshold sometimes works better as it is dependent on the data. We omit details on the proportional threshold, as the procedure is similar to the constant threshold.

Clearly, the threshold has a great impact on the accuracy of clusters as a too small or too large threshold will normally yield a low error-rate. That is why a training phase is needed to learn the threshold.
2.4.2 Results on Real Data for The Sequential Algorithm

Our experiments on real data have been conducted on the Connecticut Health Information Network (CHIN) server for security reasons. The computer has a CPU of Intel(R) Xeon(R) X5460 3.16 GHz and 4 GB RAM. The data come from 4 different datasets having a total of 1,083,878 records.

Table 2.3 shows the comparison. RLA employs two attributes, namely the first name and the last name. Within 15 sec, it outputs 112,404 exact clusters. The rest of the steps take around 19 sec. The algorithm terminates within 34.5 sec whereas TPA (FCED) spends around 2961 sec. RLA is 85 times faster than TPA (FCED) for this real data. The accuracy is 93.0% for both.

Table 2.3: Results on real datasets (1,083,878 records)

<table>
<thead>
<tr>
<th>Number Of Attributes</th>
<th>Algorithm</th>
<th>Time in seconds</th>
<th>Created Clusters</th>
<th>Correct Clusters</th>
<th>Number Of Individuals</th>
<th>Accuracy %</th>
<th>Com. %</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>TPA (FCED)</td>
<td>2961</td>
<td>94381</td>
<td>87756</td>
<td>108800</td>
<td>93.00%</td>
<td>80.70%</td>
</tr>
<tr>
<td></td>
<td>RLA-SL</td>
<td>34.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>TPA (FCED)</td>
<td>3402</td>
<td>101864</td>
<td>99562</td>
<td>108800</td>
<td>97.80%</td>
<td>91.60%</td>
</tr>
<tr>
<td></td>
<td>RLA-SL</td>
<td>48.7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

We have also used date of birth attribute in addition to the above two attributes. The running time is also impressive. RLA takes only 48.7 sec whereas TPA (FCED) takes 3402 sec. In this case, RLA is 70 times faster. 97.8

2.4.3 Results on Simulated Data for The Parallel Algorithm

In our experiments, we have used at most 32 cores from 4 nodes, 8 from each node. In this case, we have used another set of synthetic data, in which the multiplicity is nearly 1.

An algorithm is fully parallel when the speedup is linear. We have optimized our algorithm to make it almost linear. Table 2.4 analyzes the running time of PRLA for 6 million records. The first column shows the number of cores used. The total time spent in broadcast operations that take place in steps 1, 3 and 5 is shown as bcast. The total time for the other communications that happen in step 2, step 3, step 4, and step 6 is shown
as comm. As we can readily see, these communication overheads are very low. Master performs certain tasks on its own in steps 3, 4, 5, and 7. This total time is displayed as master in Table 2.4. The time for sorting and finding duplicates in step 3 is dedup. The total time for Blocking (block, in step 4), merging (merge, in step 5), distribution of blocks (dist, in step 6) and finding connected components (concomp, in step 8) is very low as well. Generating edge lists is the major time consuming step. This time is shown as edgelist. The fact that this step dominates the entire run time is also revealed in our time complexity analysis above. The first row, seq, shows the runtime consumed by sequential RLA. Figure 2.6 graphically describes the data in Table 2.4.

![Figure 2.6: Analysis of results on synthetic data using PRLA (Y axis denotes time in seconds; X axis corresponds to number of processors)](image)

The time results are also shown in Figure 2.7. The x axis represents the number of cores used and the y axis shows time in seconds.

Our results show that the speedup is around 7.5 for 8 cores (that reside in a single node), 14.1 for 16 cores (residing in 2 nodes) and 26.4 for 32 cores. Values show almost

<table>
<thead>
<tr>
<th>pr</th>
<th>bcast</th>
<th>comm</th>
<th>master</th>
<th>dedup</th>
<th>block</th>
<th>merge</th>
<th>dist</th>
<th>edgelist</th>
<th>concomp</th>
<th>total time</th>
<th>speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>3.95</td>
<td>74.44</td>
<td>4.07</td>
<td>0.26</td>
<td>50.22</td>
<td>4719.8</td>
<td>0.69</td>
<td>4853.5</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0.06</td>
<td>0.8</td>
<td>3.75</td>
<td>54.56</td>
<td>2.21</td>
<td>0.16</td>
<td>49.15</td>
<td>2329</td>
<td>0.6</td>
<td>2440.3</td>
<td>1.99</td>
</tr>
<tr>
<td>4</td>
<td>0.1</td>
<td>0.38</td>
<td>3.19</td>
<td>21.3</td>
<td>1.19</td>
<td>0.11</td>
<td>19.68</td>
<td>1220.9</td>
<td>0.54</td>
<td>1267.4</td>
<td>3.84</td>
</tr>
<tr>
<td>8</td>
<td>0.23</td>
<td>0.34</td>
<td>2.96</td>
<td>12.04</td>
<td>0.7</td>
<td>0.08</td>
<td>10.03</td>
<td>622.18</td>
<td>0.54</td>
<td>649.1</td>
<td>7.5</td>
</tr>
<tr>
<td>16</td>
<td>2.56</td>
<td>1.99</td>
<td>2.76</td>
<td>8.35</td>
<td>0.4</td>
<td>0.07</td>
<td>2.72</td>
<td>325.58</td>
<td>0.67</td>
<td>345.1</td>
<td>14.1</td>
</tr>
<tr>
<td>32</td>
<td>3.48</td>
<td>2.08</td>
<td>2.52</td>
<td>5.73</td>
<td>0.21</td>
<td>0.06</td>
<td>0.71</td>
<td>169</td>
<td>0.53</td>
<td>184.3</td>
<td>26.4</td>
</tr>
</tbody>
</table>

Table 2.4: Distribution of running time for 6,000,000 records
Figure 2.7: Results on simulated data (for 6 million and 9 million records, Y axis denotes time in seconds; X axis corresponds to number of processors)

linearity in speedup (Figure 2.8). We have tested on 1, 2, 4, 8, 16 and 32 cores.

Figure 2.8: Speed up (for 6 million and 9 million records, Y axis denotes speed up; X axis corresponds to number of processors)

2.5 Discussion

Our algorithms ensure the same accuracy as the previous algorithm TPA (FCED). Accuracy and completeness have been calculated on real dataset. Social Security Number or DDS identification number was available for these records that we utilized for calculating the accuracy. These numbers were revealed to us only after our algorithms produced the results.

To cluster records more accurately, an appropriate threshold value is necessary. Such a threshold can be obtained in a learning process as described in [102]. The idea is to have
a training phase in which records for which the right clustering is known will be utilized. The whole procedure is described elaborately in [102]. We have used a constant threshold value of 1 and a proportional threshold value of 0.1.

Besides using edit distance, we have also employed reversal edit distance and truncation distance. A common error occurring in records is the reversal of the first and last names. In these cases, reversal edit distance will yield better results. Truncation distance is used when a specific portion of records is sufficient for determining the clusters. All of these distance calculations make our algorithms versatile.

We did experiment on four real datasets of total size 1,083,878 records. Two datasets came from university of Connecticut’s Dental Clinic (UCHC) and two from the Connecticut Department of Development services (DDS).

To generate simulated data, we collected 200,000 records of dead people from ssdmf.info. Each record has SSN, last name, first name, middle name, date of death and date of birth attributes. Then we introduced 2-3 new characters in the first name or last name for 90% of the records. For the others, we have altered 1-3 characters of the first name or last name. We have thus generated 1,000,000 records. Then we replicated the file three times. We also generated another 8 datasets of 1,000,000 records introducing errors using the above procedure.
Chapter 3

Efficient record linkage algorithms using complete linkage clustering

3.1 Introduction

Health agencies keep track of patients’ health information and at the same time records of a patient reside in multiple data sources. All the records of a patient may be needed to accurately diagnose a disease or prescribe medicine for a disease for the patient [86, 135]. Disease evolution, drug discovery and side effects of a drug may require analysis of health records across these data sources [22, 35]. Record linkage, for example, can be used to merge records across educational databases, employment history, and family evolution to analyze an individual’s characteristics. It has also applications in similarity detection in digital documents [10, 125], master data management [146], social networking [120], historical research [141], gene expression [49, 46, 52, 29], information science [136], health psychology [23], data mining [142, 107], etc.

Record linkage [137] integrates records across multiple data sources as well as identifies records pertaining to same individuals. Now-a-days millions of records are stored and maintained in data sources electronically. Connections among these records provide better understanding of relationships of these data sources. Exact same records exist in multiple databases. Sometimes records get polluted unintentionally due to typing error, similarity
in pronunciation, etc. All of these issues make the record linkage problem very challenging and critical. Efficient algorithms are inevitable to address this problem.

Fortunately, a large number of algorithms are available in the real world\cite{72}. A naïve algorithm compares each pair of records to find matches. It may produce expected results but has a high time complexity. Therefore algorithms have been devised to provide best possible results within a manageable time. We have previously proposed single linkage hierarchical clustering based solutions\cite{89} for this record linkage problem. These algorithms provide very fast solutions in finding clusters of individuals with a high accuracy.

In this research work we propose a complete linkage hierarchical clustering based solution for this problem. Single linkage solution works fine for real life applications. But it has a chaining problem. We discuss the problem elaborately in this chapter. Our newly devised algorithms not only solve this chaining problem but also assure expected output. We also develop an efficient parallel version of this algorithm. Our experimental results substantiate our claim.

### 3.2 Background and Significance

Record linkage\cite{18, 41} identifies record matches across different data sets even if they have no universal identifier. The problem is to group similar records so that each group contains all records of one individual only. This problem is no more than trivial if the records do not get contaminated. Often errors are introduced unintentionally while typing, due to sound similarity, etc. Every group of similar records can be thought of as a cluster. Every cluster should contain only the records of a single person and it should contain all the records of this person. Several types of clustering algorithms such as $k$-means clustering, fuzzy clustering, hierarchical clustering, graph-based clustering, etc. are widely available\cite{55}. Our proposed algorithms are based on hierarchical clustering\cite{59}. This requires linkage criteria that define how distances are measured between any two clusters. Single linkage and complete linkage clustering are popular in use. In single linkage, the distance between two clusters $A$ and $B$ is computed as the minimum distance between a point (i.e., a record) in $A$ and a point in $B$. In complete linkage, the distance between two clusters $A$ and $B$
is computed as the maximum distance between a point in $A$ and a point in $B$. Therefore single linkage clustering can be thought of as the nearest neighbor clustering and complete linkage clustering can be thought of as the farthest neighbor clustering. In addition to defining the distance between two clusters, we also have to define the distance between two records. There are many distance measures that can be used for the records. Edit distance or Levenshtein distance calculates the number of insertions, deletions and substitutions required to transform one string to the other. (We can think of every record as a string of characters). Manhattan distance computes only the number of mismatches. There exist some other distance calculation methods such as Euclidean distance, maximum distance, etc. We have used complete linkage hierarchical clustering for our algorithms. These algorithms generally use edit distance, reversal edit distance and truncation edit distance calculation methods although our algorithms can support any distance measure. Reversal edit distance and truncation edit distance also use edit distance calculation methods.

### 3.3 Related Works

A naïve or brute force algorithm compares every pair of records and hence takes too much time. There exist a large number of efficient algorithms\cite{31, 139}. \cite{12, 82} define data cleansing and record linkage. They also present a literature survey for many proposed or developed methodologies for entity resolution and record linkage. A relational clustering algorithm uses both attribute and relational information to integrate entities\cite{6}. Discussions about deduplication quality and data linkage measurement involve different linkage processes and issues\cite{21}. Limitations in record linkage algorithms have also been discussed in the literature\cite{138}. The EMH algorithm (based on expectation maximization) provides better decision rules employing probability estimates\cite{140}. There exist some other probabilistic methods for record linkage problems\cite{36, 34}. A hybrid Markov chain Monte Carlo algorithm calculates transitive linkage probabilities across records and uses this information for post-processing procedures such as logistic regression\cite{128}. Relational probability model can solve the citation-matching problem\cite{113}. Records across multiple data sets may contain variations as well as errors\cite{16}. Edit distance calculation has been used widely to
compute variations between records\textsuperscript{[80]}. Case patient algorithm includes ‘Jaro–Winkler’, ‘Soundex’ and ‘weight matching’ for distance computation\textsuperscript{[25]}. Record linkage has also applications in record matching \textsuperscript{[58]}, text correction \textsuperscript{[74]}, substring matching\textsuperscript{[37]}, etc. Relational dependencies among different fields improve record linkage processes by reducing errors\textsuperscript{[26, 33]}. Conditional models for record linkage problem can handle varieties of features of input data sets independent of their dependencies\textsuperscript{[97, 76]}.

Blocking and indexing have been used extensively for faster computation by removing many unnecessary pair comparisons\textsuperscript{[96, 129, 19]}. Traditional blocking, sorted neighborhood indexing, \textit{Q}-gram-based indexing, suffix array-based indexing, canopy clustering, and string-map-based indexing are popular blocking techniques for reducing comparison space. \textsuperscript{[5]} proposes \textit{Q}-gram fingerprinting as a blocking technique. It transforms records into bit vectors and filters pairs of bit vectors using multibit trees. FEBRL\textsuperscript{[20, 17]}, FRIL\textsuperscript{[62, 61]}, Intelliclean\textsuperscript{[78]} are well-known and widely used record linkage algorithms and tools. FEBRL uses three different indexing methods namely standard blocking method, sorted neighbourhood approach, and \textit{n}-grams. It has a parallel implementation using MPI with python. FRIL is another good tool for record linkage with many options. It employs nested loop join (NLJ) and the sorted neighborhood method as search methods. Hierarchical clustering based solution has been popular for record linkage\textsuperscript{[118, 81, 109, 143]}. Given the exponential growth in data sizes, parallel solutions are inevitable\textsuperscript{[81, 109, 143, 65, 70, 71, 27]}. Some efficient data integration algorithms have shown very high accuracies\textsuperscript{[102, 101]}. Recently developed single linkage hierarchical clustering algorithms outperform these algorithms\textsuperscript{[89]}. Here we propose sequential and parallel record linkage algorithms that use complete linkage clustering. These algorithms offer improved accuracies and have the potential of having a greater impact on real world applications.

### 3.4 Methods

We propose sequential and parallel record linkage algorithms, which use complete linkage hierarchical clustering. These algorithms employ single linkage algorithms\textsuperscript{[89]} as a pre-processing step to generate intermediate clusters. Complete linkage method is applied within
each of these clusters. We employ some post processing steps to fine-tune the clusters thus generated.

### 3.4.1 Sequential Algorithm

RLA-CL (Record Linkage Algorithm – Complete Linkage) works in several phases and each of these phases consists of possibly multiple steps. Steps involved in RLA-CL are shown in Figure [3.1](#).

![Flow Chart](#)

**Figure 3.1:** A flow chart describing all steps involved in RLA-CL.
RLA-CL first sorts the records and identifies duplications. As different data sets may have different numbers and types of attributes, it takes pairs of data sets in which one of them has a subset attribute types of the other. Then the algorithm sorts them using efficient radix sort on common attributes. Exact matches will be adjacent in the sorted array. We do this sorting for each pair of data sets meeting our required criteria. We accumulate all of them and eliminate duplicates by merging them into the same clusters. This single phase removes many records from further consideration and shrinks the data sets. A simple example may simplify the working process of this phase. Let $A$, $B$, and $C$ be three input data sets. $A$ has $a, b, c,$ and $d$ as attribute fields, $B$ has $a$ and $d$ and $C$ has $a, d,$ and $e$. Note that the attributes in $B$ form a subset of the attributes in $A$. Also, the attributes in $B$ form a subset of the attributes in $C$. We sort $A$ and $B$ together; we also sort $B$ and $C$ together. The attributes in $A$ form neither a subset nor a superset of the attributes in $C$. Thus we do not sort $A$ and $C$ together. After sorting the records of $A$ and $B$ data sets, we accumulate duplicate records into clusters. We do the same process for $B$ and $C$ data sets. Then the algorithm merges these two arrays of clusters obtained from merging $A$ with $B$ and $B$ with $C$ data sets. This exact matching phase identifies all possible duplicates and unifies them into clusters. Therefore the remaining phases of the algorithm have to handle these reduced data sets only, which form a subset of the initial data sets. We have shown this phase in Algorithm 4.

Algorithm 4 Exact Cluster Finding Algorithm

**Input:** A list of records
**Output:** A set of clusters of identical records

```
1: procedure FINDEXACTCLUSTERS
2:     for each pair data set \{X, Y\} do
3:         if attributes$_X$ $\subseteq$ attributes$_Y$ then
4:             Combine records from X and Y
5:             Sort lexicographically using radix sort
6:             Merge duplicate records by creating clusters of identical records
7:             Remove duplicate records
8:         end if
9:     end for
10:    Merge clusters generated from all pairs
11: return the set of exact matched clusters
12: end procedure
```
Exact matching results in clusters of records. From each cluster we pick only one representative for further processing. In this way we make our algorithms independent of the number of input data sets and they can identify similar records within a data set as well as across different data sets.

Comparison between every pair of records is time consuming and impractical. Blocking helps to reduce the number of pairs to be compared. We employ \( k \)-mers or \( k \)-substrings of an attribute for blocking. If the blocking attribute contains only English letters, numbers, or alphanumeric values, then we consider only \( 26^k \), \( 10^k \), or \( 36^k \) blocks, respectively. Each block has only those records having at least one \( k \)-mer of the blocking field in common. If \( l \) is the length of the attribute of a record, then this record goes to \( (l - k + 1) \) blocks. If two records belong to the same person and if an attribute slightly differs in these two records, then there is a good chance that the two attribute instances will still have a common \( k \)-mer and hence the two records will fall into at least one block together. We measure distances among records in each block. We generally employ edit distance, reversal distance and truncation distance calculation methods although every suitable distance calculation works perfectly with our algorithm. Edit distance calculates the minimum number of insertions, deletions and substitutions of characters needed to change one string to the desired one. If \( S_1 = \) “algorithmss” and \( S_2 = \) “algorithms”, then we can convert \( S_1 \) to \( S_2 \) using the following operations: insert ‘o’ at index 3 of \( S_1 \), replace ‘l’ to ‘t’ at index 5 and delete ‘s’ from index 9 of \( S_1 \). This algorithm discards many calculations by checking when the distance surpasses the user-defined threshold value. Therefore we have to choose a suitable threshold value dependent on our input data accuracy. Threshold value defines the maximum number of errors allowed in the input records. For the above example if the threshold value is not less than 3, then the algorithm integrates them into a single cluster. These steps have been shown in Algorithm 5.

If records are considered as vertices and distances not above threshold value as edges, then we get an undirected graph. We remove multi-edges between pairs of vertices and self-loops to convert it into a simple graph. We find all the connected components of the graph. These connected components are intermediate clusters generated by single linkage clustering method. This is the third phase of our algorithm.
Algorithm 5 Single Linkage Clustering Algorithm

Input: A set of exact matched clusters and a threshold value

Output: A set of single linkage clusters

1: procedure COMPUTE_SINGLE_LINKAGE_CLUSTERS
2:   Take a record from every exact matched cluster as a representative
3:   In the next steps by a record we mean a representative record
4:   for each attribute in a user defined attribute list do
5:     Create blocks of records sharing the same $k$-mer
6:     for each block do
7:       Consider a graph where records are vertices and
8:         connections among them are edges
9:       Connect two vertices if the distance (edit distance is one of the distance
         calculation methods) between them is at most the user defined threshold value
10:   end for
11: end for
12: Remove multi-edges and self loops to make the graph simple
13: Find connected components of this graph
14: return the set of connected components in the above graph
15: end procedure

Our next phases work on only records within each cluster. Every cluster typically contains a small number of records integrated by single linkage clustering. Single linkage clustering often traps in a chaining problem. Let $A$, $B$, and $C$ be records, where $A = "sweat,exercise,gymnesium"$ having status, type, and place as attributes, $B = "sheat,gymnesium"$ with status and place as attributes and $C = "heat"$ having status as the attribute. Let the threshold value be 1. Therefore $A$ and $B$ are in one cluster, and $B$ and $C$ are in another cluster, but the distance between $A$ and $C$ is 2, which is above the threshold value. According to our first three phases all the three records should be considered in the same cluster. Complete linkage removes this problem. It may merge $A$, $B$ in a cluster and $B$, $C$ in another cluster or $A$, $B$ in a cluster and $C$ in another cluster, and so on. It never merges $A$, $B$ and $C$ in a single cluster.

