View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by University of Central Florida (UCF): STARS (Showcase of Text, Archives, Research &...

University of Central Florida
STARS

Electronic Theses and Dissertations, 2004-2019

2019

Efficient String Graph Construction Algorithm

S.M. Iqbal Morshed
University of Central Florida

Part of the Computer Sciences Commons
Find similar works at: https://stars.library.ucf.edu/etd
University of Central Florida Libraries http://library.ucf.edu

This Masters Thesis (Open Access) is brought to you for free and open access by STARS. It has been accepted for inclusion in Electronic Theses and Dissertations, 2004-2019 by an authorized administrator of STARS. For more information, please contact STARS@ucf.edu.

STARS Citation

Morshed, S.M. Iqbal, "Efficient String Graph Construction Algorithm" (2019). *Electronic Theses and Dissertations, 2004-2019*. 6303.
https://stars.library.ucf.edu/etd/6303

University of Central Florida
STARS
Showcase of Text, Archives, Research & Scholarship

EFFICIENT STRING GRAPH CONSTRUCTION ALGORITHM

by

S.M. IQBAL MORSHED
B.S. University of Dhaka, 2013
M.S. University of Dhaka, 2016

A thesis submitted in partial fulfilment of the requirements
for the degree of Master of Science
in the Department of Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Spring Term
2019

Major Professor: Shibu Yooseph

© 2019 S.M. Iqbal Morshed

ABSTRACT

In the field of genome assembly research where assemblers are dominated by de Bruijn graph-based approaches, string graph-based assembly approach is getting more attention because of its ability to losslessly retain information from sequence data. Despite the advantages provided by a string graph in repeat detection and in maintaining read coherence, the high computational cost for constructing a string graph hinders its usability for genome assembly. Even though different algorithms have been proposed over the last decade for string graph construction, efficiency is still a challenge due to the demand for processing a large amount of sequence data generated by NGS technologies. Therefore, in this thesis, we provide a novel, linear time and alphabet-size-independent algorithm SOF which uses the property of irreducible edges and transitive edges to efficiently construct string graph from an overlap graph. Experimental results show that SOF is at least 2.3 times faster than the string graph construction algorithm provided in SGA, one of the most popular string graph-based assembler, while maintaining almost the same memory footprint as SGA. Moreover, the availability of SOF as a subprogram in the SGA assembly pipeline will give user facilities to access the preprocessing and postprocessing steps for genome assembly provided in SGA.

I dedicate this work to Ammu, Abbu, Naoshin and my wife Mahfuza. Without their continuous support this work would not be possible.

ACKNOWLEDGMENTS

I am deeply indebted to my research supervisor Dr. Shibu Yooseph, Professor, Department of Computer Science, University of Central Florida. His scholarly guidance, important suggestions, constant supervision, valuable criticism from the beginning to the end of the research work has made the thesis work possible. I am also grateful to the thesis committee members – Dr. Shaojie Zhang and Dr. Sharma Thankachan – for their valuable suggestions to improve this thesis.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: INTRODUCTION

Obtaining whole genome sequence exactly as it is present in the chromosomes of living cells has always been the area of interest for biologists. That motivated the rise of different sequencing technologies in the last four decades. These sequencing machines can generate fragments of a sequence where the length of the sequence varies depending on the technology. However, sequencing machines do not have the capability to generate the entire nucleotide sequence as it is present in the chromosome. That opens for computer scientists an opportunity to be engaged in Sequence Assembly Problem. Sequence Assembly Problem is formulated as given a set of substrings, usually called reads which are generated from string *S*, the problem is to construct the superstring *S*.

Sequence Assembly Problem is usually divided into two subcategories: *de novo* assembly and *reference-based* assembly. In the reference-based assembly, a reference sequence is used as a guide to the assembly process, but in the de novo assembly, assembly is performed without taking any information from a reference sequence. In reference-based assembly, usually, the goal is to map or align reads to the most probable locus in the genome sequence from which it originated. On the other hand, de novo assembly is used to construct a chain of reads which represents the original sequence with high accuracy. Despite the difference in problem formulation and goals, sometimes both of these approaches are used in combination.

To perform de novo assembly, almost every methods or approach utilizes some kind of graph data structure to represent the relationship between reads. Two of the most popular approaches are *de Bruijn graphs* and *Overlap graphs* (and its reduced form *string graph*). The basic idea of a de Bruijn graph consists of creating a graph where a set of distinct k-mers (i.e. string of length k) generated from the sequence reads represents vertices and there is an edge between vertices if there is a $(k-1)$-mer overlap between k-mers.

String graph-based genome assembly is another popular alternative to de Bruijn graph based approach. Myers first provided the definition of string graph [Mye05] which is based on the concept of overlap graph. In an overlap graph, read sequence represents a vertex. There is an edge between two vertices if there is an overlap between two corresponding reads. But this definition of overlap graph creates many redundant vertices and edges which makes it difficult to be stored in a space-efficient way. Myers defined string graph in such a way that a string graph neither contains redundant vertices nor redundant edges. In this definition, a vertex is considered redundant if the corresponding read is identical to or a substring of another read. Also, an edge is considered redundant if there is a path which represents an assembly same as the assembly constructed from that edge. This type of edge is called transitive edge. If an edge is not transitive, it is called irreducible.

Since a string graph does not split a read sequence into k-mers, it provides advantages over de Bruijn graph during the earlier processing stages by distinguishing repeats longer than k situated within a read sequence. Besides, a path in the string graph represents a valid assembly (read coherence property) which is not always true for a de-Bruijn graph. Also, a string graph retains all the information that can be inferred from the read data. Therefore, depending on the nature of the problem, a string graph can sometimes be more useful than a de Bruijn graph.

However, constructing a string graph is computationally challenging as it requires identifying overlaps between reads and removal of transitive edges. Many different algorithms have been proposed to perform assembly through the construction of a string graph. Edena [HFF$^+$08] is one of the earliest string graph-based assembler that computed overlaps between reads using suffix array and performed transitive edge removal using the approach mentioned in [Mye05]. [BBC14] developed string graph-based assembler using hashing and bloom-filter.