The fourth phase starts with considering every record in a cluster as a cluster having only one record. Then the algorithm measures distances among each pair of clusters and populating them in 2-$d$ matrices. From these distances we generate a vector having minimum distances from every single record cluster. The algorithm finds the minimum of them, and if this minimum distance is not above the threshold value, then it merges these two clusters into one cluster and updates the distance matrix and vector. When we calculate
the minimum distance for a cluster, we measure distances of the furthest elements between every pair of clusters having this cluster at one side and take the minimum of them. This process continues till the minimum distance does not surpass the threshold value. We eventually get clusters of records of individuals using complete linkage clustering.

Algorithm 6 RLA-CL (Record Linkage Algorithm using Complete Linkage Clustering)

Input: A set of data sets and a configuration file
Output: A set of complete linkage clusters

1: procedure ComputeCompleteLinkageClusters
2: Find exact clusters using Algorithm 1;
3: Compute single linkage clusters by Algorithm 2;
4: for each single linkage cluster do
5: Consider every record of the cluster as a single node cluster
6: Generate a $2 - d$ square matrix where each entry contains the minimum distance between pairs of clusters; Each row of the matrix corresponds to a cluster
7: Generate a vector of minimum distances for each cluster
8: while the matrix has more than 1 row do
9: Merge clusters if the minimum distance between them is no more than the user defined threshold value
10: Update the matrix and vector
11: end while
12: Check whether merging is possible among the generated clusters
13: Use a priority list to resolve ambiguity in finding a perfect cluster for each record;
14: end for
15: Merge these clusters with records from exact matched clusters
16: return these complete linkage clusters
17: end procedure

The fourth phase easily eliminates the problem of merging all the records in a single cluster generated by the chaining phenomenon. But which cluster should contain which records is now a challenging task. We employ a post-processing phase to fine-tune the generated complete-linkage clusters. We require a user-defined priority list of attributes to complete this phase. We assign each priority attribute a score. We take one record from one cluster and check in which cluster it matches the best. The error-free matching with higher priority attributes, clusters having the highest number of priority attributes, etc. determine the destination cluster. This process meets the user-expectations astonishingly in real world applications. Algorithm 6 describes every step of the algorithm.

We can explain the above algorithm using a simple example. Data set $A$ has 3 records {"Cade", "Bale", 05011976}, {"Cade", "Bolt", 05021986}, and {"Thor", "Glenn", 05031996}. The fourth phase easily eliminates the problem of merging all the records in a single cluster generated by the chaining phenomenon. But which cluster should contain which records is now a challenging task. We employ a post-processing phase to fine-tune the generated complete-linkage clusters. We require a user-defined priority list of attributes to complete this phase. We assign each priority attribute a score. We take one record from one cluster and check in which cluster it matches the best. The error-free matching with higher priority attributes, clusters having the highest number of priority attributes, etc. determine the destination cluster. This process meets the user-expectations astonishingly in real world applications. Algorithm 6 describes every step of the algorithm.

We can explain the above algorithm using a simple example. Data set $A$ has 3 records {"Cade", "Bale", 05011976}, {"Cade", "Bolt", 05021986}, and {"Thor", "Glenn", 05031996}. The fourth phase easily eliminates the problem of merging all the records in a single cluster generated by the chaining phenomenon. But which cluster should contain which records is now a challenging task. We employ a post-processing phase to fine-tune the generated complete-linkage clusters. We require a user-defined priority list of attributes to complete this phase. We assign each priority attribute a score. We take one record from one cluster and check in which cluster it matches the best. The error-free matching with higher priority attributes, clusters having the highest number of priority attributes, etc. determine the destination cluster. This process meets the user-expectations astonishingly in real world applications. Algorithm 6 describes every step of the algorithm.

We can explain the above algorithm using a simple example. Data set $A$ has 3 records {"Cade", "Bale", 05011976}, {"Cade", "Bolt", 05021986}, and {"Thor", "Glenn", 05031996}. The fourth phase easily eliminates the problem of merging all the records in a single cluster generated by the chaining phenomenon. But which cluster should contain which records is now a challenging task. We employ a post-processing phase to fine-tune the generated complete-linkage clusters. We require a user-defined priority list of attributes to complete this phase. We assign each priority attribute a score. We take one record from one cluster and check in which cluster it matches the best. The error-free matching with higher priority attributes, clusters having the highest number of priority attributes, etc. determine the destination cluster. This process meets the user-expectations astonishingly in real world applications. Algorithm 6 describes every step of the algorithm.

We can explain the above algorithm using a simple example. Data set $A$ has 3 records {"Cade", "Bale", 05011976}, {"Cade", "Bolt", 05021986}, and {"Thor", "Glenn", 05031996}. The fourth phase easily eliminates the problem of merging all the records in a single cluster generated by the chaining phenomenon. But which cluster should contain which records is now a challenging task. We employ a post-processing phase to fine-tune the generated complete-linkage clusters. We require a user-defined priority list of attributes to complete this phase. We assign each priority attribute a score. We take one record from one cluster and check in which cluster it matches the best. The error-free matching with higher priority attributes, clusters having the highest number of priority attributes, etc. determine the destination cluster. This process meets the user-expectations astonishingly in real world applications. Algorithm 6 describes every step of the algorithm.
12011990}, and data set $B$ has 2 records namely \{"Thor", "Glenn", 12011990\} and \{"Cade", "Balt", 05011976\}. Both of these data sets have first name, last name and date of birth attributes. Let the blocking field be first name; comparing attributes be the first name and last name; the priority field be date of birth; and the threshold value be 1. RLA-CL first accumulates these five records and sorts. It finds four exact matched clusters. Only one cluster \{"Thor", "Glenn", 12011990\}, \{"Thor", "Glenn", 12011990\} has two records having the same first name and last name. Then the algorithm creates blocks on the first name for all of the four representative records. After blocking and constructing linkages, we find 2 clusters. One is \{"Thor", "Glenn", 12011990\}, and the other is \{"Cade", "Bale", 05011976\}, \{"Cade", "Bolt", 05021986\}, \{"Cade", "Balt", 05011976\}. The post processing phase finds an inconsistency: the ”Balt” record may go with the ”Bolt” record or the ”Bale” record since the edit distance value in both cases is 1 and the threshold value is 1. To break this tie, the priority field date of birth helps us to combine the ”Balt” record with the ”Bale” record. After expanding exact matched records we get 3 clusters \{"Thor", "Glenn", 12011990\}, \{"Thor", "Glenn", 12011990\}, \{"Cade", "Bale", 05011976\}, \{"Cade", "Balt", 05011976\}, and \{"Cade", "Bolt", 05021986\}.

3.4.2 Analysis

We analyze the time complexity by aggregating time complexities of all the steps. Step 1 calls radix sort for at most $D^2/2$ data sets, where $D$ is the number of data sets. If $D = 10$, which is very high for real world applications, the sorting algorithm is called at most 50 times. As radix sort is a linear time algorithm, this step consumes a linear amount of time on the number of records contained in those pairs of data sets. Step 1 reduces the number of records significantly in practical applications. Let the initial number of records be $N$ and this reduced number be $N'$. $K$-mer blocking is typically done on alphabet, number or alphanumeric values which generates $26^k$, $10^k$ or $36^k$ blocks, respectively. If a record length is $l$, then it should be in $(l - k + 1)$ blocks. To calculate blocking information of all the records, step 2 takes at most $(l' - k + 1)N'$ time, where $l'$ is the maximum length of any blocking attribute. Step 3 is the most time consuming step as it measures distances between records in every block. Let $b$ be the number of blocks, $b_n$ the average number of
records in these blocks and \( L \) be the maximum aggregated length of common attributes of records. Then this step takes \( O(bb'_n^2 L\tau) \) time, which can be written as \( O(b_n N' L\tau) \) as \( bb'_n = O(N') \). Step 4 scans through the generated graph and finds connected components. This step takes linear time in the number of records and connections, which is \( O(N') \). Steps 6 and 7 work on individual clusters that contain small numbers of records. If the number of these clusters is \( C \) and each cluster may contain \( O(D) \) records, then these steps take \( O(D^2 C) \) time that may be thought of as \( O(DN') \), where \( DC = O(N') \). We see that step 3 dominates the running time. Overall the running time is \( O(b_n N' L\tau) \), where \( b_n \) is the average number of records in a block (in step 3), \( N' \) is the number of clusters by exact matching, \( L \) is the maximum aggregated length of the common attributes of records and \( \tau \) is the user-defined threshold value.

### 3.4.3 Parallel Algorithm

We observe that the above RLA-CL algorithm has several phases, and almost all of these phases have independent working processes. For example, the distance calculation is done within each block. Therefore processors can perform linkage calculations independent of the others. Some steps are difficult to be parallelized optimally. For them we provide experimentally optimized solutions. Some steps are trivial to parallelize. Here we propose the PRLA-CL (Parallel Record Linkage Algorithm – Complete linkage) algorithm. One processor handles the input, output and collaboration with the other processors and is called the master processor and all the other processors are referred to as slave processors.

As displayed in Algorithm 7, after receiving data from the master, every processor selects pairs of data sets such that attributes of one data set cover all the attributes of the other data set. Then we accumulate records from each pair. Every processor sorts a specific range of records lexicographically. This range is chosen according to a prefix value of concatenated attributes of each record. If we choose the first 2 characters from each record, there are 676 combinations. If we have \( p \) processors, then every processor can keep track of records starting with \( \frac{676}{p} \) character combinations. The master collects and merges all the exact match records. Then the master chooses a representative from every exact matched group. Then it sends nearly an equal number of records to each of the slave...
Algorithm 7 PRLA-CL Parallel Record Linkage Algorithm using Complete Linkage Clustering

**Input:** A set of data sets and a configuration file

**Output:** A set of complete linkage clusters

1: procedure ComputeParallelCompleteLinkageClusters
2:   The Master reads data from the input files;
3:   The Master broadcasts data;
4:   for each processor do
5:     Determine which pairs of data sets should be sorted;
6:     Remove duplicates and merge records;
7:   end for
8:   The Master collects and merges all exact matched clusters;
9:   The Master distributes nearly uniformly representative records to each processor;
10:  for each processor do
11:     Create blocks of records sharing the same $k$-mers;
12:  end for
13:  The Master collects and merges all blocking information;
14:  The Master distributes block lists to all the processors nearly uniformly;
15:  for each processor do
16:    for each block in block list do
17:      Construct a graph where the records are vertices and the connections
18:        among them are edges;
19:      Connect two vertices if the distance (edit distance is one of the
20:        distance calculation methods) between them is at most the
21:        user defined threshold value;
22:    end for
23:  end for
24:  The Master accumulates edge lists from each processor;
25:  The Master finds connected components using these lists just as we do
26:  in Algorithm 3;
27:  The Master distributes clusters data uniformly to all the processors;
28:  for each processor do
29:    Perform complete linkage clustering and post processing
30:      (same as in Algorithm 3);
31:  end for
32:  The Master collects these clusters;
33:  The Master merges these clusters with records from initial exact matching clusters;
34: end procedure
processors. The slave processors generate blocks of records sharing some common k-mers. The master collects this blocking information. It then sorts blocks according to the number of records they contain. Then the master groups some blocks and aggregates squares of the numbers of all records in that group. The master does this grouping in such a way that all the groups have almost the same aggregate value. Then each processor finds the edge lists. The master collects them and finds connected components. Then the master splits these connected components equally among all the other processors. All of them compute the complete linkage clusters within each component. The master gathers all the clusters and expands every representative record by all of its exact matched records.

3.4.4 Analysis

This parallel algorithm distributes most of the work uniformly across all the processors. Major portions of them have been performed independently. Therefore communication cost is negligible with respect to the computational cost. Some steps have to be explained elaborately.

Step 1 takes \(O(N)\) time to read \(N\) records from \(D\) data sets and broadcast them. We see from the sequential algorithm that some pairs of data sets should be sorted to find duplicate records. In PRLA-CL, every processor determines those pairs of data sets. To compare among records, we concatenate common attributes of those records. We take the first 3 characters from each concatenated string. There may be \(s = 26^3, 10^3\) or \(36^3\) divisions of records if the characters are from English alphabet, number or alphanumeric values, respectively. Every processor sorts \(s/p\) divisions and removes duplicates by generating exact matching clusters. Although each processor does not get the same amount of records, the overall task is almost the same and the consumed time is really negligible compared to the other computations. Experimental results verify this statement. Therefore, if \(s_n\) is the maximum number of records of one division, then step 2 takes \(O(ss_n/p)\) time which is \(O(N/p)\), where \(ss_n = O(N)\). In step 3 every processor performs blocking on \(N'/p\) records which uses \(O(N'(L - l + 1)/p)\) time. In [89] we see some efficient techniques to distribute blocks among the processors. Step 4 consumes \(O(b_p b_m^2 L\tau)\) time, where \(b_p\) is the average number of blocks in a processor and \(b_m\) is the average number of records in
a block. Step 5 is straightforward as the master handles the collected data and finds the connected components in linear time in \( N' \) and number of connections, which is \( O(N') \). In step 6 the master distributes clusters in the same way it did for blocking in step 4. Every processor gets almost the same amount of workload to find complete linkages among the records. We assume that \( C \) is the number of intermediate clusters. Therefore, each processor does work in \( O(D^2C/p) \) or \( O(DN'/p) \) time. We see that the parallel algorithm has been perfectly parallelized. Experimental results show almost linear speed-up.

### 3.5 Results

We have implemented RLA-CL in C++ and PRLA-CL in C++ with MPI library. We deployed them on a HPC cluster having processors of 12 Intel Xeon X5650 Westmere cores and 48 GB RAM.

FEBRL\textsuperscript{[20, 17]} is a popular record linkage system. It generates clusters of very high accuracy. TPA(FCED)\textsuperscript{[102]} achieved a similar accuracy with much less time. From Table 6 of \textsuperscript{[102]} we see that TPA(FCED) took 203 ms in an experiment, whereas FEBRL needed 1284 ms. We outperformed TPA(FCED) by devising a novel RLA-SL algorithm\textsuperscript{[89]}. The implementation attained the same accuracy while being several times faster. The RLA-SL chapter integrated and analyzed some experimental results on real and simulated data sets. Those results exposed its efficiency and accuracy in real as well as simulated data sets. Those real data sets contained a very low percentage of errors. RLA-SL algorithm works really fine on real data sets. But yet we see it achieved not more than 98% accuracy for real data sets. Accuracy on simulated data sets varies widely due to a broad ranges of errors. In our experiments we count the possible traps of TPA(FCED) and RLA-SL algorithms and show how RLA-CL finds the expected output. We will also show how blocking information affects its performance. We will evaluate efficiency of record linkage algorithm using complete linkage hierarchical clustering over single linkage clustering. We have employed only simulated data sets, which contain much more errors than normal, to verify our statements of efficiency and accuracy of RLA-CL.
3.5.1 Generation of Simulated Data Sets

We generated three types of synthetic data sets. The first type has a data set of 1 million records. We made 10 copies of this data set. Then we introduced one insertion, deletion, or substitution error in the last name attribute of every record with a 15% probability. This means that around 15% of all the records in a data set have one mismatch from its original record. These data sets have the first name and SSN attributes along with some other attributes. We have taken an equal number of records from each data set in our experiments. If the number of records is 1 million, every data set contributes 100,000 records. SSN is a unique attribute for every record. We compute the accuracy using this attribute. This type is used to compare performances among TPA(FCED), RLA, and RLA-CL implementations. The second type of data sets were generated from the previous 1 million records. We copied this data set two more times. Then we inserted, deleted, or substituted one symbol in the last name of each record. This means that every record has at least one mismatch from its original record. We used four original data sets, and these two data sets three times. The third type is used for analysing different aspects of RLA-CL. The original data set has 1,600,000 records. We generate three copies of this data set. We remove different attributes from each data set. Then we introduce one insertion, deletion, or substitution error in the last name of every record. We analyze how RLA-CL works for different numbers and types of attributes. We have then cloned all of these three data sets.

3.5.2 Sequential Algorithm

We have categorized our experimental results into three sections. The first section shows that RLA-CL outperforms RLA-SL and TPA(FCED) in terms of accuracy and removes the chaining phenomenon. The number of blocks and types of blocking fields affect the running time and accuracy of RLA-CL. We explain them in the second section. In the third section we distribute the running time of RLA-CL and show that it does not take much time than RLA, the best-known algorithm in this category. We have divided the output data into four categories to measure accuracy. Type I includes perfect clusters.
Each cluster contains all the records of an individual and does not contain any record from other individuals. Every cluster of Type II has records of only one individual, but does not include all of them. All the records of an individual mixed with some records of the other individuals are included in Type III category. A Type IV cluster has some records from one individual mixed with some records of the other individuals. Here we see that Type I clusters are the most preferred. A Type IV cluster is a truly incorrect cluster. Therefore we prefer more records in Type I category and less records in Type IV category.

Table 3.1 compares our newly devised RLA-CL algorithm with the previously best-known RLA-SL algorithm as well as TPA(FCED). Number of records ranges from 100 thousands to 1 million across the five data sets. We have used the first name as the blocking field and Social Security Number as the accuracy testing attribute. We have used the edit distance calculation method on the first name and last name attributes. We have set 2 as the threshold calculation method on the first name and last name attributes. We have set 2 as the threshold value and 3 as the value of $k$.

Table 3.1: comparison among TPA(FCED), RLA-SL and RLA-CL on simulated data sets (generated with a low error rate)

<table>
<thead>
<tr>
<th>No Of Records</th>
<th>Algorithm</th>
<th>Time</th>
<th>Record Category(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Type I</td>
</tr>
<tr>
<td>100,000</td>
<td>TPA(FCED)</td>
<td>31.01</td>
<td>97.38</td>
</tr>
<tr>
<td></td>
<td>RLA</td>
<td>2.15</td>
<td>97.38</td>
</tr>
<tr>
<td></td>
<td>RLA-CL</td>
<td>3.08</td>
<td>99.88</td>
</tr>
<tr>
<td>200,000</td>
<td>TPA(FCED)</td>
<td>122.77</td>
<td>93.76</td>
</tr>
<tr>
<td></td>
<td>RLA</td>
<td>7.77</td>
<td>93.76</td>
</tr>
<tr>
<td></td>
<td>RLA-CL</td>
<td>10.4</td>
<td>99.05</td>
</tr>
<tr>
<td>400,000</td>
<td>TPA(FCED)</td>
<td>432.5</td>
<td>95.88</td>
</tr>
<tr>
<td></td>
<td>RLA</td>
<td>26.56</td>
<td>95.88</td>
</tr>
<tr>
<td></td>
<td>RLA-CL</td>
<td>32.15</td>
<td>99.45</td>
</tr>
<tr>
<td>600,000</td>
<td>TPA(FCED)</td>
<td>878</td>
<td>96.92</td>
</tr>
<tr>
<td></td>
<td>RLA</td>
<td>54.50</td>
<td>96.92</td>
</tr>
<tr>
<td></td>
<td>RLA-CL</td>
<td>62.54</td>
<td>99.61</td>
</tr>
<tr>
<td>800,000</td>
<td>TPA(FCED)</td>
<td>1503.53</td>
<td>97.57</td>
</tr>
<tr>
<td></td>
<td>RLA</td>
<td>87.66</td>
<td>97.57</td>
</tr>
<tr>
<td></td>
<td>RLA-CL</td>
<td>97.62</td>
<td>99.70</td>
</tr>
<tr>
<td>1,000,000</td>
<td>TPA(FCED)</td>
<td>2157.46</td>
<td>98.02</td>
</tr>
<tr>
<td></td>
<td>RLA</td>
<td>129.54</td>
<td>98.02</td>
</tr>
<tr>
<td></td>
<td>RLA-CL</td>
<td>141.17</td>
<td>99.76</td>
</tr>
</tbody>
</table>
From Table 3.1 we see that RLA-CL takes almost the same time as RLA. RLA-SL and TPA(FCED) produced the same number and types of clusters. But we see TPA(FCED) takes much more time than RLA-SL and RLA-CL. The RLA-SL chapter explained the inverse relationship between the multiplicity of exact matched records and the running time of RLA. The RLA-CL algorithm includes RLA-SL as a preprocessing step. After preprocessing is done, the generated clusters are of small size. Therefore complete linkage among the small number of records in every cluster consumes a small amount of time. Even for 1 million records RLA-CL spends only 13 seconds more than RLA. These few seconds do complete linkage clustering and post-processing of all the single linkage clusters. Figure 3.2 shows this time comparison. These results show that RLA-CL provides almost 100% Type I clusters whereas RLA-SL and TPA(FCED) produce around 96%-98% Type I clusters. If we consider 1,000,000 records of 100,000 individuals, RLA-CL only misses perfect clusters of 241 individuals whereas RLA-SL and TPA(FCED) do not find accurately all the records of 1981 people. This difference occurs because of the chaining problem of single linkage clustering. We have shown this Type I accuracy comparison in Figure 3.3.

![Figure 3.2: A comparison of running times of TPA(FCED), RLA-SL and RLA-CL on simulated data sets (generated with a low error rate)](image)

We have seen four types of accuracy in Table 3.1. Accuracy can also be calculated in terms of receiver operating characteristics (ROC). For the case of two classes, ROC-based accuracy is defined as (the number of true positives + the number of true negatives)/(the total number of records). We extend this definition of accuracy to more than two classes as follows. Each cluster is associated with a user who has a majority of records in this cluster. We say that this user owns this cluster. A record in any cluster is labeled as correct.
if it belongs to the owner of this cluster. Now we compute the accuracy as (the number of records with correct labels)/(the total number of records). Note that this definition of accuracy is a natural extension of ROC-based accuracy to more than two classes.

Table 3.2: Computation of accuracy of TPA(FCED), RLA-SL and RLA-CL on simulated data sets (generated with a low error rate)

<table>
<thead>
<tr>
<th>No Of Records</th>
<th>Algorithm</th>
<th>Records With Correct Labels</th>
<th>Accuracy in %</th>
</tr>
</thead>
<tbody>
<tr>
<td>100,000</td>
<td>TPA(FCED)</td>
<td>97880</td>
<td>97.88</td>
</tr>
<tr>
<td></td>
<td>RLA-SL</td>
<td>97880</td>
<td>97.88</td>
</tr>
<tr>
<td></td>
<td>RLA-CL</td>
<td>99949</td>
<td>99.95</td>
</tr>
<tr>
<td></td>
<td>TPA(FCED)</td>
<td>194910</td>
<td>97.46</td>
</tr>
<tr>
<td></td>
<td>RLA</td>
<td>194910</td>
<td>97.46</td>
</tr>
<tr>
<td></td>
<td>RLA-CL</td>
<td>199836</td>
<td>99.92</td>
</tr>
<tr>
<td>400,000</td>
<td>TPA(FCED)</td>
<td>392930</td>
<td>98.23</td>
</tr>
<tr>
<td></td>
<td>RLA</td>
<td>392930</td>
<td>98.23</td>
</tr>
<tr>
<td></td>
<td>RLA-CL</td>
<td>399835</td>
<td>99.96</td>
</tr>
<tr>
<td>600,000</td>
<td>TPA(FCED)</td>
<td>592100</td>
<td>98.68</td>
</tr>
<tr>
<td></td>
<td>RLA</td>
<td>592100</td>
<td>98.68</td>
</tr>
<tr>
<td></td>
<td>RLA-CL</td>
<td>599835</td>
<td>99.97</td>
</tr>
<tr>
<td>800,000</td>
<td>TPA(FCED)</td>
<td>791680</td>
<td>98.96</td>
</tr>
<tr>
<td></td>
<td>RLA</td>
<td>791680</td>
<td>98.96</td>
</tr>
<tr>
<td></td>
<td>RLA-CL</td>
<td>799836</td>
<td>99.98</td>
</tr>
<tr>
<td>1,000,000</td>
<td>TPA(FCED)</td>
<td>991500</td>
<td>99.15</td>
</tr>
<tr>
<td></td>
<td>RLA</td>
<td>991500</td>
<td>99.15</td>
</tr>
<tr>
<td></td>
<td>RLA-CL</td>
<td>999836</td>
<td>99.98</td>
</tr>
</tbody>
</table>

RLA-CL achieves more than 99.9% accuracy and TPA(FCED) and RLA-SL achieve around 97% - 99% accuracy for these data sets (shown in Table 3.2). Figure 3.4 also shows
these results graphically.

Figure 3.4: A comparison of accuracies of TPA(FCED), RLA-SL and RLA-CL on simulated data sets (generated with a low error rate)

Table 3.3: A comparison among TPA(FCED), RLA-SL and RLA-CL on simulated data sets (generated with a very high error rate)

<table>
<thead>
<tr>
<th>No Of Records</th>
<th>Algorithm</th>
<th>Time</th>
<th>Record Category(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Type I</td>
</tr>
<tr>
<td>100,000</td>
<td>TPA(FCED)</td>
<td>31.11</td>
<td>96.36</td>
</tr>
<tr>
<td></td>
<td>RLA</td>
<td>3.07</td>
<td>96.36</td>
</tr>
<tr>
<td></td>
<td>RLA-CL</td>
<td>4.21</td>
<td>99.86</td>
</tr>
<tr>
<td>200,000</td>
<td>TPA(FCED)</td>
<td>119.62</td>
<td>92.49</td>
</tr>
<tr>
<td></td>
<td>RLA</td>
<td>11.16</td>
<td>92.49</td>
</tr>
<tr>
<td></td>
<td>RLA-CL</td>
<td>14.51</td>
<td>99.54</td>
</tr>
<tr>
<td>400,000</td>
<td>TPA(FCED)</td>
<td>421.24</td>
<td>94.83</td>
</tr>
<tr>
<td></td>
<td>RLA</td>
<td>39.15</td>
<td>94.83</td>
</tr>
<tr>
<td></td>
<td>RLA-CL</td>
<td>46.67</td>
<td>99.69</td>
</tr>
<tr>
<td>600,000</td>
<td>TPA(FCED)</td>
<td>898.64</td>
<td>96.07</td>
</tr>
<tr>
<td></td>
<td>RLA</td>
<td>77.95</td>
<td>96.07</td>
</tr>
<tr>
<td></td>
<td>RLA-CL</td>
<td>88.66</td>
<td>99.77</td>
</tr>
<tr>
<td>800,000</td>
<td>TPA(FCED)</td>
<td>1507.45</td>
<td>96.91</td>
</tr>
<tr>
<td></td>
<td>RLA</td>
<td>129.79</td>
<td>96.91</td>
</tr>
<tr>
<td></td>
<td>RLA-CL</td>
<td>143.19</td>
<td>99.82</td>
</tr>
<tr>
<td>1,000,000</td>
<td>TPA(FCED)</td>
<td>2171.43</td>
<td>97.47</td>
</tr>
<tr>
<td></td>
<td>RLA</td>
<td>193.85</td>
<td>97.47</td>
</tr>
<tr>
<td></td>
<td>RLA-CL</td>
<td>209.45</td>
<td>99.85</td>
</tr>
</tbody>
</table>

We have also included results for the second type of data sets in Table 3.3. These data sets contain a very high error rate. Even for these data sets, RLA-CL shows almost 100% accuracy in finding perfect clusters.

Sometimes one attribute may be error prone than the others. Blocking on that field
produces blocks that may not hold all the records of same individuals. Multiple blocking attributes assure better results. We have explored this issue by employing three different experiments. One uses social security number (SSN) and the last name (LN) as blocking attributes, the second one uses only SSN and the last one uses only LN as the blocking attribute. We have used 5-mer on SSN, a numeric attribute, and 3-mer LN, which contains only English alphabet. Table 3.4 shows these comparisons in terms of running time and accuracy. We have used 6 data sets where 3 data sets have exact clone so that we could remove half of the records only after the exact matching phase.

Table 3.4: A comparison of runtime and accuracy using SSN-LN, SSN and LN as blocking fields

<table>
<thead>
<tr>
<th>No Of Records</th>
<th>Algorithm</th>
<th>Time</th>
<th>Record Category</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Type I</td>
</tr>
<tr>
<td>100,000</td>
<td>SSN-LN</td>
<td>111.51</td>
<td>100,000</td>
</tr>
<tr>
<td></td>
<td>SSN</td>
<td>107.71</td>
<td>89,866</td>
</tr>
<tr>
<td></td>
<td>LN</td>
<td>5.56</td>
<td>80,476</td>
</tr>
<tr>
<td>200,000</td>
<td>SSN-LN</td>
<td>252.46</td>
<td>199,988</td>
</tr>
<tr>
<td></td>
<td>SSN</td>
<td>237.83</td>
<td>179,972</td>
</tr>
<tr>
<td></td>
<td>LN</td>
<td>16.20</td>
<td>161,522</td>
</tr>
<tr>
<td>400,000</td>
<td>SSN-LN</td>
<td>537.83</td>
<td>399,952</td>
</tr>
<tr>
<td></td>
<td>SSN</td>
<td>480.49</td>
<td>359,656</td>
</tr>
<tr>
<td></td>
<td>LN</td>
<td>48.47</td>
<td>322,612</td>
</tr>
<tr>
<td>800,000</td>
<td>SSN-LN</td>
<td>1064.43</td>
<td>799,904</td>
</tr>
<tr>
<td></td>
<td>SSN</td>
<td>822.92</td>
<td>719,474</td>
</tr>
<tr>
<td></td>
<td>LN</td>
<td>169.08</td>
<td>644,204</td>
</tr>
<tr>
<td>1,600,000</td>
<td>SSN-LN</td>
<td>2657.35</td>
<td>1,599,832</td>
</tr>
<tr>
<td></td>
<td>SSN</td>
<td>1912.80</td>
<td>1,439,608</td>
</tr>
<tr>
<td></td>
<td>LN</td>
<td>622.12</td>
<td>1,290,514</td>
</tr>
<tr>
<td>3,200,000</td>
<td>SSN-LN</td>
<td>6422.64</td>
<td>3,199,676</td>
</tr>
<tr>
<td></td>
<td>SSN</td>
<td>4261.62</td>
<td>2,877,986</td>
</tr>
<tr>
<td></td>
<td>LN</td>
<td>2379.88</td>
<td>2,583,536</td>
</tr>
</tbody>
</table>

Experiments that do blocking on LN take very little time for a small number of records. But the running time increases rapidly for higher number of records. LN has an average length of 5. Therefore every record on an average goes to 3 blocks. For 100,000 records, 300,000 records are stored in 263 or 17576 blocks. Every block holds around 17 records on an average. But when we have 3,200,000 records, we have 17576 blocks to keep 9,600,000 records. Each block has to store on an average 546 records. We know that the distance
calculation occurs among records densely within every block. This is the most time consuming phase of our algorithm. On the other side SSN uses 5-mers for blocking. Therefore every record goes to 5 blocks. For 100,000 records 105 or 100,000 blocks hold 500,000 records, which is 5 per block on average. For 3,200,000 records this number is 160. But the most compelling reason is that some combinations of letters are more frequent than the others. This makes some blocks much larger than the others. But for numerical values every block is almost equally populated. The time needed for two attributes is the summation of these two attributes. We have depicted this scenario in Figure 3.5. Blocking attribute has a greater impact on accuracy. Our generated records contain errors either in the SSN or the LN. Therefore many blocks may not be able to hold records of the same individuals. But if we take blocking of two attributes, we get around 100% Type I clusters. Blocking on SSN achieves 90% and blocking on LN gets 81% Type I clusters. SSN has a better performance as each record goes to 6 blocks compared to 3 blocks for the LN attribute. Figure 3.6 displays the impact of blocking attributes over Type I accuracy.

Figure 3.5: A comparison of running time for variations of blocking attributes

Figure 3.6: A comparison of Type I accuracy for variations of blocking attributes
Table 3.5 distributes the running time of RLA-CL when SSN has been used as the blocking field. Exact matching is required to remove exact duplicates. Linear time radix sorting algorithm does this step efficiently. Approximate clustering is the most time consuming phase that includes linkage calculation steps. We find clusters as connected components with almost no time. These three portions are required for both RLA-SL and RLA-CL. The later one requires some extra time to find complete linkages. We see from the table that this value is negligible compared to approximate cluster time. We have shown these distributions in Figure 3.7.

<table>
<thead>
<tr>
<th>No of Records</th>
<th>Exact Cl T</th>
<th>Approx Cl T</th>
<th>Conn Comp T</th>
<th>Comp Link T</th>
<th>Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>100,000</td>
<td>0.77</td>
<td>105.34</td>
<td>0.01</td>
<td>1.59</td>
<td>107.71</td>
</tr>
<tr>
<td>200,000</td>
<td>1.90</td>
<td>232.25</td>
<td>0.03</td>
<td>3.65</td>
<td>237.83</td>
</tr>
<tr>
<td>400,000</td>
<td>3.62</td>
<td>468.84</td>
<td>0.06</td>
<td>7.97</td>
<td>480.49</td>
</tr>
<tr>
<td>800,000</td>
<td>6.36</td>
<td>803.65</td>
<td>0.08</td>
<td>12.83</td>
<td>822.92</td>
</tr>
<tr>
<td>1,600,000</td>
<td>13.92</td>
<td>1870.60</td>
<td>0.17</td>
<td>28.11</td>
<td>1912.80</td>
</tr>
<tr>
<td>3,200,000</td>
<td>29.59</td>
<td>4172.79</td>
<td>0.35</td>
<td>58.89</td>
<td>4261.62</td>
</tr>
</tbody>
</table>

Figure 3.7: Running time distribution of RLA-CL

3.5.3 Parallel Algorithm

We have run our parallel algorithm on 3.2 million and 6.4 million records blocking on the last name (LN) attribute. Our parallel experiments have been tested on at most 32 cores of 4 nodes, each node having 8 cores.

Figure 3.8 shows the running time on different number of processors. We get almost
linear speedup. These speedups have been drawn in Figure 3.9. We see that the most
time-consuming part is the single linkage calculation among the records within individual
blocks. Different blocks have different numbers of records. Even if two blocks have the
same number of records, they may need different time as the time needed depends on
matching of records as well as record lengths. We have distributed the runtime of PRLA-
CL in Table 3.6. Detailed time distribution of different tasks of parallel RLA-SL such as
broadcast time, communication time, time spent by the master, blocking time, merge time,
edgelist calculation time, etc. have been described in [89]. In Table 3.6 we have included
communication time, which aggregates broadcast, communication and merge time, exact
matching time, approximation clustering time that covers the generation of blocks and
calculation of linkage time, finding connected components time and complete linkage time,
which includes complete linkage and post-processing time. The first row shows the running
time of the same data for the sequential algorithm.

Table 3.6: Distribution of running time on multiple cores

<table>
<thead>
<tr>
<th>Proc</th>
<th>Comm</th>
<th>Exact</th>
<th>Approx</th>
<th>Comp</th>
<th>Total</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>55.87</td>
<td>9219.32</td>
<td>0.69</td>
<td>115.98</td>
<td>9391.86</td>
</tr>
<tr>
<td>2</td>
<td>0.87</td>
<td>28.35</td>
<td>4655.38</td>
<td>0.63</td>
<td>59.22</td>
<td>4744.45</td>
</tr>
<tr>
<td>4</td>
<td>0.91</td>
<td>15.03</td>
<td>2403.09</td>
<td>0.65</td>
<td>31.23</td>
<td>2450.91</td>
</tr>
<tr>
<td>8</td>
<td>1.08</td>
<td>7.40</td>
<td>1226.59</td>
<td>0.70</td>
<td>17.01</td>
<td>1252.78</td>
</tr>
<tr>
<td>16</td>
<td>5.27</td>
<td>4.18</td>
<td>647.63</td>
<td>0.64</td>
<td>9.33</td>
<td>667.05</td>
</tr>
<tr>
<td>32</td>
<td>7.87</td>
<td>2.39</td>
<td>343.41</td>
<td>0.63</td>
<td>5.57</td>
<td>359.87</td>
</tr>
</tbody>
</table>

Figure 3.8: Running time of PRLA-CL for 3.2M and 6.4M records

We have achieved 7.5 speedup for 8 cores in a single node, 14.1 for 16 cores across 2
Figure 3.9: Speedup of PRLA-CL for 3.2M and 6.4M records

Table 3.6 also shows that communication time is very negligible as most of the steps of the parallel algorithm are easily parallelized. Therefore communication is needed after each phase only. These speedups are great as they are almost linear, but we can improve these speedups if we can ensure a better uniform distribution of blocks among the processors in terms of needed calculation time.

3.6 Discussion

From the results section we see that the single linkage clustering algorithm suffers from the chaining problem. RLA-CL overcomes this problem by employing complete linkage clustering. Accuracy performance of our new algorithm sometimes depends on the number and type of blocking attributes. There is a trade-off between time spent and accuracy. Stable fields should be chosen as blocking attributes. The value of k also affects the running time and accuracy. If we use 4-mers instead of 3-mers, there will be more blocks. Each block will contain less records on an average and therefore it will cost less than before. But a 4-mer creates less substrings of records which will decrease the accuracy. RLA-CL works on different numbers and types of attributes. It does post-processing on complete-linkage clusters based on priority-list attributes. Another major factor that has a great impact on efficiency and accuracy is the threshold value. RLA-SL applies a constant as well as a proportional threshold value and provides those results. Our new algorithm works in the same way. We can apply different threshold values on the training data sets to find out the perfect threshold value for these data sets. There is no universal threshold value for all
types of data sets. Errors introduced in the data sets also have an effect on the threshold value and the performance of the algorithms. If the error rate is low, a threshold value of 1 works fine most of the time.
Chapter 4

RLT-S: A web system for record linkage

4.1 Introduction

Record linkage has evolved as a crucial problem in many areas of science and engineering. A large number of health agencies store medical records of patients \[22, 135\]. Finding data of an individual across these sources requires efficient algorithms. Record linkage has also applications in disease evolution \[35, 86\], master data management, copy detection in digital documents \[10, 125\], historical data management, and so on.

Record linkage collects records of same individuals from multiple data sources possibly having some corrupted records due to typo, phonetic similarity, etc. Now-a-days data of an individual reside across multiple databases and at the same time data agencies keep records of millions of people. Accuracy as well as time efficiency in finding all the records of an individual make the problem challenging. A naive algorithm compares each pair of records and measures similarities. This method is very time consuming. Many algorithms have been devised to improve this naive algorithm \[38, 41, 146, 118, 101\]. We have already proposed efficient and effective sequential and parallel record linkage algorithms \[89\], which outperform previous best-known record linkage algorithms \[102\]. Our methods use single linkage hierarchical clustering which generates a dendrogram. By applying a threshold
value on this dendrogram we get our expected clusters for individuals.

A large number of record linkage tools are widely available. Java-based fine-grained probabilistic record integration and linkage tool (FRIL) is an open source tool, which has support for parameters configuration and can handle millions of records [62, 61]. Another widely used record linkage tool is FEBRL (Freely Extensible Biomedical Record Linkage) which performs data standardization as well as probabilistic record linkage of one or more files [17].

In this work we present details on our record linkage tool, RLT-S, which implements the record linkage algorithm based on single linkage clustering of [89]. This tool is freely available in www.rlatools.com website. The website also provides proper instructions, submission history and some other necessary features to ease the usage of the tool. The tool generates a well-formatted output to facilitate user perception. Here we describe the functionalities of the tool as well as necessary parameters for input handling, linkage processing, and generation of output.

### 4.2 Implementation

RLT-S is a Java implementation of sequential RLA (Record Linkage Algorithm) [89]. This algorithm clusters records of individuals using single linkage hierarchical clustering. It merges records from all the data sets as if they were from one data set. Therefore the performance of the algorithm is independent of the number of input data sets. It sorts records on common attribute fields using radix sort. Sorting helps us to separate duplicate records, which indicates exact matching. Real life applications do not contain much error. Therefore the sorting phase reduces the size of the unified data set into a smaller data set with no exact duplicates of records. Next phases work on only the representative records from all the exact match clusters. We call the first record of each cluster as the representative record of that cluster. Our RLA employs single linkage hierarchical clustering. Comparison between each pair of records consumes a lot of time. To facilitate finding groups of similar records it employs blocking on a specified attribute field. It finds linkages among those records in a block. We use l-mers (i.e., substrings of length l) of the
attribute field for blocking. Any L-length record will be present in \((L - l + 1)\) blocks. In this way different blocks are connected. We then employ hierarchical clustering with single linkage by measuring distances between pairs of records in a block using any combination of edit distance, reversal distance, and truncation distance methods. If we consider each record as a vertex in a graph and linkage as an edge between two vertices, then we get our desired graph. We remove multiple edges and self-loops from this graph. Each connected component of this graph is a cluster of records of an individual. The algorithm outputs these clusters and all their identical records generated by exact matching as final clusters.

The website and the associated tool perform record linkage among one or more data files. We have simplified the usage of the tool by minimizing queries and text input. Whenever possible we have provided drop-down lists to select possible values. Generated output is also well formatted so that the users can easily identify records of an individual.

### 4.3 Results and Discussion

http://www.rlatools.com hosts RLT-S tool and provides all the other necessary features to ensure the best functioning of the tool. Anyone can use this freely available tool with or without login information. Registered users have the facility to view all of their submissions information and outputs. But without login also users can use the tool and find their outputs using links via email. Currently we keep all the output files in our server so that the users can view and download them at any time. Figure 4.1 shows a diagram of the website pipeline.

Our tool requires some specific information. Some of these are required, some are highly recommended and a few of them are optional. We have set default values for some attributes if the users do not want to input them. RLT-S works in three separate phases. The first and the third phases work with the input and the output. The second phase tunes parameters and controls the working process of the tool.
Figure 4.1: Web-based user interface. (A) shows the first and main page of the website, where users select data files, choose configurations and submit them. (B) is the instruction page. Users can view their submission history through login (C). (D) shows a sample submission history page.