Yet, the massive amount of sequencing data produced by Next Generation Sequencing (NGS) technologies poses a constant demand for the time and space efficient algorithm to construct string graph. To tackle this challenge, different algorithms have been proposed based on the self-indexing data structures like FM index [FM05]. Through the use of self-indexing data-structures, overlap identification from the pool of a large amount of sequence data has become efficient. Even though self-indexing data structures provide efficiency in overlap detection, transitive edge removal from the overlap graph still remains a challenge. Most popular string graph-based assembler SGA [SD12] uses BWT of the reverse reads to identify transitive edges which is a comparatively slower process than the overlap identification. Besides, storing two BWTs in the memory causes high memory footprint.

Recently, [BVP+16] have developed a space-efficient string graph construction algorithm LSG. LSG uses external memory algorithm for building a string graph which consequently reduces the main memory consumption. However, its use of external memory makes itt very slow compared to SGA. To overcome the time efficiency issue, the same lab that developed LSG, developed a time efficient string graph construction algorithm FSG [BDVP+17]. But this algorithm takes 2-3 times more space than SGA.

It is clear that to make string graph practical for usage, an efficient algorithm is needed that can make a good balance between time and space. In this thesis, we propose a novel string graph construction algorithm SOF which is at least 2.3 times faster than SGA while it takes almost the same amount of memory as SGA. SOF achieves time efficiency by using properties of transitive edges. It achieves space efficiency by storing overlaps in external memory.

NGS technologies not only influenced the efficient algorithm construction but also influenced the growth of research in metagenomics as these technologies provide unprecedented access to the bacterial data analysis in a culture-independent way. It is an important task in metagenomics to identify functional and metabolic potential present in the sequenced microbial community. This can be done through homology detection where a reference sequence is used to identify significantly similar reads from the pool of sequence data. Recently [ZYY17] has shown that string graph can also be useful in peptide sequence assembly and homolog detection from metagenomic data.

Our proposed algorithm SOF has a property that its transitive edge removal process is alphabet size independent. This property is particularly useful where alphabet size of the sequence data is large. Hence we predict that our proposed algorithm has the potential to be the best method in metagenomic peptide assembly.

Our implementation is available in the open source code repository [Mor]. To provide better user experience, instead of creating a standalone software tool, we have developed our program as a plug-in of SGA which provides a user access to all the preprocessing and postprocessing tools available in SGA.

1.1 Thesis Organization

Rest of the thesis is organized in the following way:

Chapter 2 provides the background of the research. In this chapter, we discuss earlier research on string graph construction algorithm. At the end of this chapter, we discuss our contribution and the novelty of our work.

Chapter 3 discusses in detail about our proposed string graph construction methodology. In the beginning of this chapter, we discuss relevant definitions, data-structures and proofs which supports the explanation of the proposed algorithm. Finally, we discuss about the complexity and implementation details of the algorithm.

In chapter 4, we discuss about the experimental procedure we followed to measure the performance of SOF. We also discuss about the results we have obtained by conducting the experiment. We conclude this chapter by showing validation of our result.

Finally, we conclude our discussion by providing a summary of this thesis. We also discuss about possible future work plan in this chapter.

CHAPTER 2: LITERATURE REVIEW

Genome sequencing is the process of determining the sequence of bases in a genome. Current sequencing technologies do not allow us to sequence a genome in its entirety. Instead, a sequencing machine can only generate sub-sequences of a genome. Each of the subsequence is called *read*. Therefore, a set of reads is the outcome of a sequencing process performed on the genome of an individual. Two reads can have overlap in sequence and this overlap is utilized to generate contiguous sequence through the process known as genome assembly. Since the assembly process is performed on reads sequences, the quality of a genome assembly is strongly related to the quality of reads generated by the sequencing machine.

In the last 40 years, different sequencing technologies have emerged with different characteristics. First one is the Sanger sequencing method developed in 1977 by Frederick Sanger [SNC06] which dominated the sequencing technology for 30 years. This technology has the ability to produce reads of length up to 1000 base pairs (bp) with lower error rate (10^{-4} to 10^{-5}) [KK10]. However, being slow and costly precluded this technology to be widespread. In 2005, new sequencing technologies have emerged which are called Next-Generation Sequencing (NGS) Technologies [LLL+14]. NGS revolutionized genome sequencing by reducing the cost. Reads produced by these technologies are in the range of 35 to 400 base pairs which are shorter than the reads produced in Sanger Sequencing. Short read sequences generated in NGS technology poses a challenge in repeat detection when repeat sequence length is longer than read length. Despite this shortcoming, being faster and cheaper than Sanger Sequencing [GMM16] made different NGS machines (Illumina, Ion Torrent, SOLiD, 454) popular over time [DIS+16]. Since 2011 new sequencing technologies referred to as Third-Generation Sequencing have been proposed to overcome the limitation caused by short read in NGS. Even though this new sequencing technology produces long read sequences (1000 bp to 200000 bp), it has a much higher error rate (10% - 15%) than other existing technologies [GMM16]. Therefore, Third generation technologies are now used in combination with NGS technologies.

Despite the difference in technologies, every sequencing machine has one common characteristic and that is to generate a set of subsequences (i.e. reads). The next challenge is to assemble these subsequences to generate a sequence which is the closest approximation of the original sequence. This problem is known as sequence assembly problem in bioinformatics. This problem is generally classified into two major categories – *de novo* assembly and *reference-based* assembly. In a de novo assembly, the assembly operation is performed using only the set of read sequences. On the other hand, in a reference-based assembly, a reference sequence is used as a guide in the assembly process. In reference-based assembly, the goal is to map each read to the most probable locus in the reference genome whereas in de novo assembly, the goal is to build contiguous sequences by creating a chain of reads. Even though in this thesis, we primarily focus on strategies related to de novo assembly, our proposed method can also be applicable in reference-based assembly.

There are two prominent graph data structures used in de novo assembly – de Bruijn graph and string graph. Pevzner [PTW02] defines the de Bruijn graph used for DNA assembly problem in this way: Given a set of reads $S = s_1, s_2, ..., s_n$, the de Bruijn graph $G(S_k)$ with vertex set S_{k-1} (the set of all $(k-1)$-mers from S) is defined as follows. A $(k-1)$-mer $v \in S_{k-1}$ is joined by a directed edge with an $(k-1)$-mer $w \in S_{k-1}$, if S_k contains an k-mer for which the first $k-1$ nucleotides coincides with v and the last $k-1$ nucleotides coincides with w. By allowing the multiplicity of edges, genome assembly problem can be represented as a Euler path problem where an assembly is obtained by visiting every path exactly once. Construction of de Bruijn graph requires performing exact matches between k-mers which can be done in linear time using a hash table. This simplicity in construction process made de Bruijn graph the dominant method of de novo assembly.