4.3.1 Input Data Sets and Configurations

The tool starts with reading and organizing the input data sets. The number of input data sets is required to browse those data files from the user’s computers. It accepts .txt or .csv extension files where the attributes of each record are comma separated and each record is in a separate line. We illustrate the working of RLT-S with an example. This example pertains to Tables 4.1, 4.2, and 4.3.

Table 4.1: Records for 5 people having 9 attributes

<table>
<thead>
<tr>
<th>ID</th>
<th>FN</th>
<th>LN</th>
<th>SSN</th>
<th>DoB</th>
<th>G</th>
<th>SchID</th>
<th>MN</th>
<th>SSID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Risa</td>
<td>Pierce</td>
<td>133183594</td>
<td>9261990</td>
<td>M</td>
<td>1524</td>
<td>Vesta</td>
<td>676221410</td>
</tr>
<tr>
<td>2</td>
<td>Maile</td>
<td>Kramer</td>
<td>135370878</td>
<td>7261991</td>
<td>F</td>
<td>1526</td>
<td>Lenna</td>
<td>957261480</td>
</tr>
<tr>
<td>3</td>
<td>Kimberly</td>
<td>Battle</td>
<td>141274186</td>
<td>4071982</td>
<td>F</td>
<td>1527</td>
<td>Jacki</td>
<td>144591609</td>
</tr>
<tr>
<td>4</td>
<td>Kamal</td>
<td>Mcclain</td>
<td>148965694</td>
<td>10091991</td>
<td>M</td>
<td>70000</td>
<td>Luisa</td>
<td>278635088</td>
</tr>
<tr>
<td>5</td>
<td>Yvonne</td>
<td>Vaughan</td>
<td>153614228</td>
<td>2061992</td>
<td>F</td>
<td>70003</td>
<td>Basil</td>
<td>368901550</td>
</tr>
</tbody>
</table>

Each row of the table represents each row of Input01.csv file.
Table 4.2: Records for 5 people having 4 attributes

<table>
<thead>
<tr>
<th>ID</th>
<th>First Initial</th>
<th>Last Name</th>
<th>Social Security Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>R</td>
<td>Pierce</td>
<td>133183594</td>
</tr>
<tr>
<td>2</td>
<td>M</td>
<td>Kramer</td>
<td>135370878</td>
</tr>
<tr>
<td>3</td>
<td>K</td>
<td>Battle</td>
<td>141274186</td>
</tr>
<tr>
<td>4</td>
<td>K</td>
<td>Mcclain</td>
<td>148965694</td>
</tr>
<tr>
<td>8</td>
<td>L</td>
<td>MUELLER</td>
<td>184498846</td>
</tr>
</tbody>
</table>

Each row of the table represents each row of Input02.csv file

Table 4.3: Records for 5 people having 8 attributes

<table>
<thead>
<tr>
<th>ID</th>
<th>First Name</th>
<th>Last Name</th>
<th>Date of Birth</th>
<th>Gender</th>
<th>SchID</th>
<th>MN</th>
<th>SSID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>RISA</td>
<td>PIERCE</td>
<td>9261990</td>
<td>M</td>
<td>1524</td>
<td>VESTA</td>
<td>676221410</td>
</tr>
<tr>
<td>2</td>
<td>MAILE</td>
<td>KRAMER</td>
<td>7261991</td>
<td>F</td>
<td>1526</td>
<td>LENNA</td>
<td>957261480</td>
</tr>
<tr>
<td>3</td>
<td>KIMBERLY</td>
<td>BATTLE</td>
<td>4071982</td>
<td>F</td>
<td>1527</td>
<td>JACKI</td>
<td>144591609</td>
</tr>
<tr>
<td>5</td>
<td>YVONNE</td>
<td>VAUGHAN</td>
<td>2061992</td>
<td>F</td>
<td>70003</td>
<td>BASIL</td>
<td>368901550</td>
</tr>
<tr>
<td>8</td>
<td>KELSIE</td>
<td>MUELLER</td>
<td>1131992</td>
<td>M</td>
<td>70020</td>
<td>JAKE</td>
<td>7243583370</td>
</tr>
</tbody>
</table>

Each row of the table represents each row of Input03.csv file

Table 4.1 shows records of five people. Each record has ID, FN, LN, SSN, DoB, Gender, SchID, MN, and SSID. Table 4.2 has also 5 records having ID, First Initial, Last Name and Social Security Number as attributes. We have another 5 records from Table 4.3, each of which has ID, FirstName, LastName, DateOfBirth, Gender, SchID, MN, and SSID as attributes. These 3 tables have different numbers of attributes. We see that the attribute names are quite different from each other although some of them represent the same type. FN, First Initial and FirstName represent first name of a person. Similarly LN, Last name and LastName are similar types.

Consider the task of integrating these 3 tables using our tool. The first required field in RLT-S is the number of input data sets. In this case we select 3 from drop-down list. Then we have to select input files from our computer. As we see some attribute type names are different although they represent the same attribute, we remove first row from each input file. There are 3 browse fields. We browse our computer and select one file at a time. Input01.csv is added at the first browse field, Input02.csv and Input03.csv are added at the second and third fields, respectively. Next required field is the number of attribute types. In this example we see that 9 unique attribute types are present. So we select 9 from the
drop-down list. We have seen that the same type has been represented differently in these files. So we choose 9 suitable names for these attribute types, for example, ID, FirstName, LastName, SSN, DateOfBirth, Gender, SchoolID, MiddleName and SSID. Now we have to connect these names with attribute field names of each data set. We note that ID is the 0th index of 1st input file, FirstName is the 1st, LastName, SSN, DateOfBirth, Gender, SchoolID, MiddleName and SSID are subsequent indices. So we select 0, 1, 2, 3, 4, 5, 6, 7, 8 from the drop-down list for dataset0 or Input01.csv. For dataset1 or Input02.csv, ID is the 0th, FirstName is the 1st, LastName is the 2nd and SSN is the 3rd index. This file has no DateOfBirth, Gender, SchoolID, MiddleName and SSID attribute fields. So we select -1 for each of them. Dataset1 should have 0, 1, 2, 3, -1, -1, -1, -1. Input03.csv has no SSN field. So we put -1 for this index. DateOfBirth is at index 3 of this file, Gender is at 4, and so on. Therefore we select 0, 1, 2, -1, 3, 4, 5, 6, 7 for this dataset. This is the last step of the input phase. Figure 4.2 shows the above selection of input files.

![Figure 4.2: Screenshot of input parameter selection for our 3 example files.](image)

4.3.2 Linkage Parameters

This stage handles the working process of the tool. RLT-S works on common attributes across all the data sets. We need a measure of distance between two attributes (i.e., two strings of characters). RLT-S supports three distance measures. The tool has three types of fields for each comparison. The first field is to select the comparison method (i.e., the distance measure), second one is for selection of the index on which the selected comparison method should be applied and the third type is truncation count, which is the number of characters of the selected attribute that should be used for comparison. We integrate three different comparison methods. The first comparison method is the edit distance calculation.
Edit distance or Levenshtein distance measures the minimum number of edit operations required to transform one string to another. Operations include insertion, deletion and substitution. For example, consider the strings $A = \text{"computer"}$ and $B = \text{"conuterr"}$. If we substitute 'n' to 'm', insert 'p' after this 'm' and delete the last 'r' from $B$, we get $A$. Edit distance method needs at least three operations to transform $B$ to $A$. To use this method, the user has to select an index of the common attribute on which this method will apply.

Another distance measure is the reversal distance. Consider a string of two attributes separated by comma, $A = \text{"James, Hudson"}$ and another string $B = \text{"Hudson, Hames"}$. This method first calculates the edit distance between $A$ and $B$. We note that a large number of operations are required to transform one string to another. Then it alters the positions of the two strings of $B$ generating $B' = \text{"Hames, Hudson"}$ and then measures the distance between $A$ and $B'$. It finds that only one operation is needed for the transformation. The reversal distance measure is defined as the minimum of the distance between $A$ and $B$ and the distance between $A$ and $B'$. This method is very useful for the first name and the last name attributes or any other related attributes as users may occasionally input the first name in the last name field and vice versa. Reversal edit distance method needs two attributes to work with. So there are two index fields, each of which should be a unique common attribute index. Truncation distance method is the last method used in our tool. This method is the same as the edit distance method except that it only compares truncation count number of initial characters of both attributes. For example, if $A = \text{"James"}$ and $B = \text{"J"}$ and if the truncation count is 1, this method calculates the edit distance between $A' = \text{"J"}$ and $B' = \text{"J"}$. Truncation often occurs for first names as some sources keep only the first name initials. Our single linkage-clustering algorithm reduces a major portion of time for linkage calculation by using blocks on a specific attribute. Each block stores information of similar records. Therefore output accuracy also depends on the choice of the attribute field and its average length. In [89], we have used the last name as the blocking field because the last name was the most appropriate attribute in our collected records. Any other important and reliable common attribute may be the blocking field as well. In most of the cases there is no way to measure a perfect average length of the block field. But an approximate average length makes our job easy. If the user does
not fill in this field, the tool uses 7 as the default value. In [90], we show how the value of \( k \) affects the blocking performance. In our example we have used 2 comparisons, edit distance calculation on the last name, and truncation distance method on the first name with a truncation count of 1. Our linkage criteria are shown in Figure 4.3.

![Linkage Criteria Screenshot](image)

Figure 4.3: Screenshot of linkage criteria for our 3 example files.

### 4.3.3 Output

The third stage requires some information to generate and send outputs. RLT-S employs hierarchical clustering and generates a dendrogram. A dendrogram represents clusters produced by hierarchical clustering in a tree in a well-organized way. A threshold value is needed to output the desired clusters. This threshold value says how many errors RLT-S endures. To understand fully the threshold value, let us consider another example. One record has James, Rodriguez, and 01011990 for FirstName, LastName and DateOfBirth fields, respectively, and another record has Rodriuez, 123456789, and Hames as LastName, SocialSecurityNumber and FirstName attributes. We see that 'James' has been transformed into 'Hames' which indicates that one substitution is needed to correct it and one character has been deleted from 'Rodriguez'. The matching process finds 2 errors between these two records. If we input a threshold value of 1, then the tool produces 2 clusters, each of which contains only one record. But if the threshold value is at least 2, only one cluster having these two records will be generated. A threshold value of 0 generates exact clustering in which every record matches on common attributes. If the users do not input any value, the tool uses the most used threshold value of 1 as the default value. User’s email address is required as the output link is sent to this address. For our example of 3 input files, if we choose 1 as the threshold value, then RLT-S will generate the output shown in Table 4.4.
4.3.4 Submissions History

Users may want to check their previous submissions and outputs. The website allows users to login to view their submissions history. They can check query id, submission time, processed time, download link, etc. of all the submitted jobs.

Any user can use the tool without login information. Valid email address is needed to get the download link of the output. First time users need to reset their password from the “submission info” section to be registered. They can also change their current password from this page.

4.3.5 Feedback

The website provides a feedback option for further improvements to RLT-S. Users can select a feedback type and post comments. We will study the feedbacks very carefully and modify the system accordingly.

RLT-S application has been implemented in Java. We have used Apache server with MySQL to host the website. A service works in the background to trigger the application when a new job is posted. This service also keeps track of the finished jobs, updates

<table>
<thead>
<tr>
<th>C ID</th>
<th>F ID</th>
<th>ID</th>
<th>F Name</th>
<th>L Name</th>
<th>SSN</th>
<th>DoB</th>
<th>G</th>
<th>SID</th>
<th>M Name</th>
<th>SSID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>r</td>
<td>pierce</td>
<td>133183594</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>risa</td>
<td>pierce</td>
<td>133183594</td>
<td>9261990</td>
<td>m</td>
<td>1524</td>
<td>vesta</td>
<td>676221410</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>1</td>
<td>risa</td>
<td>pierce</td>
<td>9261990</td>
<td>m</td>
<td>1524</td>
<td>vesta</td>
<td>676221410</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>m</td>
<td>kramer</td>
<td>135370878</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>maile</td>
<td>kramer</td>
<td>135370878</td>
<td>7261991</td>
<td>f</td>
<td>1526</td>
<td>lenna</td>
<td>957261480</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>2</td>
<td>maile</td>
<td>kramer</td>
<td>7261991</td>
<td>f</td>
<td>1526</td>
<td>lenna</td>
<td>957261480</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
<td>k</td>
<td>battle</td>
<td>141274186</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>3</td>
<td>kimberly</td>
<td>battle</td>
<td>141274186</td>
<td>4071982</td>
<td>f</td>
<td>1527</td>
<td>jacki</td>
<td>144591609</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>kimberly</td>
<td>battle</td>
<td>4071982</td>
<td>f</td>
<td>1527</td>
<td>jacki</td>
<td>144591609</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>4</td>
<td>k</td>
<td>mcclain</td>
<td>148965694</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>4</td>
<td>kamal</td>
<td>mcclain</td>
<td>148965694</td>
<td>10091991</td>
<td>m</td>
<td>7000</td>
<td>luisa</td>
<td>278635088</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>5</td>
<td>yvonne</td>
<td>vaughan</td>
<td>153614228</td>
<td>2061992</td>
<td>f</td>
<td>7000</td>
<td>basil</td>
<td>368901550</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>5</td>
<td>yvonne</td>
<td>vaughan</td>
<td>2061992</td>
<td>f</td>
<td>7000</td>
<td>basil</td>
<td>368901550</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>8</td>
<td>kelsie</td>
<td>mueller</td>
<td>1131992</td>
<td>m</td>
<td>70020</td>
<td>jake</td>
<td>7243583370</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>8</td>
<td>l</td>
<td>mueller</td>
<td>184498846</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
database and sends email to corresponding users. MySQL database stores users’ information and their submission history. Our tool takes negligible amount of time for thousands of records. If needed, we will integrate our parallel implementation of single linkage clustering algorithm into this website in future.

4.4 Comparisons

FEBRL and FRIL are well known and widely used freely available record linkage tools. These tools perform standardization or deduplication of a file or linkage between two files. On the other hand, our system RLT-S can handle any number of input datasets. Several experimental results reported in [89] show that our RLA algorithm outperforms previous best-known algorithms for error-induced datasets. Those experiments also describe the process of choosing suitable threshold values for different datasets. Our algorithm achieves around 98% accuracy on four real datasets having 1 million records in total. This algorithm was 70 times faster than the previous best-known algorithm, TPA (FCED) [102], for these datasets.

<table>
<thead>
<tr>
<th>Tool Name</th>
<th>(1000 &amp; 1000)</th>
<th>(2000 &amp; 2000)</th>
<th>(3000 &amp; 3000)</th>
<th>(4000 &amp; 4000)</th>
<th>(5000 &amp; 5000)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RLT-S</td>
<td>95</td>
<td>110</td>
<td>142</td>
<td>212</td>
<td>237</td>
</tr>
<tr>
<td>FEBRL</td>
<td>330</td>
<td>834</td>
<td>1630</td>
<td>2770</td>
<td>4150</td>
</tr>
<tr>
<td>FRIL</td>
<td>841</td>
<td>1992</td>
<td>3555</td>
<td>6043</td>
<td>8683</td>
</tr>
<tr>
<td>TPA (FCED)</td>
<td>172</td>
<td>223</td>
<td>274</td>
<td>360</td>
<td>433</td>
</tr>
</tbody>
</table>

Times shown are in milliseconds. Computation times are taken for (number of records in first file, number of records in second file)

Many of the available record linkage tools achieve very good accuracy, but they suffer from higher time complexities to generate linkages among datasets. Efficient TPA (FCED) (Two Phase Algorithm with Faster Computation of the Edit Distance) has been compared with FEBRL for two datasets of different sizes [102]. In this chapter we go through some experiments which use the same parameter configurations as [102]. All of these four tools, namely RLT-S, FEBRL, FRIL, and TPA (FCED), achieve 100% accuracy for these simu-
lated datasets. Table 4.5 shows computation times of these four tools for 1000, 2000, 3000, 4000, and 5000 records. In this table the notation (X, Y) stands for the number (X) of records in the first file and the number (Y) of records in second file.

We see that FEBRL is performing better than FRIL for each of these data sets. TPA (FCED) is faster than FEBRL that is also shown in [102]. RLT-S outperforms all of these tools. Our tool performs its best for real datasets, where the possibility of error occurrences and the number of errors in the input datasets are low.
Part III

Problems with Algorithmic Challenges
Chapter 5

An efficient minimum spanning tree algorithm

5.1 Introduction

The minimum spanning tree (MST) problem of a connected graph $G = (V, E)$ is to find a spanning tree with minimum total edge weight. This problem has been extensively studied and is a subproblem of many well known network problems. It has practical applications in wireless networks and VLSI layout design [99, 134, 147], many graph problems such as ear decomposition, connectivity checking [93, 103, 133], ovarian and bronchial cancer detection and many other medical diagnoses [11, 66, 67, 95], network evolution [14], and so on.

A large number of algorithms have been devised to find minimum spanning trees in graphs. Boruvka’s algorithm finds the cheapest edges from each component until only one component is left. The algorithm runs in $O(|E|\log|V|)$ time where $V$ is the set of vertices and $E$ is the set of edges. Jarnik-Prim algorithm [57, 116] starts with a single node and grows the tree by adding one edge at a time until all vertices are in the tree. Implementation of the algorithm uses efficient priority queues, and the runtime is mostly dependent on the specific implementation of a priority queue. It takes $O(|E|\log|V|)$ time if a binary heap is used to realize a priority queue. A Fibonacci heap version takes $O(|E|+|V|\log|V|)$ time. Kruskal’s algorithm [73] is a greedy algorithm which finds a minimum spanning forest. It
sorts the edges in increasing order of their weights. It starts with forest of $|V|$ forests where each tree is a single node (corresponding to a vertex in the graph). From thereon it tries to add one edge at a time into the forest. If the edge does not cause a cycle it is added in which case the number of trees decreases by one. When the forest has only one tree, the algorithm terminates. The running time of the algorithm is $O(|E|\log |V|)$. More efficient algorithms have been developed based on these algorithms.

Prim’s algorithm performs better than Kruskal’s algorithm for dense graphs whereas Kruskal’s algorithm is faster for sparse graphs. The algorithm we propose in this chapter, Edge Pruned MST (EPMST), exploits the ideas of both of these algorithms. In particular, EPMST uses Kruskal’s algorithm on a small subset of the edges. Then if needed, it uses Prim’s algorithm on a denser compact graph. EPMST employs random sampling to choose the subset of edges.

The rest of this chapter is organized as follows. In section 5.2 we provide a brief summary of existing MST algorithms. In section 5.3 we describe a recently devised efficient algorithm, Filter-Kruskal [110], briefly. In this section we also present and analyze our algorithm, EPMST. Section 5.4 presents an experimental comparison between EPMST and Filter-Kruskal over a wide variety of input graphs.

5.2 Related Studies

Algorithms developed by Boruvka, Kruskal, and Prim provide the base for a large number of efficient algorithms. Since Kruskal’s algorithm works on sorted edges and MST edges are likely to be among the $O(n \log n)$ smallest weight edges, partial sorting of edges work greatly to find out lighter edges [9] [112]. Filtering of edges connecting nodes in the same component makes the algorithm much faster [69]. Bucket sorting on edges has been also employed in Kruskal’s algorithm which performs greatly for uniformly distributed random edge weights [64]. A recently devised Filter-Kruskal algorithm has an improved performance over a wide range of input types [110]. The Filter-Kruskal algorithm has been compared with some well-known fast algorithms and has been shown to yield a better performance. As a result, in this work, we have chosen the Filter-Kruskal algorithm to compare with our algorithm.
5.3 Methods

In this section we devise and analyze an efficient minimum spanning tree algorithm. The algorithm performs much better for not very sparse graphs. We have chosen the Filter-Kruskal algorithm to compare with as this is the latest well-known algorithm which has a better performance over previous algorithms.

5.3.1 Filter-Kruskal Minimum Spanning Tree Algorithm

Filter-Kruskal is a modified and extended version of Kruskal’s algorithm which avoids sorting all of the input edges [110]. The algorithm runs Kruskal’s algorithm on subsets of candidate edges repeatedly. To find these subsets, it chooses a pivot edge which separates unused edges into two sets. The method is called on these sets recursively. Lighter sets are more probable edge sets. If all the required edges are not found in these sets, then it explores the heavier edge sets. Another important feature of this algorithm is that it chooses only those edges whose end points are in two different components.

The algorithm selects \( c|V| \) number of edges each time it invokes Kruskal’s algorithm, where \( c \) is a constant. It chooses the median of a random sample of size \( \sqrt{k} \) where \( k \) is the size of an input segment as the pivot value. The running time of the algorithm is \( O(m + n \log n \log \frac{m}{n}) \) for not too sparse graphs, where \( m = |E| \) and \( n = |V| \).