However, de Bruijn graph has some limitations. Since a de Bruijn graph breaks reads into k-mers, it may fail to retain all information that is obtainable from a read set. Therefore, a tour in the de Bruijn graph may not represent a valid sequence. So, an effort was made to develop a string-labeled graph such that a tour in the graph would represent a valid sequence and the graph has few extraneous edges and alternate tours as possible. This effort led to the first formalization of the concept of string graph by Eugene Myers [Mye05]. Formal definition of the string graph is discussed in section 3.1.2.

Because of the properties of string graph, it provides some advantage over de Bruijn graph. One of the properties of a string graph is *read coherence* which means that any path in the graph represents a valid assembly of the reads. Since this property does not hold true in de Bruijn graph, in the de Bruijn graph-based assembly, a separate procedure is required to restore read coherence. Also, a string graph provides an advantage in disambiguation of repeats. Since a string graph does not break reads into k-mers, it can immediately distinguish repeat larger than k but smaller than read length, whereas de Bruijn graph $G(S_k)$ can distinguish repeat larger than k only in its later stages.

On the other hand, string graph is computationally expensive than the de Bruijn graph. De Bruijn graphs are easier to store and compute because nodes and edges have a consistent structure. But in a string graph, overlap identification requires much computation. Besides, by definition, a string graph does not contain any transitive edges. Therefore, a string graph construction algorithm needs to provide an efficient transitive edge reduction process.

In other words, there are two main computationally intensive tasks in string graph construction: 1. construction of overlap graph by identifying all overlaps between sequence reads. 2. Removal of transitive edges from the overlap graph. In [Mye05], Myers also provided a linear time transitive reduction algorithm. In this transitive reduction algorithm, edges of a node in overlap graph are traversed in ascending order of edge label length. For a node x, if an adjacent node u can be traversed through another adjacent node w where the label of (x, w) is shorter than (x, u), then (x, u) is considered transitive. In this way, all transitive edges are identified and consequently removed. Even though it is a linear time algorithm, it requires whole overlap graph to be stored in memory which makes this algorithm memory intensive. Edena [HFF^{+}08] is one of the earliest string graph-based assembler which computes overlaps between reads using suffix array and performs transitive edge removal using the approach mentioned in [Mye05].

With the emergence of NGS technologies, a massive amount of sequence data with high coverage rate has been produced. Therefore, identifying overlaps between reads from the set of millions of reads was a major challenge for a string graph algorithm to be considered useful in genome assembly. String graph construction algorithm developed by [SD10] which was later included as the part of string graph-based assembly pipeline SGA [SD12] has made a major breakthrough in this regard. By using self-indexing data structure FM-index [FM05], SGA has made overlap detection phase of the string graph construction faster. In the overlap identification step, SGA performs backward search on BWT [BW94] for each read r. For each suffix s of the read r, it identifies a interval Q in BWT such that this interval represents a read set r_s where all of reads have prefix s. In this way, it identifies all overlaps for all suffixes of r. In the next step, SGA performs transitive removal by using BWT of the reverse reads. SGA performs a rightward extension of each overlap interval by adding all different characters from the alphabet set and checking whether this addition leads to the terminal of some reads represented in the Q. If a read is found to be terminated, this read is considered to form an irreducible edge with r. Rest of reads are considered transitive edge forming reads and consequently discarded. Currently, SGA is one of the most popular string graph-based assembler [SN18] thanks to its assembly pipeline and stable implementation. Readjoiner [GK12], another string graph-based assembler, showed improvement over SGA. Readjoiner identifies overlaps by computing a proper subset of suffix-prefix matches and removes transitive edges by performing a traversal algorithm on the graph.

Alongside the self-indexing data structure, hashing and a probabilistic membership data structure called bloom-filter [Blo02] is shown to be useful in de novo assembly algorithms. [CR12] first showed the use of hashing and bloom-filter for the efficient representation of a de Bruijn graph. Their results were further improved by the use of cascading Bloom filter [SSK13]. [BBC14] first presented that the hashing and bloom-filter can also be effective for the construction of a string graph. Their method detects overlaps by using incremental hash function Karp-Rabin [KR10] for all relevant prefixes of a particular read. To ensure memory efficiency, it uses the bloom filter to store hash values for different prefix length of each read. To perform transitive edge reduction, it follows a strategy of the right extension similar to SGA. Since Bloom Filter is a probabilistic data structure, set of identified irreducible edges may contain false positive and false negative reads which are later removed by performing a verification step. Performance of this Hashing and Bloom-Filter based method is comparable with the older version of SGA. But it did not show significant improvement in comparison with the Readjoiner and the recent implementation of SGA.

Recently, external memory based approach LSG [BVP$^+$16] has been proposed which uses BWT indexes to construct string graph. Through the use of external memory, this approach is shown to be very memory efficient. In comparison to the SGA, this approach only takes 2% of main memory used by SGA. Indeed, use of the external memory made the program slower than the SGA.

To solve the time efficiency issue, the same lab that developed LSG developed another algorithm FSG [BDVP$^+$17] which is 2-3 times faster than SGA. To collect irreducible edges, FSG performs right extension using BWT of the reverse reads, which is the same approach as SGA. However, FSG achieves time efficiency by extending only the unique overlaps. Another reason for FSG's performance improvement is the indexing of both forward reads and reverse complement of reads. This obviates the necessity to identify overlaps of the reverse complement of a read separately. Since FSG performs BWT indexing of reads and their reverse complement, BWT now takes twice as much memory as SGA. Even though FSG was able to improve time efficiency, it has a weakness in memory management.

Clearly, even though there are many algorithms proposed for string graph construction, still there is a need for an algorithm which can improve speed without compromising space efficiency. To fill this gap, in this thesis, we propose a novel and linear time efficient algorithm SOF (**S**tring graph construction by **O**verlap **F**iltering) for constructing a string graph. Experiments show that SOF is at least 2.3 times faster than SGA while it takes approximately same amount of memory as SGA.