[110] compares Filter-Kruskal algorithm with Kruskal, qKruskal - a modified version of Kruskal, qJP - Jarnik-Prim implementation combined with Paredes’s quick-Heap priority queue, and pJP - Jarnik-Prim implementation with pairing heaps. This algorithm has been tested over a wide range of edge densities. The comparison has been done using five types of graphs, namely random graphs with random edge weights, difficult instances where decreaseKey method of JP algorithm is called for almost every edge, random geometric graphs, lollipop graphs and image segmentation instances as real instances.

Filter-Kruskal performs better than all other algorithms for random graphs with random edge weights. When the edge density is high, it is outperformed only by pJP. For difficult instances, JP and Filter-Kruskal perform in the same way, but Kruskal and qKruskal do better. Filter-Kruskal outperforms all the others for random geometric graphs with \( 2^{16} \).
nodes. pJP and qJP run faster than Filter-Kruskal by a small margin for lollipop graphs. For real instances, this algorithm performs the best.

5.3.2 Edge Pruned Minimum Spanning Tree (EPMST) Algorithm

Our EPMST algorithm uses ideas of randomized selection, Kruskal’s MST and Prim’s MST algorithms. The algorithm selects a portion of the input edges. For dense graphs with \( n \) vertices and \( m \) edges with random edge values, the \( n \log n \) lightest edges are more probable candidates to be minimum spanning tree edges. Therefore the algorithm chooses the \( n \log n \)th lightest edge as the pivot using a randomized selection algorithm. Then it identifies the edges having weights no greater than that of the pivot edge. It uses Kruskal’s algorithm on the graph with these edges only. Kruskal’s algorithm finds a minimum spanning forest. If the number of trees in this forest is 1, then the algorithm has already found a MST for the input graph and hence terminates. Otherwise it labels the connected components in this forest as super vertices and finds spanning tree using Prim’s algorithm.

**Algorithm 8 EPMST (Edge Pruned Minimum Spanning Tree)**

**Input:** A graph \( G(V, E) \)

**Output:** A list of edges of a minimum spanning tree

```plaintext
1: procedure EPMST
2: Let \( G(V, E) \) be the input graph. Choose a pivot edge using a randomized selection algorithm. This pivot edge is the \( cn \log n \)th lightest edge, where \( c \) is a constant
3: Identify edges having weights no greater than that of the pivot edge. Let \( E' \) be the set of these edges
4: Run Kruskal’s algorithm on \( G'(V, E') \) to get a forest \( F \)
5: Find the connected components in \( F \)
6: if the number of connected components in \( F \) is exactly one then
7: the connected component is the desired spanning tree
8: else
9: construct a graph \( G''(V'', E'') \) where each connected component of \( F \) is a vertex in \( V'' \). \((a, b) \in E'' \) if \((a, b) \) is an edge in \( G \) that connects the two components \( a \) and \( b \) and has the least weight among all the edges connecting \( a \) and \( b \)
10: Run Prim’s or Kruskal’s algorithm on \( G'' \)
11: end if
12: Output edges used in Prim’s or Kruskal’s algorithm on \( G(V, E) \)
13: end procedure
```
5.3.3 Analysis of Algorithm EPMST

Lemma 1  The expected run time of EPMST on any weighted graph $G(V, E)$ is $O(|E| + |V| \log^2 |V|)$. 
Here the expectation is over the space of all possible inputs.

Proof: Consider the random graph model $G(n, m)$, where $n = |V|$ and $m = |E|$. In this model we assume that each set (out of the $\binom{n^2}{m}$ possible such sets) of $m$ edges is equally likely. Assume that the input is a random graph from $G(n, m)$ in which the edge weights are chosen randomly.

Let $q$ be the number of edges whose weights are less than that of the pivot in EPMST. We can think of the subgraph induced by these edges as a $G(n, q)$. The reason is as follows. The edge with the least weight can be any one of the $m$ edges with equal probability, the edge with the next least weight could be any one of the remaining edges with equal probability, etc. As a result, these $q$ edges can be any $q$-subset of the $m$ edges with equal probability. This implies that the subgraph induced by the $q$ edges is indeed a $G(n, q)$.

Theorem VII.3 in [8] implies that $G(n, q)$ is connected with high probability when $q = \Omega(n \log n)$. This means that in EPMST when we call Kruskal’s algorithm, there will be only one connected component with high probability.

EPMST finds the edge with the $n \log n$th smallest weight as the pivot, using a randomized selection algorithm. Finding the pivot among the $m$ edges takes $O(m)$ time. Partitioning of the edges takes $O(m)$ time. Kruskal’s algorithm then runs on a graph with $n$ vertices and $O(n \log n)$ edges. Sorting within Kruskal’s algorithm takes $O(n \log n \log n)$ time. Overall, Kruskal’s algorithm takes this amount of time. With high probability, Kruskal’s algorithm finds a minimum spanning tree. Therefore there is no need to create super vertices and run Prim’s algorithm. Since this happens with a low probability, it follows that the expected run time of EPMST is $O(m + n \log^2 n)$. □

If in EPMST, Kruskal’s algorithm does not find the MST, an additional $O(m)$ time is needed to create super vertices. Kruskal’s algorithm outputs a number of connected components. Experiments show that the number of generated components in most cases is very low. Therefore, Prim’s algorithm on $G''(V'', E'')$ takes a negligible amount of time in practice.
5.3.4 Parallel EPMST algorithm

The more time consuming part of the sequential EPMST is partitioning of edges to choose a subset of edges for Kruskal’s method as it requires $O(|E|)$ time to scan through all the edges. If the input graph is dense, this is the most time taking phase compared to the other portions of the algorithm. In the parallel version a master processor distributes the edges among all the other processors. Each processor chooses a set of random edges and sends them to the master. The master collects all the edges from the slaves and chooses a pivot edge from this random sample. The master then broadcasts this pivot to all the slaves. Then each slave identifies edges whose weights are less than that of the pivot and sends these edges to the master. The master runs the Kruskal’s algorithm on the edges received from the slaves. Since the Kruskal’s method takes a negligible amount of time compared with the partitioning (done by the slaves), currently we keep this method sequential by calling this only from the master. Other portions of the algorithm are trivial as they do not consume comparable time. For random dense graphs with 10,000 nodes the speed up we get is 2.13 using 4 cores in a single node. The speed up is 2.40 and 2.79 for graphs with 20,000 and 30,000 nodes, respectively. Speedup using more cores is not promising since Kruskal’s algorithm run by the master takes time comparable to partitioning.

5.4 Experiments

We have implemented our EPMST algorithm in C++. We do not have access to any running version of Filter-Kruskal implementation. As a result, we have implemented this algorithm ourselves. In presenting timing results, we not only show the times taken by our implementation of Filter-Kruskal and EPMST but also the times reported in the Filter-Kruskal paper [110]. The timing results shown in [110] have been obtained on a machine with two AMD Opteron 2.0 GHz quad-core CPUs. We have run our implementations on a computer with 2.4 GHz Intel Core 2 Duo processors. We have thus scaled the running times (linearly) while reporting the timing results of [110]. It is true that this scaling may not be 100% accurate since factors such as cache memory size, main memory size, etc. also play a role. However with the absence of a running program for Filter-Kruskal, we
feel that this the best we could do. We also show timing results for our implementation of the Filter-Kruskal algorithm. GNU compiler and optimization level O3 have been used to compile our implementations. All the programs can be downloaded from: https://github.com/abdullah009/MinimumSpanningTree

We have used synthetic graphs to test the algorithms as real graphs are hard to find. We have generated five types of graphs. [105] describes generation processes of some types of graphs. Random graphs are generated by varying edge values within a range randomly. We use difficult instances of graphs where decreaseKey method of Prim’s algorithm is called for almost every edge. In geometric graphs, each point is connected with \( k \) nearest neighbors in the Euclidean space. A path of length \( \frac{n}{2} \) connected with a random graph of \( \frac{n}{2} \) nodes generates a difficult graph instance namely a lollipop graph. Our algorithm shows much better performance for dense graphs. We have used dense graphs having randomly generated edge values.

Figure 5.1: Time per edge for random graphs with \( 2^{10} \) nodes and random edge weights.

We have generated sparse graphs with random edge weights. Two graphs with different numbers of nodes have been employed in our experiments. To compare the performance with the Filter-Kruskal algorithm, we plot two lines for these two algorithms in all the figures. Our algorithm is denoted as EPMST and our implementation of the Filter-Kruskal algorithm is denoted as FilterKruskal. Another line is drawn to show (a scaled version of) the times reported in the Filter-Kruskal paper [110]. This line is denoted as FKGraph. Fig. 5.1 displays the running times of the algorithms for \( 2^{10} \) nodes and different numbers of edges per node. We choose \( 2n \) as the kruskalThreshold value. We use these many edges for the Kruskal method in the Filter-Kruskal algorithm. We use \( 2n \) as the kruskalThreshold value for all the graphs. Fig. 5.1 shows that the Filter-Kruskal algorithm works better.
when the edge per node value is below 32. Beyond that point, EPMST outperforms the former one. The figure clearly shows that our algorithm is at least two times faster than Filter-Kruskal when the edge per node value is 256. The trend also indicates that the margin will be higher for denser random graphs.

Table 5.1: Analysis of Different Phases of EPMST for Random Graphs with $2^{10}$ Nodes and Random Edge Weights

<table>
<thead>
<tr>
<th>$m/n$</th>
<th>T_Select</th>
<th>Sample Size</th>
<th>T_Kruskal</th>
<th>Num Trees</th>
<th>T_SV</th>
<th>T_Kruskal2</th>
<th>T_Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>238</td>
<td>2079</td>
<td>307</td>
<td>43</td>
<td>104</td>
<td>6</td>
<td>655</td>
</tr>
<tr>
<td>32</td>
<td>279</td>
<td>2033</td>
<td>301</td>
<td>34</td>
<td>110</td>
<td>5</td>
<td>695</td>
</tr>
<tr>
<td>64</td>
<td>377</td>
<td>1807</td>
<td>269</td>
<td>52</td>
<td>181</td>
<td>10</td>
<td>837</td>
</tr>
<tr>
<td>128</td>
<td>432</td>
<td>1986</td>
<td>282</td>
<td>25</td>
<td>210</td>
<td>5</td>
<td>929</td>
</tr>
<tr>
<td>256</td>
<td>592</td>
<td>2463</td>
<td>341</td>
<td>13</td>
<td>355</td>
<td>4</td>
<td>1292</td>
</tr>
</tbody>
</table>

Table 5.1 explains different phases of EPMST for random graphs with $2^{10}$ nodes and random edge weights. The table has eight columns, namely, number of edges per node, elapsed time for selection and partition methods, number of vertices used for Kruskal’s algorithm, time spent on Kruskal’s algorithm, number of trees in the forest after the Kruskal phase, time used for the creation of supervertices, time used on the second call to Kruskal’s method and the total time spent. The table is self explanatory.

Figure 5.2: Time per edge for random graphs with $2^{16}$ nodes and random edge weights.

Fig. 5.2 compares FilterKruskal and EPMST on random graphs with $2^{16}$ nodes. We see that these results are similar to the ones in Fig. 5.1. Again EPMST outperforms Filter-Kruskal when the number of edges per node is 16 or more. These figures also show that our implementation of Filter-Kruskal and the scaled values from the original paper of Filter-Kruskal algorithm follow almost the same path.

Another type of graph that Filter-Kruskal algorithm has used is the so called difficult
instance. The graph is also generated by producing random edge weights. An instance of this graph is such that each edge will call for the expensive decreaseKey operation in Prim’s algorithm. FKGraph takes much more time than FilterKruskal and EPMST. This is because of the difference between our input data sets. FilterKruskal works better when the number of edges per node is at most 64. After that point, EPMST takes less time. These comparisons have been shown in Fig. 5.3. Our EPMST consumes 14.78 ns per edge whereas FilterKruskal takes 22.49 ns when the edge per node value is 128. But this margin increases when this value is 256. FilterKruskal needs 20.96 ns for this case and EPMST takes only 9.78 ns.

Geometric graphs are generated by connecting each node with their $k$ nearest neighbors. As nodes are positioned randomly and nodes are connected with their $k$ nearest neighbors, desired edges have edge values within a range. Therefore the pivot value of our algorithm separates these edges efficiently. Fig. 5.4 illustrates this for random geometric graphs with $2^{16}$ nodes. All the three lines follow the same pattern as Fig. 5.1 and Fig. 5.2. EPMST outperforms FilterKruskal and FKGraph when the edge per node value is 16 or more.

A lollipop graph has the shape of a lollipop with a linear part and a circular part.
This graph has a single path among $\frac{n}{2}$ nodes which is very sparse and a dense part of the remaining $\frac{n}{2}$ nodes. Therefore, it is really difficult to choose a suitable pivot value to separate most of the desired edges for the Kruskal’s algorithm. But we can easily identify very sparse portions of the graph as well as the dense portions. Then we can apply the simple Kruskal algorithm on the very sparse components and EPMST on the dense components.

Fig. 5.5 and Fig. 5.6 show results for lollipop graphs, where we observe a similar trend for all of the curves. For these instances EPMST outperforms the Filter-Kruskal algorithm when the numbers of edges per node is 16 or more.

All these figures illustrate that our algorithm outperforms the Filter-Kruskal algorithm very clearly for not so sparse graphs. The trend shows that if the graphs are more dense, then our algorithm works much better. When the number of edges per node increases, our algorithm outperforms Filter-Kruskal with an increasing performance gap.

EPMST performs its best for random dense graphs. Fig. 5.7 compares the performance of the algorithm with Filter-Kruskal for random dense graphs with different numbers of nodes. $X$ axis shows the number of nodes and $Y$ axis shows time in seconds. For these
graphs, we choose the $n \log n^{th}$ lightest edge as the pivot for EPMST. In this graph, we see that EPMST runs 20 times faster than Filter-Kruskal for complete graphs with 30,000 nodes.

We have employed five types of input graphs to compare EPMST with Filter-Kruskal. We provide the total time consumed by EPMST and Filter-Kruskal on dense graphs. In the other figures we show the time spent per edge to compute a minimum spanning tree. For very sparse graphs both of our algorithms do not take much time. Even any simple minimum spanning tree algorithm works fine for very sparse graphs. But our EPMST outperforms Filter-Kruskal for not too sparse graphs. The margin between these two algorithms increases with an increment on the edge per node value. Filter-Kruskal algorithm has better performances over Prim’s and Kruskal’s algorithms for most of the cases [110]. Therefore we have only compared our algorithm with the Filter-Kruskal algorithm.
Chapter 6

Efficient sequential and parallel algorithms for estimating higher order spectra

6.1 Introduction

Fast computation of HOS such as the bispectrum and the trispectrum becomes especially crucial for long nonlinear time series. For example, intra-day financial data analysis usually involves very long time series of stock or index returns or time durations between events of interest such as price or volume changes, see [149]. Typically, depending on the liquidity of a stock, the time series length within a single day can be as high as 20,000 or more. [126] discusses the use of HOS for monitoring the condition of rotating machinery due to cracks whose signatures are captured as long nonlinear time series consisting of 2,560 observations per second. Existing algorithms are very slow. For instance, the MATLAB code [131] to compute the bispectrum takes 23 seconds on an input of size 2,048. In the application of cracks and misalignment detection [126], if we collect samples for one hour, the sample size will be more than 9 million and the MATLAB code will take an estimated time of more than 14 years! Thus it is essential to improve existing sequential algorithms, see [50]. It is also important to develop effective parallel algorithms. Existing parallel algorithms
are either inefficient or apply to only specific architectures. In this research work we offer general parallel algorithms that are very efficient.

If $X(i)$ is a stationary random process ($i$ denoting discrete time), the moments of order $k$ are given by [108]:

$$m_k^X(w_1, w_2, \ldots, w_{k-1}) = E[X(i)X(i+w_1)X(i+w_2)\cdots X(i+w_{k-1})].$$

Cumulants are functions of the moments. For example, the first order cumulant is defined as $c_1^X = m_1^X = E[X(i)]$, the second order cumulant as $c_2^X(w_1) = m_2^X(w_1) - (m_1^X)^2$, and so on. The moments and cumulants defined above are based on expectations over the (infinite) ensemble. For ergodic processes, these ensemble averages may be estimated using the corresponding time averages. The Fourier transform of the third and fourth cumulants are respectively the bispectrum and the trispectrum. The problem we address is the following: Given a finite sequence $X(1), X(2), \ldots, X(n)$, compute smoothed sample bispectrum and trispectrum which are statistically consistent estimates of the corresponding true HOS.

HOS are useful in unsupervised and supervised classification of long sequences of non-linear time series with applications in finance, geoscience, neuroscience, etc. HOS can also be used as a test for Gaussianity of any data, since if $X(i)$ is Gaussian, the cumulant $c_k^X(w_1, w_2, \ldots, w_{k-1}) = 0$ for $k > 2$ [108, 119]. HOS can also be used to test for linearity of any data. Other applications include characterization of coronary artery disease [2], analysis of breast thermograms [1], communication systems [123], etc.

In this chapter, a major part is devoted to a discussion on computing the bispectrum. However the techniques proposed extend to trispectra as well (as we explain toward the end).

**Direct Method for HOS:** Two kinds of algorithms can be found in the literature for HOS: direct method using the Fast Fourier Transform (FFT) and an indirect method via the Fourier transform of the third cumulant. Here we use the direct method. However, the algorithms we propose can be used for the indirect method as well. The following summary of the direct method can be found in [108]. Let $X(1), X(2), \ldots, X(n)$ be the input sequence. The direct bispectrum (DBS) method works as follows:
1) Partition the input into \( K \) parts with \( M \) samples in each part. Let \( X_i \) stand for the \( i \)-th part, for \( 1 \leq i \leq \frac{n}{M} \).

2) In each part subtract the mean of that part from each element in the part.

3) Compute the Discrete Fourier Transform \( F^i_X(k) \) for each part: 
\[
F^i_X(k) = \sum_{u=0}^{M-1} X^i(u) e^{-j \frac{2\pi}{M} uk},
\]
for \( k = 0, 1, \ldots, M - 1 \); \( i = 1, 2, \ldots, K \), and \( j = \sqrt{-1} \).

4) Estimate the raw bispectrum of each part as:
\[
C^X_i(k_1, k_2) = \frac{1}{M} F^1_X(k_1) F^1_X(k_2) F^i_X(k_1 + k_2),
\]
for \( i = 1, 2, \ldots, K \). Due to various symmetries, it suffices to compute the bispectrum \( C^X_i(k_1, k_2) \) only in the principal domain: \( 0 \leq k_2 \leq k_1, k_1 + k_2 < M/2 \).

5) This step performs some smoothing over a window of size \( M_3 \times M_3 \) and yields a consistent estimate of the true bispectrum:
\[
\tilde{C}^X_3(k_1, k_2) = \frac{1}{M_3} \sum_{n_1=-M_3/2}^{M_3/2-1} \sum_{n_2=-M_3/2}^{M_3/2-1} C^X_i(k_1 + n_1, k_2 + n_2).
\]

6) The estimated bispectrum of the entire time series is computed as the average over all parts: 
\[
\hat{C}(w_1, w_2) = \frac{1}{K} \sum_{i=1}^{K} \tilde{C}^X_3(w_1, w_2).
\]

**Time Complexity Analysis:** Step 2 in the direct method takes \( O(n) \) time. Step 3 takes a total of \( O(n \log M) \) time. Step 4 takes \( O(M^2) \) time per part. Thus Step 4 takes a total of \( O(KM^2) = O(Mn) \). In Step 5 smoothing is done. For every point \( (k_1, k_2) \), the smoothed value \( \tilde{C}^X_3(k_1, k_2) \) is computed as the average value of \( C^X_i(k_1, k_2) \) over a region of size \( O(M_3^2) \). Thus each such computation takes \( O(M_3^2) \) time. The total time taken in Step 5 is \( O(M^2KM_3^2) = O(MnM_3^2) \).

In summary, the total run time of the direct method is \( O(MnM_3^2) \). In this work we show that this run time can be improved to \( O(Mn) \). Note that this run time is independent of \( M_3 \).

**Known parallel algorithms for HOS:** We summarize below some of the the known algorithms. As we can see, these algorithms are very inefficient and restricted to specific architectures.