Algorithms like SGA and FSG which uses FM-index [FM05] to construct a string graph, follows two steps: 1. identify all of the overlaps using FM-index 2. perform an extension of the overlaps to identify irreducible edges. While SOF follows a similar approach in computing overlaps by using FM-index, it applies a completely different approach in removing transitive edges. Using properties of transitive edges, SOF can efficiently remove transitive edges without performing any extension of the overlaps. As a result, SOF obviates the necessity to load two BWTs (both for forward and reverse reads) simultaneously in the memory and achieves space-efficiency.

The rise of NGS technologies have also allowed us to analyze the genomic content of a microbial community in a culture-independent way. This branch of research is called metagenomics where genomics techniques are applied on microbial data. One of the key research in metagenomics is to identify functional and metabolic potential present in the sequenced microbial community. This can be done by homology detection where an annotated reference sequence is used to identify significantly similar reads (homologs). Though there is availability of homolog detection tools [BLA13] [AMS+97], recently a set of tools have been designed specially for homolog detection in metagenomic data [ZYY15a], [ZYY15b], [ZYY17]. In [ZYY17], it is shown that string graph is useful for assembly of metagenomic peptide data which can be used for search and alignment in later steps.

In this thesis, we show that SOF has linear time computational complexity which is completely independent of the alphabet size of the sequence dataset. Therefore, we predict that SOF has the potential to be the best alternative for constructing string graph from datasets with larger alphabet size (e.g. protein sequence data), consequently making metagenomic homolog search more time-efficient.

In recent years, user experience design for an application is gaining more attention while functional efficiency was traditionally the concern [HR19]. With this in mind, we have implemented SOF as a plugin of SGA assembly pipeline [Mor] which makes it easy for the user to access the preprocessing and postprocessing steps provided in the SGA.

CHAPTER 3: STRING GRAPH CONSTRUCTION BY OVERLAP FILTERING

In this chapter, we discuss in details about our proposed algorithm SOF (**S**tring graph construction by **O**verlap **F**iltering). We discuss some definitions, notations, and properties related to string graph in section 3.1 which are necessary for the explanation of the algorithm described in section 3.2. Finally, we discuss about the implementation strategies in section 3.3 which we use to make this algorithm space efficient. Implementation of this algorithm is available in the github repository [Mor].

3.1 Preliminaries

3.1.1 Sequences

Let s be a string over alphabet Σ, $s[i]$ be the i-th symbol of s, $s[i,j]$ be the substring $s[i][i+1]...s[j]$ and $|s|$ be the length of s. Let r_i be a read sequence and $R = \{r_1, r_2, r_3.r_n\}$ where n is the total number of reads. For a genome sequence, $\Sigma = \{A, T, G, C\}$. We denote N as total length of all reads, hence $N = \sum_{i=1}^{n} |r_i|$.

3.1.2 String Graph

Concept of the string graph is based on the concept of overlap graph. In an overlap graph, read sequence in R represents a vertex. There is an edge between two vertices if there is an overlap between two corresponding reads. String graph is defined as a refined version of an overlap graph, where a vertex is considered redundant if the corresponding read is identical to or substring of another read. Also, an edge is considered redundant if there is a path which represents an assembly same as the assembly constructed from that edge. This type of edge is called *transitive edge*. If an edge is not transitive, it is called *irreducible*. A string graph neither contains redundant vertices nor transitive edges.

To define a string graph, we define the term *overlap* and *edge* in the following way: a sequence s *overlaps* a sequence q and thereby forms an *edge* (s, q) if and only if suffix of s is the prefix of q. In other words, in an edge (s, q), q is $overlapped$ by s (i.e. prefix of q is the suffix of s). Let β be the overlap sequence with minimum length of μ, α be the unmatched sequence of s and γ be the unmatched sequence of q. In a string graph, each edge is bidirectional and each direction contains label along with end type information dictating where the label will be added during a walk in the graph [Mye05]. We define a tuple *(label, end)* for each direction. An *end* can be B or E representing the position – beginning or end of the source vertex sequence respectively – where the *label* should be added during a walk in the graph.

For two read sequences r_i and r_j where $r_i, r_j \in R$, if r_i and r_j forms an edge (r_i, r_j) in string graph, two directed edge $r_i \rightarrow r_j$ with tuple (γ, E) and $r_i \leftarrow r_j$ with tuple (α, B) are added in the description of that string graph. For edge (r_i, r_j), we call label in $r_i \rightarrow r_j$ as *forward label* and label in r$_i \leftarrow r_j$ as *backward label.*

Now, we formally define the concept of transitive edges. Let (s, q) be an edge with forward label γ_1 and (s, r) be another edge with forward label γ_2 where γ_1 is the prefix of γ_2. Then edge (s, r) is defined as transitive edge [SD10].

Figure 3.1 shows a simple string graph representation based on the definition discussed in this subsection.

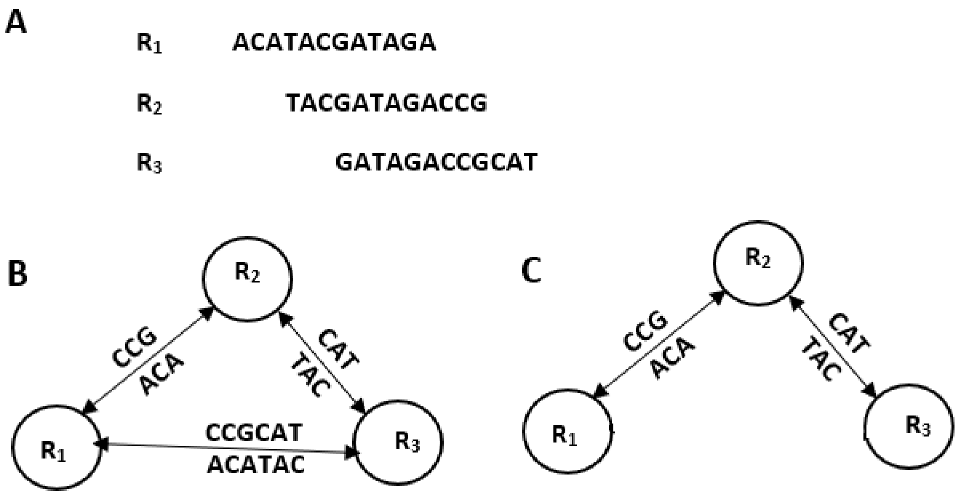

Figure 3.1: Simple string graph representation for three reads. **(A)** shows three overlapping reads R_1, R_2, R_3. **(B)** shows the overlap graph constructed from these reads. Since (R_1, R_3) is transitive, this edge is not be present in string graph shown in **(C)**.

3.1.3 FM Index

Like many previous algorithms [SD10][BVP$^+$16][BDVP$^+$17], SOF uses FM-index [FM05] to identify overlaps between reads. SOF uses BWT for multiple strings defined in [BCR11] where each read r_i is appended with \$ and \$ is the lexicographically smallest symbol in $\Sigma^\$ = \Sigma \cup \{\$\}$. Each end marker \$ is considered a different symbol for the purpose of uniquely sorting every suffixes of every read sequence. \$ appended after r_i is defined to be smaller than \$ append after r_j if $i < j$. Based on this definition, we define generalized suffix array for R, SA_R such that $SA_R[x] = (k, j)$ where suffix starting from index k of read j is the xth lexicographically smallest suffix in the set of all suffix constructed from the \$ appended read set R. From the definition of SA_R, we define BWT_R. If $SA_R[x] = (k, j)$, then $BWT_R[x] = r_j[k-1]$ if $k > 1$ or $BWT_R[x]$ =\$ if $k = 1$.

To find the existence of a particular pattern p in sequence set R, *Backward-Search* algorithm [FM05] is used on FM Index of R. *Backward-Search(p)* returns an interval $[l, u]$ which represents a set of indexes x in SA_R where for all $l \leq x \leq u$, $SA_R[x]$ contains start index of the suffix which contains p as prefix.

To identify all the overlaps for a sequence, an algorithm has to detect all the sequence whose prefix matches with the suffixes of that sequence. In the remaining part of this subsection, we discuss that it possible to represent all of the reads in a read set by their lexicographic order. We also discuss that from a set of lexicographic orders we can identify the set of reads which have prefixes that match with the suffix/es of a sequence in question.

Let $rank(c, i)$ denote total number of a character c in BWT_R within position $[1, i]$. Since SA_R represents all suffixes sorted in lexicographical order and for any x if $SA_R[x]$ = $(1, j)$ then $BWT_R[x]$ =\$, $rank(\$, x)$ in BWT_R represents the lexicographic order of the read sequence r_j . Consequently, every read sequence can be uniquely represented by the rank of the corresponding \$ symbol in BWT. We call rank of \$ as *terminal* and denote it by t. We represent a set of consecutive terminals T in the interval form $[t_l, t_u]$ and denote it by symbol $p(T)$ such that *p(T)* = $[t_l, t_u]$ where t_l and t_u are the lowest and highest terminal in the set T respectively . We call this interval *terminal interval.*

It is important to note that a terminal from BWT_R represents the lexicographic order of a sequence $r_i \in R$, not the order i within R. To get the order within R, SOF uses *Lexicographic Index Array* [SD10] denoted by L_R which stores order i of each sequence $r_i \in R$ according to the lexicographic order of r_i. Therefore, if r_i is j-th lexicographically smallest string, then $L_R[j] = i$. Consequently, we say that a terminal t from BWT_R *represents* a sequence $r_i \in R$ if and only if $L_R[t] = i$.

For a sequence s and index i, we call set of all terminals t_x from BWT_R representing sequences $r_y \in R$ as *overlapping terminals of s at i* where $s[i : |s|] = r_y[1 : |s| - i + 1]$ and $L_R[t_x] = y$. When all the overlapping terminals of s at i are represented in the form of terminal interval, we call this interval *overlapping terminal interval* of s at i. We denote set of all overlapping terminals of s at i by $T_R(s,i)$ and we denote $T_R(s) = \bigcup_{i=2}^{|s|-\mu+1} T_R(s,i)$. SOF uses *findOverlaps*(r,μ) algorithm provided in [SD10], to collect all overlapping terminal intervals for a sequence r that represents overlap of length at least μ.

3.1.4 Properties of Irreducible and Transitive Edges

In this thesis, we use an observation based on definition of transitive edges provided in [Mye05] that if a sequence s and a sequence q forms an edge (s,q), then all the other sequences r_i which are overlapped by both s and q makes all the edges (s,r_i) transitive. We formalize this observation in the following lemma.

Lemma 1. *For an edge (s,q), all edges (s,r_i) are transitive where $r_i \in R$ and $i = L_R[x]$ for all $x \in T_R(s) \cap T_R(q)$*

Proof. Let forward label for edge (s,q) be γ_{sq}. Let r be a read which is overlapped by both s and q. Let forward label for edge (s,r) be γ_{sr}. Now, $|\gamma_{sr}| > |\gamma_{sq}|$ must hold true because by definition of string graph, r can not be a substring of q. Since s overlaps both q and r, γ_{sq} is the prefix of γ_{sr}. Therefore, edge (s,r) is a transitive edge. □

From this lemma, we come to the following corollary. For an edge (s,q), let β_{sq} represent overlap sequence. For a sequence s, let $l_{\beta_{max(s)}}$ be the maximum overlap length by which s overlaps other sequences.

Corollary 1.1. *For an edge (s,q), if $|\beta_{sq}|$ = $l_{\beta_{max(s)}}$, then edge (s,q) is irreducible.*

Proof. From lemma 1, an edge (s, q) becomes transitive when there is a sequence r such that r overlaps q while r is overlapped by s. In other words, an edge (s, q) becomes transitive when both the edge (s, r) and edge (r, q) are present in the overlap graph. However when a sequence q is overlapped by sequence s with length $l_{\beta_{max(s)}}$, then there is no sequence which can be overlapped by s and overlap q. Hence, (s, q) is irreducible. □

3.2 Algorithm

3.2.1 Overview

SOF algorithm collects irreducible edges and removes transitive edges from the set of all overlaps. *Algorithm 1* provides the pseudo-code of the irreducible edge collection process. In this algorithm, for each read r_i, all of its overlaps are stored in a data-structure named *Current Read*. From the set of overlaps of r_i, all of the reads r_j that form maximum overlap with r_i are identified as irreducible edge forming reads. Then all of the reads r_k which are overlapped by both r_i and r_j are considered as transitive edge forming reads and records of r_k are subsequently deleted from the Current Read. From the remaining overlaps of r_i, next set of irreducible edges are collected and transitive edge forming reads are removed. This process continues until no irreducible edges for r_i remains to be collected. To identify common overlaps between r_i and r_j, another data-structure named *Overlap Container* is used. In both Current Read and Overlap Container, overlap information is stored in the form of overlapping terminal intervals. In subsection 3.2.2 and 3.2.3, we discuss in details about the Overlap Container data-structure and Current Read respectively. In subsection 3.2.4, we discuss how these two data-structures are used to collect irreducible edges and remove transitive edges.