Manolakos, et al. [92] discuss the importance of power spectra in signal processing. Followed by this, they employ the canonical mapping methodology (CMM) to derive par-
allel programs for computing bispectrum. This paper focused exclusively on the design of the systolic array and no experimental results were presented. In [63], the authors present data parallel algorithms for computing 3rd and 4th order moments on the MasPar-1 SIMD parallel system. Their program handles input sequences of length up to $2^{10}$. Their algorithm can be thought of as a mesh algorithm. No time complexity analyses were given in the chapter and the algorithm was very specific for the MasPar-1 machine. In [77] also, the authors consider the parallel computation of bispectrum. They have implemented the direct and the indirect methods using two different parallel programming techniques: semi-automatic and fully automatic using the Power C Analyzer. The machine used was the Silicon Graphics Power Challenge MIMD Machine HOTBLACK. This work also falls under the category of developing a parallel program for a specific machine. In [43] and [42] the authors consider parallel reconstruction of images using bispectra. They parallelize the bispectrum algorithm in a straight forward manner without worrying about achieving optimal run times.

**Contributions of this work:** None of the above papers deals with the problem of constructing smoothed sample HOS which are consistent estimates of the true HOS. In this research work our focus is on developing generic parallel algorithms that can be employed on any parallel machine or platform. We also provide experimental evaluations of our algorithms. For HOS computing algorithms one of the major bottlenecks could be in the memory needed. For computing order $k$ moments the memory needed is $\Omega(n^{k-1})$. This could indeed be prohibitive. For example, when $k = 3$ and $n = 10^6$, the memory needed will be at least 1,000 GB. Thus it is essential to develop memory efficient algorithms. Here we address this crucial problem. Also, for bispectrum computation with smoothing over a window of size $M_3$, existing algorithms take $O(nMM_3^2)$ time. In this article we present sequential and parallel algorithms that do only $O(nM)$ work. Here $M$ is the size of each part of the input.
6.2 A Better Algorithm for the Direct Method

In this section we show how to improve the run time of the direct method from $O(MnM_d^2)$ to $O(Mn)$. The new algorithm is based on an efficient way of computing window sums that we describe next.

6.2.1 Computing window sums

The case of 1D data: Let $X = k_1, k_2, \ldots, k_n$ be any sequence of real numbers and let $w$ be a window size. The problem is to compute $s_i = \sum_{j=1}^w k_{i+j-1}$, for $1 \leq i \leq (n - w + 1)$.

A straightforward algorithm for this problem will take $O(nw)$ time. We can improve this to $O(n)$ using overlaps in successive window sums. Specifically, $s_{i+1} = s_i - k_i + k_{i+j}$, for $1 \leq i \leq (n - w)$. This means that $s_{i+1}$ can be obtained from $s_i$ in $O(1)$ time. Therefore, if we compute the window sums in this order: $s_1, s_2, \ldots, s_{n-w+1}$, then we can compute all of them in $O(n)$ time.

The case of 2D data: The above idea can be extended to 2D data as well. Let $A = (a_{i,j})$ be an $n \times n$ matrix and let $w$ be a window size. Consider the problem of computing $s_{i,j} = \sum_{u=1}^w \sum_{v=1}^w a_{i+u-1,j+v-1}$, for $1 \leq i \leq (n - w + 1)$ and $1 \leq j \leq (n - w + 1)$.

A trivial algorithm for solving the above problem will take $O(n^2w^2)$ time. We can improve this run time to $O(n^2)$ as follows.

Analysis: The total run time of the above algorithm is $O(n^2)$.

6.2.2 Direct Method for Bispectrum

We can employ the above window sums algorithms in the smoothing step (5) of the direct method. In this case we get the following theorem.

Theorem 2 We can compute bispectrum of any input of size $n$ using the direct method in $O(Mn)$ time, $M$ being the partition size. □
6.3 Parallel Models and Preliminaries

In this section we describe the parallel models of computing that we employ, namely, the PRAM and the mesh. A Parallel Random Access Machine (PRAM) is a collection of RAMs working in synchrony where communication takes place with the help of a common block of shared memory [56, 51]. Depending on how read and write conflicts are handled, a PRAM can further be classified into three: Exclusive Read and Exclusive Write (EREW) PRAM, Concurrent Read and Exclusive Write (CREW) PRAM, and Concurrent Read and Concurrent Write (CRCW) PRAM. There are variants of a CRCW PRAM depending on how write conflicts are handled. In a Common-CRCW PRAM, concurrent writes are permissible only if the processors trying to write in the same cell at the same time have the same data to write. In an Arbitrary-CRCW PRAM, if more than one processor tries to write in the same cell at the same time, an arbitrary one of them succeeds. In a Priority-CRCW PRAM, processors have assigned priorities. Write conflicts are resolved using these priorities.

An $n \times n$ mesh can be represented as a directed $n \times n$ grid-graph whose nodes correspond to processing elements and whose edges correspond to bidirectional communication links [51]. If two processors are connected by an edge, they can communicate in a unit step. Otherwise, they communicate by sending a message along a connecting path. The work done by a parallel algorithm that uses $P$ processors and runs in time $T$ is defined as the product $P \times T$.

Let $\oplus$ be any associative unit-time computable binary operator defined in some domain $\Sigma$. Given a sequence of $n$ elements $k_1, k_2, \ldots, k_n$ from $\Sigma$, the problem of prefix computation is to compute $k_1, k_1 \oplus k_2, \ldots, k_1 \oplus k_2 \oplus \cdots \oplus k_n$. Proof of the following Lemma can be found in relevant texts (such as [56, 51]).

**Lemma 3** Prefix computation on a sequence of $n$ elements can be performed in $O(\log n)$ time using $\frac{n}{\log n}$ EREW PRAM processors.
6.3.1 Window sums on the PRAM

We now show how to implement the direct method on an EREW PRAM optimally. First we consider the computation of window sums in 1D and 2D.

The case of 1D data in parallel: Let \( X = k_1, k_2, \ldots, k_n \) be any sequence of real numbers and let \( w \) be a window size. The problem is to compute \( s_i = \sum_{j=1}^{w} k_{i+j-1} \), for \( 1 \leq i \leq (n - w + 1) \).

A straightforward PRAM algorithm for this problem could use \( (n - w + 1) \) processors. Each processor can in parallel compute one window sum in \( O(w) \) time. The work done will be \( O(nw) \). We can improve these bounds using the prefix computation.

1) Perform a prefix sums computation on \( k_1, k_2, \ldots, k_n \).
   
   Let the results be \( q_1, q_2, \ldots, q_n \); Let \( q_0 = 0 \);

2) for \( i = 1 \) to \( (n - w + 1) \) in parallel do
   
   3) \( s_i = q_{i+w-1} - q_{i-1} \);

Analysis: Step 1 can be done using \( \frac{n}{\log n} \) EREW PRAM processors in \( O(\log n) \) time (c.f. Lemma 3). The for loop of line 2 can be performed in \( O(1) \) time using \( n \) EREW PRAM processors. Using the slow-down lemma (see e.g., [56, 51]), Step 2 can also be completed in \( O(\log n) \) time using \( \frac{n}{\log n} \) processors. Thus we arrive at the following lemma.

Lemma 4 The window sums computation problem on any input sequence of length \( n \) can be solved in \( O(\log n) \) time using \( \frac{n}{\log n} \) EREW PRAM processors. \( \square \)

The case of 2D data in parallel: Let \( A = (a_{i,j}) \) be an \( n \times n \) matrix and let \( w \) be a window size. We are interested in computing \( s_{i,j} = \sum_{u=1}^{w} \sum_{v=1}^{w} a_{i+u-1,j+v-1} \), for \( 1 \leq i \leq (n - w + 1) \) and \( 1 \leq j \leq (n - w + 1) \).

A trivial algorithm for solving the above problem will do \( O(n^2w^2) \) work. We can improve this work to \( O(n^2) \) as follows. In this algorithm, \( t_{i,0} = 0 \), for \( 1 \leq i \leq (n - w + 1) \).

Analysis: In line 1, for a specific value of \( j \), window sums can be computed in \( O(\log n) \) time using \( \frac{n}{\log n} \) EREW PRAM processors (c.f. Lemma 4). Thus the for loop of line 1 can be completed in \( O(\log n) \) time given \( \frac{n^2}{\log n} \) EREW PRAM processors.

In line 5, for any given value of \( i \), prefix sums computation can be performed in \( O(\log n) \) time using \( \frac{n}{\log n} \) EREW PRAM processors (c.f. Lemma 3). As a result, the for loop of line
5 takes \(O(\log n)\) time given \(\frac{n^2}{\log n}\) EREW PRAM processors.

Line 11 can be performed (for a given \(i\) and \(j\)) in \(O(1)\) time using one processor. Therefore, the for loop of line 9 can be performed in \(O(1)\) time given \((n - w + 1)^2\) EREW PRAM processors. Using the slow-down lemma, the for loop of line 9 can also be completed in \(O(\log n)\) time using \(\frac{n^2}{\log n}\) processors.

Put together, the above algorithm runs in a total of \(O(\log n)\) time using \(\frac{n^2}{\log n}\) EREW PRAM processors. Clearly, this algorithm is asymptotically work-optimal. We arrive at the following lemma:

**Theorem 5** The window sums computation problem can be solved in \(O(\log n)\) time using \(\frac{n^2}{\log n}\) EREW PRAM processors. □

### 6.3.2 Direct method for bispectrum on a PRAM

In this section we present a PRAM algorithm for direct bispectrum computation. There are 5 steps in the algorithm (c.f. Algorithm DBS). We discuss how to parallelize each step.

Let \(X(1), X(2), \ldots, X(n)\) be the input sequence.

Step 1 is that of partitioning the data into \(K\) parts and this does not cost any time since the input will be given in the common memory. Let the parts be \(X_i^i, 1 \leq i \leq K\).

In Step 2, finding the mean of \(X_i\) can be done in \(O(\log M)\) time using \(\frac{M}{\log M}\) processors, for a specific \(i\). Thus the mean of all the parts can be found in \(O(\log M)\) time using \(\frac{n}{\log M}\) processors. Using the slow down lemma, Step 2 can be performed in \(O(\log n)\) time using \(\frac{n}{\log n}\) EREW PRAM processors.

Step 3 involves the computation of the Discrete Fourier Transform (DFT) \(F_X^i(k)\) for each part: 
\[
F_X^i(k) = \sum_{u=0}^{M-1} X^i(u) e^{-j \frac{2\pi}{M} uk}, \quad \text{for } k = 0, 1, \ldots, M - 1; i = 1, 2, \ldots, K.
\]
For each part, the time taken is \(O(\log M)\) using \(M\) processors (see e.g., [56, 51]). Therefore, the DFT of all the parts can be computed in \(O(\log M)\) time using \(n\) processors.

We have to estimate the third order spectrum of each part in Step 4. Specifically, we have to compute \(C_X^3(k_1, k_2)\), for \(1 \leq i \leq K\) and \(0 \leq k_2 \leq k_1, k_1 + k_2 < M/2\). This can be done in \(O(1)\) time using \(O(nM)\) processors. Equivalently, Step 4 can also be done in \(O(\log n)\) time using \(\frac{nM}{\log n}\) EREW PRAM processors (using the slow down lemma).
Step 5 is concerned with the smoothing operation. The value of the bispectrum at any point is computed as an average over a surrounding window of size $M_3 \times M_3$. This Step can be performed using the Algorithm WS_PRAM (c.f. Theorem 5). For each part, this Step can be completed in $O(\log M)$ time using $\frac{M^2}{\log M}$ processors. For all the $K$ parts together, Step 5 takes $O(\log M)$ time using $\frac{nM}{\log M}$ processors. Using the slow down lemma, Step 5 can be completed in $O(\log n)$ time employing $\frac{nM}{\log n}$ processors.

In Step 6, the bispectrum is computed as the average over all parts. In particular, we have to compute $\hat{C}(w_1, w_2) = \frac{1}{K} \sum_{i=1}^{K} \tilde{C}^X_i(w_1, w_2)$. For a given $w_1$ and $w_2$, $\hat{C}(w_1, w_2)$ can be computed using a prefix sums computation on $K$ elements and hence can be done in $O(\log K)$ time using $\frac{K}{\log K}$ processors. Thus Step 6 can be completed in $O(\log K)$ time using $\frac{M^2K}{\log K} = \frac{nM}{\log K}$ processors. The slow down lemma implies that Step 6 can also done in $O(\log n)$ time using $\frac{nM}{\log n}$ processors.

In summary, we get the following theorem.

**Theorem 6** We can compute the bispectrum on any sequence of length $n$ in $O(\log n)$ time using $\frac{nM}{\log n}$ EREW PRAM processors, where $M$ is the size used to partition the input sequence. □

The following theorems pertain to computing the bispectrum computation in a memory efficient manner. Proofs are omitted due to space constraints and will be supplied in the full version.

**Theorem 7** We can solve the window sums problem on any $n \times n$ matrix in $O(n \log n)$ time using $\frac{n}{\log n}$ EREW PRAM processors using only $O(nw)$ memory, $w$ being the window size. □

**Theorem 8** Bispectrum computation on any given sequence of length $n$ can be computed in $O(n \log n)$ time using $\frac{M}{\log n}$ EREW PRAM processors and $O(M M_3)$ memory, where $M$ is the size of each part and $M_3$ is the window size of smoothing (assuming that $M = \omega(\log n)$). □

**Theorem 9** Window sums on any $n \times n$ data can be computed in $O \left( \frac{n^2}{w^2} \log n \right)$ time using $\frac{w^2}{\log n}$ EREW PRAM processors and $O(w^2)$ memory, $w$ being the window size. □
Theorem 10  Window sums on any $n \times n$ data can be computed in $O(n^2 \log n)$ time using $\frac{w}{\log n}$ EREW PRAM processors and $O(w)$ memory, $w$ being the window size. □

Note that the work done in the above algorithm is $O(n^2w)$ and hence the algorithm is not work optimal. However, the memory used is very small. Theorems 5, 7, 9 and 10 consider memories of different specific sizes. Theorems 5 and 9 can be used to develop work optimal algorithms when the memory available is $m$ for any $w^2 \leq m \leq n^2$. The following theorems consider the mesh model and higher order spectra, respectively. Proofs are omitted due to space constraints.

Theorem 11  Bispectrum computation of a sequence of length $n$ can be performed on an $n \times n$ mesh in $O(n)$ time. □

Theorem 12  On any input of size $n$, we can compute $k$th order spectrum in $O(nM^{k-2})$ time, for any $k \geq 3$ where $M$ is the size of each part in the input. □

6.4 Experimental Results

We have conducted extensive experiments to evaluate the performance of our proposed approaches. In this section we report the results.

6.4.1 Test Platform

All the experiments have been performed on the test server, which is equipped with Intel(R) Xeon(R) CPU E5-2667 v3 @ 3.20GHz, with 16 cores (Hyperthreading to 32 threads), 256 GB main memory and 4 TB HDD disk. All the algorithms have been implemented using C++ and the standard GCC compiler. The parallel version is implemented using OpenMP. We have used a value of $K = 1$ throughout. We have generated different types of the time series data sequences for our experiments using guidelines given in [44, 60].

We have implemented algorithms for spectral computation for bispectrum and trispectrum computations. For both of them, we have compared 5 different approaches: Naive approach with $O(n^2m^2)$ run time, denoted as Naive. Here $m$ is nothing but $M_3$; Our sequential algorithm that takes $O(n^2)$ time (c.f. Theorem 2) - Call this algorithm as WS in
consistent with above sections; Our fastest algorithm of Theorem 8 that does $O(n^2)$ work and uses $O(nm)$ space - Call this algorithm Fast; The most efficient algorithm in both time and memory ($O(n^2)$ time, $O(m^2)$ memory) - Call this algorithm Efficient; Parallel approach (denoted as Parallel) with $P$ threads, $P = 2, 4, 8, 16$.

### 6.4.2 Run Time and Memory Comparisons

We have set a run time threshold of 10 hours and a memory threshold of 100 GB. Any algorithm exceeding one or both of these thresholds was forced to stop. For large datasets such as those with $n = 2^{14}, 2^{15}$, some of the algorithms exceeded these thresholds. In Figure 6.1 we show the running time of different approaches for bispectrum and trispectrum, respectively. Note that this is a log plot. Thus the parallel curves show orders of magnitude difference. From this figure we can clearly see that compared with the naive algorithm, all of our algorithms offer much better run times.

![Figure 6.1: Run time comparison for different values of n](image)

We provide the maximum memory cost during the running time of each algorithm in Table 6.1. From this table we see that the memory cost in our experiments is consistent with our theoretical analyses. As it is shown, the Efficient approach is extremely memory efficient. For instance, for the time series sequence with a length of $n = 2^{16}$, it only requires less than 100 MB of memory. In contrast, even the second memory efficient approach Fast uses around 1 GB, and the others occupy more than 30 GB. Efficient would take a longer time than Fast, which demonstrates the trade-off between space and time.

We have compared running times of bispectrum computation by our Fast implementation and HOSA Toolbox in MATLAB [131] for single thread. The results from the two
Table 6.1: Memory (in MB) Comparison

<table>
<thead>
<tr>
<th>n</th>
<th>Naive</th>
<th>WS</th>
<th>Fast</th>
<th>Efficient</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Bispectrum</td>
</tr>
<tr>
<td>2^{12}</td>
<td>504.8</td>
<td>504.7</td>
<td>14.5</td>
<td>5.2</td>
</tr>
<tr>
<td>2^{13}</td>
<td>2030.7</td>
<td>2030.5</td>
<td>38.6</td>
<td>9.0</td>
</tr>
<tr>
<td>2^{14}</td>
<td>8176.3</td>
<td>8176.3</td>
<td>113.8</td>
<td>17.6</td>
</tr>
<tr>
<td>2^{15}</td>
<td>NA</td>
<td>32890.0</td>
<td>344.3</td>
<td>37.9</td>
</tr>
<tr>
<td>2^{16}</td>
<td>NA</td>
<td>NA</td>
<td>1055.8</td>
<td>86.3</td>
</tr>
<tr>
<td>n</td>
<td>Naive</td>
<td>WS</td>
<td>Fast</td>
<td>Efficient</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Trispectrum</td>
</tr>
<tr>
<td>2^{17}</td>
<td>56.8</td>
<td>56.7</td>
<td>8.4</td>
<td>3.6</td>
</tr>
<tr>
<td>2^{18}</td>
<td>448.0</td>
<td>448.0</td>
<td>38.2</td>
<td>7.3</td>
</tr>
<tr>
<td>2^{19}</td>
<td>3682.7</td>
<td>3682.7</td>
<td>215.8</td>
<td>19.3</td>
</tr>
<tr>
<td>2^{20}</td>
<td>NA</td>
<td>30224.3</td>
<td>1297.7</td>
<td>64.6</td>
</tr>
<tr>
<td>2^{21}</td>
<td>NA</td>
<td>NA</td>
<td>7869.3</td>
<td>228.4</td>
</tr>
</tbody>
</table>

programs match exactly. Table 6.2 shows how fast our Fast implementation becomes when series lengths increase. We have computed bispectrum for every pair of frequencies with linear smoothing window using both of these implementations. In our experiments with HOSA Toolbox, we supplied 0.0 for overlap value and series length as segment size.

Table 6.2: Comparison of running times (in sec) of Fast and HOSA Toolbox

<table>
<thead>
<tr>
<th>series length</th>
<th>window length</th>
<th>Fast</th>
<th>HOSA Toolbox</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>21</td>
<td>0.001</td>
<td>0.010</td>
</tr>
<tr>
<td>256</td>
<td>33</td>
<td>0.005</td>
<td>0.042</td>
</tr>
<tr>
<td>512</td>
<td>49</td>
<td>0.011</td>
<td>0.220</td>
</tr>
<tr>
<td>1,024</td>
<td>77</td>
<td>0.032</td>
<td>1.755</td>
</tr>
<tr>
<td>2,048</td>
<td>117</td>
<td>0.126</td>
<td>23.342</td>
</tr>
<tr>
<td>4,096</td>
<td>181</td>
<td>0.448</td>
<td>329.961</td>
</tr>
<tr>
<td>8,192</td>
<td>279</td>
<td>1.751</td>
<td>3102.4</td>
</tr>
</tbody>
</table>

6.4.3 Multi-core Parallel Approach Evaluation

Next, we evaluate our proposed parallel algorithm. Due to the fact that the Efficient approach has a significant advantage in memory, we have implemented the parallel version of Efficient to offer a fast and memory efficient approach in high order spectra computations. We have tested the Parallel algorithm using \( P = 2, 4, 8, 16 \) and \( n \) from \( 2^7 \) to \( 2^{15} \).

In Figure 6.2, we plot the parallel speedup against number of cores \( P \). As we can see, from 2 cores to 16 cores (\( \log_2(P) = 1, 2, 3, 4 \)), the speedup is increasing if the number of cores is increasing. Sometimes super-linear speedup could also be achieved for 2 cores and 4 cores.

Another interesting fact is that for small data lengths (small \( n \) values), the gain of
multi-cores is not as significant as for larger \( n \) values. This is due to the overhead introduced in multi-core implementations, such as processor scheduling and communication. For instance, in the case of \( n = 2^7 \) of bispectrum computation, our 2-core \textbf{Efficient} approach has a run time of around 1 ms. However, using 16-threads \textbf{Parallel} still took 1 ms to finish. Thus for small datasets, the overhead of work scheduling becomes dominant.

On the contrary, the computation time is still the dominant part for large datasets, e.g., more than 10 hours for \( n = 2^{15} \) using the sequential algorithm. This makes our parallel algorithm especially useful for large datasets.

From the memory point of view, the memory cost for the parallel implementation is linearly dependent on the number of cores. This is due to the fact that each processor is independently working on its own smoothing window.

### 6.4.4 Summary of Experiments

We have evaluated four approaches, \textbf{Naive}, \textbf{WS}, \textbf{Fast}, and \textbf{Efficient}, respectively, as well as the \textbf{Parallel} algorithm. Both bispectrum and trispectrum implementations have been tested on different lengths of time series data.

All of our proposed algorithms outperform the \textbf{Naive} algorithm by orders of magnitude, in terms of both run time and memory. Please note that the naive algorithm is the best found in the literature. \textbf{Fast} is \( 25 \times 10^3 \) times faster when \( n = 2^{15} \), and \textbf{Efficient} uses less than 1/200th of the memory used by the naive algorithm. Even though \textbf{WS} is simpler, \textbf{Fast} runs the fastest. It could be due to the cache misses and memory accessing time costs, as \textbf{WS} occupies a significantly larger memory. \textbf{Fast} and \textbf{Efficient} display a memory-time
trade-off. **Efficient** has a better balance and is extremely frugal in memory usage.

**Parallel** is a fast and memory saving algorithm, suitable for problems with very large \( n \). A linear speedup can be achieved by **Parallel** on larger datasets. The memory cost for **Parallel** is also linearly dependent on \( P \). For large \( n \), **Parallel**’s performance is better than for small \( n \). This is due to the overhead of parallel implementation, making the parallel approach more preferred for large datasets. Large data sets are quite relevant in today’s world of big data.
Chapter 7

Efficient algorithms for finding the closest $l$-mers in biological data

7.1 Introduction

Large amounts of data get generated in every area of science and engineering. This is especially true in the biological domain. Currently, the bottleneck is not in generating data but is in processing these data. Efficient big data analytics algorithms are called for. A powerful analytics paradigm is patterns finding. In this chapter we study an important pattern that can be used to solve many other problems including motif search. Specifically, we investigate the problem of finding the closest $l$-mers in an input of strings. The biological strings could be DNA sequences, protein sequences, etc. Algorithms for finding the closest $l$-mers have been used to solve the $(l,d)$-motif search problem, see for example [114, 28].

The pattern finding problem of interest can be stated as follows. The input are $m$ biological sequences $S_1, S_2, \ldots, S_m$, each of length $n$, and an integer $l$. The problem is to find $m$ $l$-mers $X_1, X_2, \ldots, X_m$ such that $X_i$ is in $S_i$ (for $1 \leq i \leq m$) and the Hamming distance among these $l$-mers is the least (from out of all such $l$-mers). $X$ is an $l$-mer in a sequence $S$ if $X$ is a subsequence of $S$ of length $l$. Each input sequence can be thought of as a string of characters from a finite alphabet $\Sigma$. For instance, each input sequence could be a DNA sequence or a protein sequence. We refer to this pattern finding problem as the
The closest l-mers problem (CLP). If $X_i = x_1^i x_2^i x_3^i \ldots x_l^i$, for $1 \leq i \leq m$, are any l-mers, then the Hamming distance among them is defined as $\sum_{j=1}^{l} d(x_1^j, x_2^j, \ldots, x_m^j)$ where $d(x_1^j, x_2^j, \ldots, x_m^j)$ is zero if all of the characters $x_1^j, x_2^j, \ldots, x_m^j$ are the same; and $d(x_1^j, x_2^j, \ldots, x_m^j)$ is 1 otherwise.

The longest common substring (LCS) problem could be viewed as a dual version of CLP. While CLP finds l-mers that are the closest for a given l, LCS finds the length of the longest common substring. Some relevant papers are: [39, 79]. Another related problem is finding the closest pair of points. CLP could be viewed as a special case of CP. Numerous papers have been written on this problem (see e.g., [32, 24, 53, 132]).

A special case of the CLP when $m = 2$ has been studied in the literature before. For instance, [114] shows that this problem can be solved in $O(n^2)$ time for $m = 2$, where $n$ is the length of each of the two input sequences. Note that a trivial algorithm to solve this problem will examine each pair of l-mers A and B where A comes from the first sequence and B comes from the second sequence, compute the Hamming distance between A and B, and output the pair of l-mers with the least distance. This brute force algorithm runs in time $O(n^2l)$. The $O(n^2)$-time algorithm has been used in solving the $(l, d)$-motif search problem (see e.g., [114, 28]). Time series motif mining could be viewed as a special case of CLP, and many algorithms have been recently used to solve this problem, such as FFT technique in [84] and $O(n^2)$ methods in [144] [148], and embedding-based approach in [111].

The case of $m > 2$ is very important as well. For instance, in the case of $(l, d)$-motif search, an algorithm for the case of $m > 2$ can be used in the algorithms of [114] [28] in which case the performance of these algorithms will improve. Also, for the time series motif mining problem, $m$ being more than 2 can provide deeper insights. The problem of time series motif mining can be thought of as that of detecting two events (that occur in two different times) that are very similar to each other. Equally (and perhaps more) important will be the problem of detecting $m$ (> 2) events that are very similar among themselves.

In this work we present novel algorithms for solving the CLP when $m = 3$. We refer to this special case of the CLP as the closest triplet problem. Specifically, we offer three different algorithms. Two of these are exact and the third one is approximate. An algorithm is exact if it always outputs the closest l-mers. On the other hand, an
approximate algorithm may not output the closest \( l \)-mers all the time. In general it outputs \( l \)-mers whose distance is very nearly the same as that of the closest \( l \)-mers. There is a closely related problem that \( l \)-mers could come from the same sequence, and we also extend our algorithms to address this problem, by putting one additional constraint that the \( l \)-mers should not overlap.

The CLP has many applications. From among these, the \((l,d)\)-motif search is an important problem since motifs can be used to identify transcription factors and their binding sites, composite regulatory patterns, similarity between families of proteins, etc. The \((l,d)\)-motif search (LDMS) problem is stated as follows: Input are \( n \) sequences \( S_1, S_2, \ldots, S_n \) and integers \( l \) and \( d \). The task is to find all the strings \( M \) of length \( l \) such that \( M \) occurs in each of the input sequences within a Hamming distance of \( d \). Each such string \( M \) is called an \((l,d)\)-motif. This problem is known to be \( \mathcal{NP} \)-hard. The algorithm of [114] uses the \( O(n^2) \)-time closest \( l \)-mers algorithm as a crucial step in solving the LDMS problem. Subsequently, this closest \( l \)-mers algorithm has also been used in the PMSPrune algorithm of [28] for solving the LDMS problem.

Another important application is that of finding time series motifs. The problem of finding time series motifs can be stated as follows: We are given a sequence \( S \) of real numbers and an integer \( l \). The goal is to identify two subsequences of \( S \) of length \( l \) each that are the most similar to each other (from among all pairs of subsequences of length \( l \) each) [106]. These most similar subsequences are referred to as time series motifs.

When CLP is defined for biological sequences the distance of interest is the Hamming distance. On the other hand, time series data are sequences of real numbers. The distance between two \( l \)-mers has to be modified. Several possibilities such as Euclidean distance and Pearson’s correlation coefficient have been explored in the literature (see e.g., [15, 106, 148]).

When we extend the CLP for \( m > 2 \), we have to revisit the notion of distance. When the input has biological sequences, we can continue to use Hamming distance as defined above. If the input consists of time series data, many possibilities arise. Consider the case of \( m = 3 \). Let \( X \), \( Y \), and \( Z \) be any three \( l \)-mers. Then, one possible distance among these three is the **pairwise-sum distance** \( d(X, Y, Z) = d(X, Y) + d(Y, Z) + d(Z, X) \) where \( d(X, Y) \) is the Euclidean distance between \( X \) and \( Y \). Hamming distance could be also
calculated in a pairwise-sum manner. Thus we refer to the previous definition of Hamming distance as **direct** Hamming distance of a tuple. Other distance metrics are also possible.

The rest of this chapter is organized as follows. In Section 7.2 we first review existing algorithms for CLP when \( m = 2 \). This special case is called the closest pair of subsequence problem. Next in Section 7.3 we propose two exact algorithms. The first algorithm uses \( O(n^2) \) multiplications and \( O(n^3) \) addition operations, and uses \( O(n^2) \) memory. We call this algorithm **Exact-0**. The second algorithm has a run time of \( O(n^3) \), but only uses \( O(1) \) memory. We call the second algorithm **Exact-1**. Another version of the second algorithm takes \( O(n) \) memory but reduces the running time to \( O(n^3 - n^2l) \). Note that the second version only applies to pairwise-sum distances. In the subsequent section we present our approximate algorithm, called **Approx**. We show that the run time of this algorithm is \( O(n^2 + nKl) \) with a high probability. Here \( K \) is a parameter to be chosen in the algorithm. In Section 7.5 we present our experimental results. We have used both biological and time series data, and employed direct Hamming distance and pairwise-sum Euclidean distance, respectively.

### 7.2 Background Knowledge

In this section we provide a summary of some basic techniques that have been used to solve the CLP when \( m = 2 \). An important algorithm in this context is the \( O(n^2) \) algorithm proposed by [114]. The early abandoning technique proposed by [106] is also relevant.

#### 7.2.1 The \( O(n^2) \) Time Algorithm of [114]

For solving the closest pair of \( l \)-mers problem, Pevzner and Sze exploit the overlaps during the process of computing pairwise distances. This eliminates the dependence of the run time on \( l \) [114]. Let \( S = s_1, s_2, \ldots, s_n \) be any given sequence data and let \( l \) be the length of the subsequences we are interested in. The problem of finding the closest pair of subsequences in \( S \) can be decomposed to \( (n - l + 1) \) subproblems. Let these subproblems be referred to as \( P_i \), for \( 1 \leq i \leq (n - l + 1) \). Each \( P_i \) computes the distance between the following pairs of subsequences of length \( l \): \([s_j, s_{j+1}, \ldots, s_{j+l-1}], [s_{i+j-1}, s_{i+j}, \ldots, s_{i+j+l-2}]\), for \( 1 \leq j \leq n - l + 1 \). Each \( P_i \) computes the distance between the following pairs of subsequences of length \( l \): \([s_j, s_{j+1}, \ldots, s_{j+l-1}], [s_{i+j-1}, s_{i+j}, \ldots, s_{i+j+l-2}]\), for \( 1 \leq j \leq n - l + 1 \).
(n − l + 1). Note that in these distance calculations, we can ignore any pair if the elements $s_{n'}$ (for $n' > n$) appear in any of the two subsequences. Let the distance between the pair 
$((s_j, s_{j+1}, \ldots, s_{j+l-1}), (s_{i+j-1}, s_{i+j}, \ldots, s_{i+j+l-2}))$ be $d_j^i$, for $1 \leq j \leq (n − l + 1)$.

[114]'s algorithm makes use of the overlaps in consecutive pairs. We use the Euclidean distance metric as an example here but it is easy to extend our discussion to Hamming distance as well. Since $(d_j^i)^2 = (s_j - s_{i+j})^2 + \ldots + (s_{j+l-1} - s_{i+j+l-1})^2$, the next pair’s squared distance could be expressed as $(d_{j+1}^i)^2 = (s_{j+1} - s_{i+j+1})^2 + \ldots + (s_{j+l} - s_{i+j+l})^2 = (d_j^i)^2 - (s_j - s_{i+j})^2 + (s_{j+l} - s_{i+j+l})^2$.

Clearly, the computation of $(d_1^i)^2$ takes $O(l)$ time. Note that $(d_j^i)^2$ can be obtained from $(d_{j-1}^i)^2$ in an additional $O(1)$ time (for $j > 1$). Thus the problem $\mathcal{P}_i$ can be solved sequentially in a total of $O(n)$ time (for any specific value of $i$, $1 \leq i \leq (n − l + 1)$). Since there are a total of $n$ subproblems, the total running time is $O(n^2)$, which is independent of $l$. In cases of even moderately large dimensions, e.g., $l = 100$, the speedup over brute-force could be as large as 100 times, which is a non-trivial improvement.

7.2.2 Early-abandoning Methods in [106]

In [106], the authors proposed an enhanced version of the brute-force algorithm for finding the time series motifs. The techniques they use improve the run time by a large factor, and one of them is the early-abandoning method. Early-abandoning method takes advantage of the fact that the distance is computed as a summation of $l$ elements, or $\sum_{i=1}^{l} d(A_{j+i}, A_{k+i})$, sequentially. If in the middle of the process when the partial sum exceeds the current best-so-far, which is an upper bound of the closest distance, then we can immediately stop the computation of the distance between the current pair. This method is very useful in practice and will be also employed in our approximate algorithm.

7.3 The Exact Algorithms

When $m = 3$ we can solve the CLP in $O(n^3l)$ time in a straight forward way. The idea is to compute the distance among every triplet of $l$-mers. For each triplet the time spent is $O(l)$ and there are $O(n^3)$ triplets.
7.3.1 Exact-0 Algorithm for Pairwise-sum Distances

We can solve the CLP for \( m = 3 \) in \( O(n^3) \) time using the algorithm of [114] as a subroutine. This algorithm will work as follows: 1) Use the algorithm of [114] to compute pairwise distances in \( O(n^2) \) time. Store all of these distances. Followed by this, compute the distance for each possible triplet of \( l \)-mers. Note that the distance for any triplet can be computed in \( O(1) \) time (since the pairwise distances are available). For instance if \((X,Y,Z)\) is the triplet under concern, its distance is \( d(X,Y) + d(Y,Z) + d(Z,X) \) and the distances \( d(X,Y), d(Y,Z), \) and \( d(Z,X) \) have already been computed and are available. Since there are \( O(n^3) \) triplets, the total addition operations will be \( O(n^3) \). Note that for this algorithm we need \( O(n^2) \) space. We get the following Theorem:

**Theorem 13** We can use Exact-0 algorithm to solve the CLP for \( m = 3 \) using \( O(n^2) \) multiplications and \( O(n^3) \) addition operations, as well as \( O(n^2) \) space. □

7.3.2 Exact-1 Algorithm

If the input size \( n \) is large, the \( O(n^2) \) memory cost may be prohibitive. For example, when \( n = 40 \times 10^3 \), using double precision storage, the algorithm would require roughly 10 GB of memory. This is quite large. Besides, as memory usage increases, the memory accessing cost will become dominant and make the algorithm take longer time to finish.

Motivated by this, we have developed a memory efficient algorithm that solves this problem in \( O(n^3) \) time with only a constant memory requirement. In the case of pairwise-sum distance measurement, \( O(n^2l) \) time could be saved at the cost of \( O(n) \) memory. We thus have two versions: The first version takes \( O(n^3) \) time and uses \( O(1) \) memory; the second version takes \( O(n^3 - n^2l) \) time and employs \( O(n) \) memory. The second version is very useful when \( l \) is not far less than \( n \). For instance, if \( l = 0.3n \), then 30% of the total running time could be reduced.

**Version 1: \( O(1) \) Memory**

The key idea to reduce the memory cost from \( O(n^2) \) to \( O(1) \), is by exploiting the overlaps like in [114]. Rather than using [114]'s algorithm as a subroutine to compute all pairwise
distances in the first step, we split the entire procedure into subproblems \( P_{ik} \) such that each subproblem represents a unique alignment \((i, k)\) and outputs distances of the triplets \((a, a + i, a + i + k), a \in [1, n]\). Clearly, consecutively outputting the distance as \(a\) shifts, would cost \(O(n)\) time for each subproblem, and there’ are a total of \(O(n^2)\) subproblems. So the total running time for this algorithm is \(O(n^3)\). Besides, since only one set of distances (for pairwise-sum distance, three pairwise distances are stored; for direct distance, one triplet distance is stored) needs to be stored in memory, the memory cost becomes \(O(1)\) during the entire process. This can be seen as an enhanced version of \([114]\)’s algorithm. We arrive at the following Theorem:

**Theorem 14** The CLP can be solved in \(O(n^3)\) time using \(O(1)\) space applying Exact-1 algorithm version 1.

**Version 2: \(O(n)\) Memory**

Both Exact-0 and Exact-1 constant memory version are two extreme cases and we are seeking one in the middle by using an affordable amount of memory to reduce the computation time. This could be achieved in the case of pairwise-sum distance metric, because the pairwise distances that have been calculated could be partially stored instead of fully storing (as in Exact-0).

Without out loss of generality, we give an illustration using the example of finding the closest 3 \(l\)-mers from one single sequence under pairwise-sum measurement metric, with a constraint that there are no overlaps for \(l\)-mers in the closest triplet. In the previous \(O(1)\) version, for each alignment \(<i, k>\), the starting cost to compute \(d(0, i), d(i, i+k), d(0, i+k)\) still requires \(O(l)\) time each. And since there are \(O(n^2)\) alignments, the subproblems’ starting costs accumulate to \(O(n^2l)\). After starting, all the remaining distances are calculated in only \(O(1)\) time. As a result, removing the starting cost could save a decent fraction of the total running time. As noticed, the majority of starting cost is in the form of \(d(0, i), i \in [l, n - 2l]\). Thus a simple solution is to store these values in memory to avoid repetition in computing them. This only requires \(O(n)\) storage, and the running time is reduced to \(O(n^3 - n^2l)\) as a consequence.
Algorithm 9 Exact-1 Algorithm with $O(n)$ Memory

**Input:** Sequence $A = s_1, s_2, \ldots, s_n$; subsequence $A_t$ is defined as $A_t = [s_{i}, s_{i+1}, \ldots, s_{i+l-1}]$; $\hat{d}(A_{t_1}, A_{t_2})$ denotes squared Euclidean distance between $A_{t_1}, A_{t_2}$

**Output:** A triplet of subsequences that has the least pairwise-sum Euclidean distance

1: procedure FindExactClosestTuple
2: Set best-so-far $b = \infty$
3: for $i = 0$ to $n - l$ do
4: Compute and store $D_1[i] = \hat{d}(A_0, A_i)$
5: end for
6: for $k = l$ to $n - l$ do
7: Obtain $\hat{d}_1 \leftarrow D_1[k]$
8: for $j = l$ to $n - k$ do
9: Compute $\hat{d}_2 = d(A_k, A_{k+j})$; Obtain $\hat{d}_3 \leftarrow D_1[k+j]$
10: $\text{tmp} = \sqrt{\hat{d}_1} + \sqrt{\hat{d}_2} + \sqrt{\hat{d}_3}$
11: if $\text{tmp} < b$ then
12: update $b \leftarrow \text{tmp}$ and the corresponding indices
13: end if
14: for $i = 0$ to $n - l - k$ do
15: $\hat{d}_1 = \hat{d}_1 - (s_i - s_{i+k})^2 + (s_{i+l} - s_{i+k+l})^2$
16: $\hat{d}_2 = \hat{d}_2 - (s_{i+k} - s_{i+k+j})^2 + (s_{i+k+l} - s_{i+k+j+l})^2$
17: $\hat{d}_3 = \hat{d}_3 - (s_i - s_{i+k+j})^2 + (s_{i+l} - s_{i+k+j+l})^2$
18: $\text{tmp} = \sqrt{\hat{d}_1} + \sqrt{\hat{d}_2} + \sqrt{\hat{d}_3}$
19: if $\text{tmp} < b$ then
20: update $b \leftarrow \text{tmp}$ and the corresponding indices
21: end if
22: end for
23: end for
24: end for
25: return $b$ and the corresponding indices
26: end procedure
Details of this algorithm are given in Algorithm 9. Note that the problem of finding 3 closest \( l \)-mers among 3 separate sequences \((A, B, C)\) can be solved using Algorithm 9 by removing the non-overlapping constraint, and storing 2 sets of distances \( d(A_0, B_i), d(A_i, B_0), i \in [0, n] \). Each set needs \( O(n) \) memory and hence only \( O(n) \) memory is required in total. We obtain the following Theorem:

**Theorem 15** We can solve the CLP in \( O(n^3 - n^2 l) \) time using \( O(n) \) memory applying Exact-1 algorithm version 2. \( \square \)

### 7.4 An Approximate Algorithm: Approx

The brute-force algorithm for \( m = 3 \) takes \( O(n^3 l) \) time, which is very large even for moderately large values of \( n \) and \( l \). The \( O(n^3) \) algorithms take significantly less time by removing the \( l \) factor. Still it takes hours to compute the required triplet from a time series or genome sequence of length 20,000. Therefore it will take days or months to solve the CLP when \( n \) is a million or more. To address this problem we have developed a fast approximate algorithm, which has a running time of \( O(n^2 + nKL) \) with a high probability, where \( K \) is a user defined parameter.