3.2.2 Overlap Container Construction

Overlap Container Q_R is a data structure that contains overlap information for all reads contained in a read set R in the form of overlapping terminal intervals and the corresponding indexes. For each read $r_i \in R$, $Q_R[i]$ contains a set of tuples in the form $< I, s >$ where I is the overlapping terminal interval for read r_i at index s. In other words, $I = p(T_R(r_i, s))$. For a tuple v, variable $v.I$ denotes terminal interval stored in that tuple and $v.s$ denotes index stored in that tuple. Overlapping Terminal Intervals are obtained by performing *findOverlaps*(r_i, μ) [SD10] for all $r_i \in R$ using a BWT_R. Therefore, R and BWT_R are the parameters of *Construct-Overlap-Container* function. At line 2 in Algorithm 1, *Construct-Overlap-Container*(R, BWT_R) is called to create overlap container Q_R by the use of read set R and BWT_R.

3.2.3 Current Read Construction

For each read $r_i \in R$, a data-structure named *Current Read*, denoted by C_i, is constructed in Algorithm 1 (line 5). $C_i[k]$ stores the overlapping terminal intervals of r_i at k. Therefore, during initialization, for an index k where $1 < k \leq |r_i| - \mu + 1$, $C_i[k] = p(T_{R_2}(r, k))$ and $C_i[1] = \emptyset$. Initially, when C_i is constructed (line 5), there is at-most one interval in $C_i[k]$ for all k. However, in the next subsection, we will see that $C_i[k]$ may contain more than one intervals due to splitting of an interval.

3.2.4 Collecting Irreducible Edges

After construction of C_i (line 5), *Algorithm 1* collects irreducible edges from C_i and filters out transitive edges from this data structure until no irreducible edge remains to be collected (line 6-14). At each iteration , this algorithm identifies the index *idx* where $C_i[idx]$ contains all terminals representing reads that are overlapped by r_i with maximum overlap (line 6). Then it takes out all the terminal intervals I contained in $C_i[idx]$ (line 7-8). All the read sequence r_j, represented by these terminals $t \in I$ where $j = L_R[t]$, are considered to form irreducible edge (r_i, r_j) (line 10-11). Then using each of these reads r_j, the algorithm identifies common overlapping terminals between read r_i and read r_j. Since C_i and $Q_R[j]$ stores the overlapping terminals for read r_i and r_j respectively, *RemoveTransitive* function removes all the common terminals between C_i and $Q_R[j]$ from C_i (line 12-14). Reads r_k represented by these common terminals are considered as transitive edge (r_i, r_k) forming reads. Finally, all the irreducible edges are returned (line 15).

Now we prove that this algorithm correctly collects irreducible edges and filters out transitive edges. Read r_j, which forms maximum overlap with r_i, is considered for forming irreducible edge (r_i, r_j). This is correct according to corollary 1.1. Then all overlapping terminals, which represent reads r_k that are overlapped by both r_i and r_j, are considered as representative of reads that form transitive edge (r_i, r_k). This is also a correct process of identifying transitive edges according to lemma 1. Therefore, algorithm 1 correctly removes transitive edges from C_i.

Now we discuss some parts of the algorithm in more details . Intervals with the maximum overlap are situated in the lowest index idx where $C_i[idx] \neq \emptyset$. Therefore, GetMaxOverlap() function (line 6) actually returns this index idx for which $C_i[idx]$ contains at-least one element. The algorithm collects all the terminal intervals I from $C_i[idx]$. Then these intervals are removed from $C_i[idx]$ (line 8) so that these intervals will not be considered again in the next iteration and the value of idx will be different in the next iteration. Then, for each terminal $x \in I$, terminals are converted back to their actual order j in R using Lexicographic Index Array L_R (line 10). Edge (r_i, r_j) is stored in set E which contains all irreducible edges (line 11). Then, all overlapping terminals of read r_j which are common to read r_i are removed from C_i using *RemoveTransitive(ϕ, ψ)* function where ϕ is the set of terminal intervals and ψ is a terminal interval. *RemoveTransitive(ϕ, ψ)* function removes all the terminals common to both ϕ and ψ from the ϕ. *RemoveTransitive* function reduces or splits a interval in ϕ if that interval contains terminals represented by ψ. For example, suppose $\phi = \{[2,10]\}$ and $\psi = [4,6]$. Then *RemoveTransitive* function would make $\phi = \{[2,3],[7,10]\}$ which does not contain any terminals in ψ. If the value of ψ were $[8,12]$, *RemoveTransitve* would reduce the interval $[2,10]$ to interval $[2,7]$. Because of the possibility of split operation, ϕ may contain multiple intervals and *RemoveTransitive* may have to iterate through elements of ϕ to identify which interval needs to be splitted or reduced.

Finally, all irreducible edges, collected from each read r_i and stored in E, are returned (line 15).

Algorithm 1

```
 1: procedure COLLECT-IRREDUCIBLE-EDGES($R, BWT_R, L_R$)
 2:     $Q_R \leftarrow$ CONSTRUCT-OVERLAP-CONTAINER($R, BWT_R$)
 3:     $E \leftarrow \emptyset$
 4:     for each $r_i \in R$ do
 5:         Construct $C_i$
 6:         while $idx \leftarrow$ GETMAXOVERLAP() do
 7:             $I \leftarrow C_i[idx]$
 8:             $C_i[idx] \leftarrow \emptyset$
 9:             for each $t \in I$ do
10:                 $j = L_{R[t]}$
11:                 $E \leftarrow E \cup (r_i, r_j)$
12:                 for each $v \in Q_R[j]$ do
13:                     if $v.S < |r_i| - \mu$ then
14:                         REMOVETRANSITIVE($C[idx + v.S], v.I$)
15:     return $E$
```

3.2.5 Complexity

Now we discuss the complexity of the proposed algorithm. Overlap container construction takes $O(N)$ time because for each read r_i, collecting all overlapping intervals takes $O(|r_i|)$ time. For the same reason, Current Read construction also has $O(N)$ time complexity. In irreducible edge collection step, for each read r_i, read $e_j \in E_i$ is collected by SOF where E_i is the set of all reads that form an irreducible edge with r_i. For each $e_j \in E_i$, *Remove-Transitive* function is called at most $|\beta_{i,j}|$ times where $|\beta_{i,j}|$ represents overlap length between r_i and e_j. In each call, it performs interval splitting by traversing $|C_i[k]|$ intervals where $|C_i[k]|$ is the number of intervals present in $C_i[k]$ for $|r_i| - \beta_{i,j} < k \leq |r_i| - \mu + 1$. In case of the genome sequence, generally, $|C_i[k]|$ is bounded by the coverage d of read set R except in case of repeat sequences. Assuming all input sequences are equally likely, for r_i, at most $\sum_{j=1}^{|E_i|} |\beta_{i,j}|d$ interval traversal are performed. So for all $r_i \in R$, total complexity of this step is $O(B)$ where B is the sum of the length of all overlaps represented by the irreducible edges in the string graph. Therefore, we can say that complexity of the SOF algorithm is $O(N + B)$.

3.3 Implementation

While discussing the methodology in section 3.2 we have assumed that we are building the string graph for the read set obtained from the single strand of a genome. To incorporate overlaps created from both forward and the reverse strand, we include reverse complement of each reads in the read set. Let $\tilde{R}$ denote read set generated by taking reverse complement of all reads in R. So we consider R' as our new read set where $R' = R \cup \tilde{R}$. We precompute $BWT_{R'}$ and $L_{R'}$. When we implement SOF algorithm using the method described in section 3.2 , we observe that Overlap Container takes huge amount of memory space (almost 4 to 5 times more space than SGA). That happens because R' is twice the size of R and it collects overlaps by string matching with R'.

To reduce the large memory footprint, we have implemented Overlap Container using external memory. Instead of keeping all the overlaps in the main memory, SOF writes all overlaps of the Overlap Container in the disk. Then it iteratively collects overlaps from the disk to a *Partial Container* in a chunk by chunk manner. A *Partial Container* is basically a Overlap Container which contains overlap information for the subset of reads. We denote Partial Container $Q_{a,b}$ which only contains overlap information for read r_a to read r_b. So for a read r_j where $a \leq j \leq b$,

$Q_{a,b}[(j \bmod a) + 1] = Q_R[j]$.

While storing overlaps in $Q_{a,b}$, SOF selects b in such way that memory footprint of $Q_{a,b}$ does not exceed the memory footprint of $BWT_{R'}$. Since $BWT_{R'}$ is the largest data structure that we must load temporarily in the memory for the overlap detection, we limit our memory consumption for $Q_{a,b}$ based on the size of $BWT_{R'}$.This ensures that maximum memory usage by SOF does not exceed beyond the size of BWT.

Algorithm 2

```
 1: procedure Collect-Irreducible-Edges($R', BWT_{R'}, L_{R'}$)
 2:     $Q_{R'} \leftarrow$ Construct-Overlap-Container($R', BWT_{R'}$)
 3:     Write $Q_{R'}$ in Disk
 4:     $b \leftarrow 0$
 5:     while $b \neq 2n$ do
 6:         $b, E \leftarrow$ Collect-Irreducible-By-Partial-Container($R', L_{R'}, b + 1$)
 7:     return $E$
 8: procedure Collect-Irreducible-By-Partial-Container($R', L_{R'}, a$)
 9:     $Q_{a,b} \leftarrow$ Load-Partial-Container($a$)
10:     $E \leftarrow \emptyset$
11:     $idx \leftarrow 0$
12:     for each $r_i \in R'$ do
13:         Load $C_i$ from Disk
14:         while $idx \leftarrow$ GetMaxOverlap($idx$) do
15:             $I \leftarrow C_i[idx]$
16:             for each $t \in I$ do
17:                 $j = L_{R'}[t]$
18:                 if $a \leq j \leq b$ then
19:                     $E \leftarrow E \cup (r_i, r_j)$
20:                     for each  $v \in Q_{a,b}[(j \bmod a) + 1]$ do
21:                         if $v.S < |r_i| - \mu$ then
22:                             RemoveTransitive($C[idx + v.S], v.I$)
23:         Write $C_i$ to Disk
24:     return $b, E$
```

We have modified algorithm 1 to work for Partial Container and now discuss the modified version – Algorithm 2. In *Collect-Irreducible-Edges* function of this algorithm, after constructing Overlap Container $Q_{R'}$, SOF writes it on the disk (line 2-3). Then SOF iteratively collects irreducible edges by calling *Collect-Irreducible-By-Partial-Container* function (line 4-6). Variable b stores last read id of the partial container. For this reason, the while loop (in line 5) terminates when filtering process is done with the Partial Container that contains last chunk of overlaps collected from the overlap container.

Procedure *Collect-Irreducible-By-Partial-Container* in *Algorithm 2* is similar to *Collect-Irreducible-Edge* in *Algorithm 1*. We only discuss the differences here. In this procedure of Alogrithm 2, SOF collects overlaps in the partial container (line 9) instead of Overlap Container as in Algoirithm 1, and performs *RemoveTransitive* operation using the Overlapping Intervals collected from Partial Container. As a result, transitive edge removal process described in Algorithm 2 may not remove all reads that represents transitive edges in the Current Read C_i in just one iteration. This necessitates C_i to be stored in (line 23) and loaded from (line 13) disk in each iteration. Also, since partial container does not contain overlap information of all reads in R, only reads r_j, where $a \leq j \leq b$, is allowed perform transitive removal by calling *RemoveTransitive* Function. Lastly, *GetMaxOverlap* function (line 14) is slightly modified to take a parameter. *GetMaxOverlap(k)* returns the lowest index idx where $C_i[idx] \neq \emptyset$ and $idx > k$. This obviates the necessity to remove all the previously collected reads that represents irreducible edges.