#### 7.4.1 Description

Our approximate algorithm works in two phases. In the first phase, the algorithm computes pairwise distances among all possible \( l \)-mers and keeps \( K \) edges which have the smallest distances. An edge here refers to a pair of \( l \)-mers. A priority queue \( Q \) is used to identify the best \( K \) edges efficiently. To reduce the number of edges that will be inserted into \( Q \), an upper bound on the distance between the closest pair of \( l \)-mers is first obtained using random sampling. During initial random sampling, we pick \( s \) edges randomly. In each pick, each possible edge has an equal probability. We compute the distance of each edge in the sample and identify the edge with the least distance. Let the distance of this edge be \( \delta_s \). We use \( \delta_s \) as the threshold for edges for inserting them into \( Q \).

The \( K \) edges that are in \( Q \), after processing all possible edges, will be used in the second phase. We form candidate triplets as follows: For each of the edges in \( Q \) form triplets with
every \(l\)-mer in the input sequence \(A\). From out of all of these candidate triplets identify and output the one with the least distance. Algorithm 10 shows the details of the algorithm for single sequence version (\(l\)-mers come from single sequence). The multi sequences CLP could be easily solved by removing the overlapping constraint.

**Algorithm 10**  Approx algorithm

**Input:** A sequence \(A = s_1, s_2, \ldots, s_n\), integer \(l\), priority queue \(Q\) of size \(K\)

**Output:** A triplet of \(l\)-mers whose members have the least distance.

1: procedure FINDAPPROXCLOSESTTUPLE  
2:   Choose randomly \(s\) pairs of \(l\)-mers (edges) from \(A\)  
3:   Compute the distance for each pair in the sample, and identify the closest pair in the sample; let this closest distance be \(\delta_s\)  
4:   for All pairs in \(A\) do  
5:     Compute the distance between each pair of \(l\)-mers  
6:     if Any pair’s distance < \(\delta_s\) then  
7:       Push into \(Q\)  
8:     end if  
9:   end for  
10:  Set best triplet-distance as \(b = \infty\)  
11:  for each \(l\)-mer \(u\) do  
12:    for each pair \((v, w)\) in priority queue do  
13:      Compute distances of \((u, v), (u, w)\) pairs  
14:      Set triplet distance \(TD = \text{distance}(u, v) + \text{distance}(v, w) + \text{distance}(u, w)\)  
15:      if \(TD < b\) then  
16:        \(b = TD\), update corresponding indices to \((u, v, w)\)  
17:      end if  
18:    end for  
19:  end for  
20: return \(b\) with associated indices  
21: end procedure

7.4.2 Analysis

We choose a random sample of \(s\) pairs of \(l\)-mers from the input sequence \(A\). The algorithm takes \(O(sl)\) time to calculate pairwise distances of these pairs. In our implementation we choose \(s = \Theta(n)\). We get our threshold value \(\delta_s\) by finding minimum of these distance values. Then we compute the distance between every pair of \(l\)-mers of the input sequence \(A\). This can be done in \(O(n^2)\) time. From out of these, we identify the \(K\) least distances. Identification of these \(K\) pairs is done using a priority queue \(Q\). We insert any pair into \(Q\) only if its distance is less than \(\delta_s\). \(Q\) will have at most \(K\) pairs at any time. For each pair
that enters $Q$ another pair may have to be deleted. An important question is how many pairs will enter $Q$ in the worst case. We claim that the number of pairs that will enter $Q$ is $O \left( \frac{N}{s} \log n \right)$ with a high probability, where $N$ is the number of possible pairs of $l$-mers. (Note that $N = O \left( \binom{n}{2} \right)$). This can be proven as follows.

By high probability we mean a probability of $\geq (1 - n^{-\alpha})$, where $\alpha$ is a probability parameter typically assumed to be $\geq 1$ (see e.g., [51]). Let $G$ stand for the set of pairs of $l$-mers of $A$ with the least distances, where $|G| = q$. I.e., $q$ pairs with the least distances are in $G$. (The value of $q$ will be fixed soon). Let the pairs in $G$ be $p_1, p_2, \ldots, p_q$. Probability that $p_1$ is in the random sample is $\frac{s}{N}$. Probability that $p_1$ is not in the sample is $1 - \frac{s}{N}$. This means that the probability that none of $G$ is in the sample is $\left(1 - \frac{s}{N}\right)^q$. This probability is $\leq \exp \left( -\frac{sq}{N} \right)$ using the fact that $(1 - x)^{1/x} \leq \frac{1}{e}$ when $0 < x < 1$. This probability will be $\leq N^{-\alpha}$ when $q \geq \alpha \frac{N}{s} \log e N$. This in turn means that at most $\alpha \frac{N}{s} \log e N$ pairs will ever enter $Q$ with a probability of $\geq 1 - N^{-\alpha} \geq (1 - n^{-\alpha})$.

The above analysis implies that the total time spent in maintaining $Q$ is $O \left( \frac{N}{s} \log^2 n \right) = O \left( \frac{n^2}{s} \log^2 n \right)$ with a high probability. Also, steps 1 and 2 in Algorithm 10 take a total of $O(sl)$ time. Step 4 can be done in $O(n^2)$ time. As we have shown before, steps 5 and 6 take a total of $O \left( \frac{n^2}{s} \log^2 n \right)$ time with a high probability. The for loop of Step 8 takes $O(nKl)$ time. Therefore, the total run time of Algorithm 10 is $O \left( sl + n^2 + \frac{n^2}{s} \log^2 n + nKl \right)$ with a high probability. If $s = \Theta(n)$, this run time becomes $O(n^2 + nKl)$. Thus we arrive at the following Theorem.

**Theorem 16** The run time of Approx is $O(n^2 + nKl)$ with a high probability. □

### 7.5 Experimental Evaluation

In this section, we evaluate our proposed algorithms for run time and/or accuracy using two existing datasets. Each dataset is tested using one measurement metric (direct Hamming distance and pairwise-sum Euclidean distance). The test platform we are using is equipped with Intel Xeon CPU @ 2.67GHz.
7.5.1 Genome Dataset

We have performed intensive experiments on human genome data set [117]. We chose 21 chromosomes and grouped them into 7 files each having 3 chromosome sequences. We have run Exact-1 and Approx algorithms in order to identify the closest \( l \)-mers among three sequences in each set, and there are 7 sets of genome sequences. For Approx algorithm, we set the priority queue size as \( K = n \) to store the top \( n \) pairs of candidates as a default. We have used different values for \( n \) ranging from 4,000 to 60,000. The first \( n \) elements of the 7 sets of genome sequences are used to form the input sequences. Direct Hamming distance is used as the distance metric. For a fixed \( n \) and \( l \), we call such a combination a test group, and the running time is calculated as an average over the 7 sets of genome sequences for this group. We report the accuracy of Approx using the number of hits (# Hits), which measures how many times out of the 7 runs (7 sets), the closest \( l \)-mers are identified.

At first we compare our proposed algorithms with the \( (O(n^3l)) \) time brute-force algorithm. The result is given in Table 7.1. From the table we see that the brute-force algorithm performs worse even when \( n = 4,000 \). Thus in later experiments we will not include the brute-force algorithm and only compare our proposed approaches.

<table>
<thead>
<tr>
<th>( n = 4,000 )</th>
<th>( l=100 )</th>
<th>( l=200 )</th>
<th>( l=300 )</th>
<th>( l=400 )</th>
<th>( l=500 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Approx</td>
<td>4.2</td>
<td>5.6</td>
<td>7.6</td>
<td>9.4</td>
<td>7.7</td>
</tr>
<tr>
<td>Exact</td>
<td>207.0</td>
<td>215.1</td>
<td>212.3</td>
<td>203.0</td>
<td>199.0</td>
</tr>
<tr>
<td>Brute-force</td>
<td>8,353.0</td>
<td>19,613.0</td>
<td>28,744.6</td>
<td>37,178.9</td>
<td>44,607.9</td>
</tr>
</tbody>
</table>

The next experiment provides a full evaluation when \( n \) ranges from 4,000 to 10,000, with \( l = 100 \) to 500. The running time of Exact-1 and Approx are provided in Semilog-Y plot for a better illustration in Figure 7.1. From the plot, we clearly see that the Approx algorithm outperforms Exact-1 by more than one order of magnitude for all the 5 different \( l \) values. Also, as the dataset size \( n \) increases, the running time difference increases.

Next we want to investigate how the performance changes as \( l \) varies, for a fixed \( n \). We choose \( n = 6,000 \) and change \( l \) from 100 to 500. The running times are shown in Figure 7.2. The upper plot shows the running time for the Approx algorithm and the lower
one represents Exact-1 algorithm. The observation here is that since Approx algorithm’s running time depends on \( l \), as \( l \) increases, the running time of Approx slightly increases. On the contrary, Exact-1 algorithm is dimension free. When \( l \) increases, the actual number of triplets \((n - l)^3\) decreases, making the algorithm’s running time slightly decrease. But still, Approx is much faster than Exact-1 even when \( l = 500 \).

The speedups for all \( n \) and \( l \) combinations are given in Figure 7.3. Speed up is defined as the running time of Exact-1 divided by the running time of Approx algorithm. As can be seen, speedup decreases as \( l \) grows, and increases as \( n \) grows, which matches our expectation and theoretical analysis.

In the last part of time comparison, we are testing our algorithm on large datasets with \( n = 10,000,\ 20,000,\ 40,000,\ 60,000 \) and \( l = 200,\ 400,\ 600,\ 800,\ 1,000 \). Using this setting, Approx could output the results within two hours, while Exact-1 exceeds our
experimental limit of 15 hours for \( n = 40,000 \) and above. The details of running time for large datasets are given in Table 7.2.

Table 7.2: Running times for large datasets

<table>
<thead>
<tr>
<th>n</th>
<th>l=200</th>
<th>l=400</th>
<th>l=600</th>
<th>l=800</th>
<th>l=1,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>10,000</td>
<td>32.2</td>
<td>61.2</td>
<td>92.9</td>
<td>86.0</td>
<td>112.3</td>
</tr>
<tr>
<td>20,000</td>
<td>125.4</td>
<td>207.3</td>
<td>397.2</td>
<td>518.5</td>
<td>609.8</td>
</tr>
<tr>
<td>40,000</td>
<td>617.6</td>
<td>903.5</td>
<td>1,387.7</td>
<td>1,318.4</td>
<td>1,388.6</td>
</tr>
<tr>
<td>60,000</td>
<td>853.2</td>
<td>1,949.6</td>
<td>3,009.3</td>
<td>4,467.5</td>
<td>4,984.7</td>
</tr>
</tbody>
</table>

Since time and accuracy are trade-offs for an approximate algorithm, we also investigate the output accuracy of Approx with a default setting of \( K = n \). The result is given in Table 7.3. From the table we clearly see that the accuracy of Approx algorithm is very high. Especially for larger sequence lengths such as \( n = 10,000 \), most of the times Approx could achieve 100% Hits for different \( l \)-mer lengths.

As \( K \) value would affect the time and accuracy of the Approx algorithm, we conduct another experiment to test how the performance changes as \( K \) varies. We vary \( K \) from \( 0.001n \) to \( 10n \), and measure both the accuracy and running time when \( n = 10,000 \) with different \( l \) values. The Hit rate defined by \# of Hits divided by the number of runs, 7 in this case, against \( K \) value is provided in the first plot of Figure 7.4. Figure 7.5's left plot illustrates the running time as \( K \) changes. We can easily see that as \( K \) increases, both the
hit rate and run time increase, for all different $l$ values. This is because small $K$ means a small priority queue and hence the time to maintain $Q$ decreases. As a result, only storing a small number of candidate pairs could reduce the chance of identifying the true closest triplet. From the figures we observe that if $K = 0.1n$, the total run time could be reduced by a large factor, while still maintaining a good accuracy.

Figure 7.4: Hit rate for different $K$ values, $n = 10,000$

Figure 7.5: Running time for different $K$ values, $n = 10,000$

### 7.5.2 Human Activity Dataset

In this experiment, we evaluate our algorithms under the pairwise-sum distance measurement using Euclidean distance, i.e. $d(A_i, A_j, A_k) = d(A_i, A_j) + d(A_j, A_k) + d(A_i, A_k)$. 

<table>
<thead>
<tr>
<th>$n$</th>
<th>$l=100$</th>
<th>$l=200$</th>
<th>$l=300$</th>
<th>$l=400$</th>
<th>$l=500$</th>
</tr>
</thead>
<tbody>
<tr>
<td>4000</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>6000</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>8000</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>10000</td>
<td>7</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$n$</th>
<th>$l=200$</th>
<th>$l=400$</th>
<th>$l=600$</th>
<th>$l=800$</th>
<th>$l=1,000$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>7</td>
<td>6</td>
<td>7</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>20000</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>
The goal is to identify 3 \( l \)-mers from one single sequence \( A \), such that their pairwise-sum distance is minimum, under the constraint that they do not overlap with each other.

The dataset we use is from UCI Machine Learning Repository \[3\]. For a fair comparison, we have randomly selected one dataset which happens to be the Heterogeneity Activity Recognition Data Set \[130\]. This contains around \( 1 \times 10^7 \) real numbers. This dataset includes cellphone accelerometer and gyroscope recorded data for human activity. There are 6 sensor coordinates in total and each forms a long sequence of numbers.

To perform evaluations, we downsampled the dataset with an interval of 10 for each sequence, and then applied a shifting of 0 and 5 to obtain a total of 12 downsampled sequences. We have performed evaluations on different \( n \) and \( d \) values. The first \( n \) elements in each sequence have been pulled out to form a group of data sequences. The evaluation is based on average performance across 12 data sequences in each group, and accordingly the accuracy is reported as the number of Hits out of 12. Three algorithms are evaluated on this dataset, which are Exact-0, Exact-1 and Approx. For Approx, we still set \( K = n \) as a default.

Similar to previous experiment, we first inspect the running time of all the three algorithms under different settings of \( n \) and \( l \) combinations. As shown in Figure 7.6, three clusters of curves represent three algorithms, respectively. Among them, Approx still runs the fastest and Exact-1 is the most time consuming. Between Approx and Exact-1, Exact-0 gives a moderate running time at the cost of \( O(n^2) \) memory. For datasets up to \( n = 10,000 \), around 600MB memory is occupied by Exact-0. However, since Exact-0 is five times faster than Exact-1, it is very competitive on small to moderate datasets.

Also, the speedup of Approx against both Exact-0 and Exact-1 are given in Figure 7.7. In the figure, "Speedup 1" and "Speedup 0" represent Approx’s speedup over Exact-1 and Exact-0, respectively. The observation is that for small dimensions \( (l) \), and for large sequence lengths \( (n) \), much higher speedup could be achieved using Approx algorithm. For large \( l \) values, the running time of Approx could be almost the same as Exact-0, such that Exact-0 could be preferred in these cases as it is an exact algorithm that always outputs the correct answer. However, as stated above, if \( n \) is large, Exact-0 may no longer be applicable due to the huge memory cost.
In the next test we demonstrate how these three algorithms’ running time varies as $l$ changes. Using the same setting as in the previous experiment, we pick $n = 6,000$ and change $l$ from 100 to 500. Figure 7.6 plots three curves representing three the algorithms, respectively. As expected, for both exact algorithms Exact-0 and Exact-1, the running time decreases as $l$ increases, because the actual number of triplets $(n - l)^3$ decreases. For Approx, the running time increases due to its dependence on $l$.

The next experiment is performed on larger $n$ and $l$ values. In particular, $n = 10,000$, $20,000$, $d$ ranges from 200 to 2,000. From Table 7.4 we see that Approx is more than 10 times faster than Exact-0 and 100 times faster than Exact-1 on small $l$ values. For larger $l$ values, Approx is still more than twice faster than Exact-0 and 20 times faster than Exact-1.

Besides the running time performance, we also provide accuracy (# Hits) for all the
above tests in Table 7.5. Due to the fact that $K$ is set depending on $n$ ($K = n$ in the above tests), the accuracy of Approx is consistent for all the different $n$ values, making this a fair experiment for running time comparison. (Approx’s running time also depends on $K$). As shown in Table 7.5, using a default $K = n$, we are able to obtain a very high accuracy. We get full Hits (12 Hits out of 12 runs) for $l = 100$ and many other settings. One interesting fact is that as $l$ increases, # Hits decreases slightly. The reason behind this is that since $l$ increases, there are more elements contributing to the total distance computation (note that the distance is a summation of $l$ elements), such that the randomness increases a lot. So there are higher chances that the 3 pairs $(a,b), (b,c), (a,c)$ within the closest triplet $(a,b,c)$ might not exist in the top $K$ closest pairs, but still making the triplet closest.

At the end, we are inspecting how Approx performs if $K$ changes. The Hit rate and the running time are provided in the second plots of Figure 7.4 and Figure 7.5. The figure shows similar trends as in the Genome dataset.
Table 7.5: #Hits out of 12 for Approx, $K = n$

<table>
<thead>
<tr>
<th>n</th>
<th>l=100</th>
<th>l=200</th>
<th>l=300</th>
<th>l=400</th>
<th>l=500</th>
</tr>
</thead>
<tbody>
<tr>
<td>4000</td>
<td>12</td>
<td>12</td>
<td>7</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>6000</td>
<td>12</td>
<td>11</td>
<td>8</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>8000</td>
<td>12</td>
<td>12</td>
<td>10</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>10000</td>
<td>12</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>n</th>
<th>l=200</th>
<th>l=400</th>
<th>l=600</th>
<th>l=800</th>
<th>l=1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>12</td>
<td>10</td>
<td>9</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>20000</td>
<td>12</td>
<td>12</td>
<td>8</td>
<td>7</td>
<td>9</td>
</tr>
</tbody>
</table>

### 7.5.3 Summary of Experimental Evaluation

In this section we have performed comprehensive evaluations on Genome dataset and Activity dataset. The measurement metrics we used are direct Hamming distance and pairwise-sum Euclidean distance. For Genome dataset, two algorithms Exact-1 and Approx are tested; for Activity dataset, three algorithms Exact-0, Exact1 and Approx are compared. The experiments are carried on different $n$ and $l$ values. Most of the experiments for Approx are using $K = n$ as a default. We have the following observations:

- The performances are consistent using both measurement metrics on two different datasets, showing our proposed algorithms are robust.

- Exact-0 algorithm runs faster than Exact-1, at a cost of $O(n^2)$ memory. On small datasets, it is very competitive.

- Exact-1 is performing much better than brute-force, making it a good candidate for exact algorithm that always outputs correct answer.

- Exact algorithms’ running times decrease as $l$ increases, while Approx’s running time increases.

- Approx runs much faster than both the exact algorithms, while maintaining a very high accuracy.

- Approx’s running time almost increases quadratic or even less on $n$, while exact algorithms grow cubic on $n$. 

121
• As $l$ increases, the accuracy for Approx slightly decreases.

• $K$ can be set to $0.1n$ to achieve even better speedups while maintaining a similar accuracy.
Part IV

Conclusions and Future Directions
This thesis has focused on three big data problems: $k$-mer counting problem, record linkage problem, and some problems with algorithmic challenges.

We have investigated the $k$-mer counting problem in Part I. Implementation of our proposed algorithm has produced expected results in around 30% less time than the previous best-known implementation. Our algorithm consumes more memory compared to some other efficient algorithms. Therefore further improvement is possible by transforming this in-memory algorithm into an external memory algorithm. We will work on this problem in future.

In Part II, we have studied the record linkage problems in detail. Our algorithms have employed single linkage and complete linkage based hierarchical clustering. For distance measurement, we have used edit distance, reversal edit distance, and truncation edit distance. Our algorithms can be extended easily to support other types of distance measurement methods. This is worth investigating in future.

Our discussion in Part III has covered three problems with algorithmic challenges having wide applications. We can accommodate billions of data points in our minimum spanning tree algorithm by extending our implementation to distributed computing. In the closest $l$-mers problem, we have mainly studied the case of three $l$-mers. We can broaden its impact by covering groups of larger number of $l$-mers. We will work on this in future. Current higher order spectra algorithms have only considered univariate nonlinear time series data. They are generic enough to deal with multivariate time series data. This problem is also of interest to us in future.
Bibliography


[17] Peter Christen. Febrl: a freely available record linkage system with a graphical user interface. In *Proceedings of the second Australasian workshop on Health data and


[104] Jason R Miller, Arthur L Delcher, Sergey Koren, Eli Venter, Brian P Walenz, Anushka Brownley, Justin Johnson, Kelvin Li, Clark Mobarry, and Granger Sut-