3.4 Comparison with SGA

Lastly, as SOF uses methods and data-structure described in [SD10] which is later included in the SGA assembler [SD12], we briefly discuss similarity and difference between the two approaches. SOF is similar to SGA in identifying overlaps from the sequence data using FM Index. But SOF follows a completely different approach than SGA in collecting irreducible edges from the set of all overlaps. After identifying overlaps, SGA performs right extension for each of the overlap interval using BWT of the reverse reads. When a read is found to reach its terminal point during right extension, this read is considered to form an irreducible edge and all other non-terminated reads with the same extension are discarded as transitive edge forming reads. On the other hand, SOF doesn't perform the right extension and consequently, doesn't need to have two BWTs (both for forward and reverse reads) simultaneously in the memory. Instead, SOF stores all the overlaps for all of the reads in Overlap Container. Then for each read, using the overlap information stored in the overlap container, SOF identifies overlaps with maximum overlap length and considers them as irreducible edge forming reads. Common overlaps between the read and its overlaps with maximum length are discarded as transitive edge forming reads. We also observe that during the right extension stage, SGA iterates over all characters in the alphabet set to check whether it is possible to extend the sequence with that particular character. For a small alphabet set as in genome sequence, the cost for iterating over alphabet set is negligible, but for a set of sequence with larger alphabet set (e.g. protein sequence) the cost would be significant. Since SOF doesn't require to iterate over alphabet set in any stage, we hypothesize that our approach will significantly perform better for the sequence set with large alphabets.

CHAPTER 4: RESULTS AND DISCUSSION

4.1 Dataset and Experiment Setup

We have implemented our algorithm SOF using C++11 and compared its performance with the performance of SGA. SGA provides two string graph construction algorithm based on both exact and inexact overlap. We call SGA-Inexact and SGA-Exact to subprograms based on inexact and exact overlap respectively. We compared the performance of SOF with both SGA-Inexact and SGA-Exact. Like FSG, we have included our program as a plugin of SGA. Therefore, we can compare the performance on the same datasets that have been preprocessed by the same preprocessing pipeline. However, we have not included FSG in our comparison because FSG requires input read sequence dataset and BWT made from both the original read sequences and their reverse complements, but FSG does not provide any subprogram to do that processing.

We have compared the performance on 5 datasets: *Arabidopsis thaliana* (SRR7637136), *Caenorhabditis elegans* (SRR7594466), *Deinococcus radiodurans* (SRR1027618), *Escherichia coli* (SRR857279) and *Danio rerio* (ERR2094318) . For each dataset, we have compared the performance of three different minimum overlap length: 55%, 65% and 75% of the read length. Table 4.1 shows the size information of these 5 dataset in terms of the number of reads, read length, and number of base pair.

All these dataset have gone the through the same preprocessing steps of SGA (*preprocess, index, correct, index, filter*) with default parameters. We have performed BWT indexing using the *ropebwt* algorithm. Following the examples provided for the sample data in SGA implementation, we have chosen the value of -x parameter to be 2 in the *filter* step. SGA constructs the string graph through *overlap* subprogram of the assembly pipeline.

Table 4.1: Size information for 5 paired-end sequence dataset

	Number of reads	Read length	Number of base pair
A. thaliana	25,190,636	101	5.1G
C. elegans	67,586,475	76	10G
D. radiodurans	27,063,778	90	4.9G
E. coli	4,273,258	150	1.3G
D. rerio	64,904,138	101	13.1G

We have executed *overlap* subprogram in its default settings (i.e. in serial processing mode). SOF constructs string graph by execution of the *sof* subprogram. Since SOF requires BWT to be built from both original read and their reverse complement, we have written a subprogram *indexSOF* which builds BWT of both forward and reverse reads using *ropebwt* algorithm. We executed *indexSOF* on the preprocessed data, before executing *sof* subprogram.

We conducted our experiment on a cluster server node built using Intel(R) Xeon(R) CPU E5-2640 v4 2.40GHz processor with 32 cores and 512G RAM. The result of our experiment is shown in table 4.2 and 4.3.

4.2 Results and Discussion

From the experiment (table 4.2), we observe that SOF is at least 2.3 times faster than both SGA-Inexact an SGA-Exact. We also observe that with the decrease of minimum overlap length, performance ratio measured by (SGA time)/(SOF time) generally increases suggesting better time efficiency of SOF for lower minimum overlap length. In terms of memory, SOF takes almost same amount of memory as SGA (table 4.3). This is because of the implementation technique followed by SOF. However, we also see few cases where memory consumption of SOF is higher than SGA with highest (SOF memory/ SGA memory) being 1.8 .

Table 4.2: Performance comparison in terms of run-time among SGA-Inexact, SGA-Exact and SOF for five datasets with three different minimum overlap length for each datase

Dataset	Minimum overlap (% of read length)	SGA-Inexact time (second)	SGA-Exact time (second)	SOF time (second)	SGA-Inexact time/SOF time	SGA-Exact time/ SOF time
A. thaliana	55	16414	15900	4286	3.8	3.7
	65	12553	12642	3880	3.2	3.3
	75	10523	11100	3142	3.3	3.5
C. elegans	55	13810	16273	3400	4.1	4.7
	65	8880	11128	2780	3.2	4
	75	6592	8311	2174	3	3.8
D. radiodurans	55	1748	2409	749	2.3	3.21
	65	1468	1913	600	2.4	3.2
	75	1248	1386	503	2.5	2.75
E. coli	55	1269	1133	493	2.6	2.3
	65	1125	1039	440	2.6	2.4
	75	1007	939	384	2.6	2.5
D. rerio	55	175895	205760	25787	6.8	8
	65	92997	74987	16293	5.7	4.7
	75	44342	37332	10311	4.3	3.7

4.3 Validation

Finally, we validate the correctness of our implementation by comparing the assembly quality of the resulting string graph constructed by SGA-Inexct, SGA-Exact, and SOF. We performed assembly of the string graph constructed from *E. coli* and *A. thailana* dataset using the *assemble* subprogram of the SGA pipeline with minimum overlap length being 75% of the read length (same minimum overlap length is used in the string graph construction step). We compared the assembly quality using QUAST software [GSVT13]. Table 4.4 shows that the quality of assembled contigs generated from the string graph of SGA-Inexact, SGA-Exact and SOF are almost identical indicating the identity in the string graph constructed by these three algorithms.

Table 4.3: Performance comparison in terms of memory among SGA-Inexact, SGA-Exact and SOF for five datasets with three different minimum overlap length for each dataset

Dataset	Minimum overlap (% of read length)	SGA-Inexact memory (MB)	SGA-Exact memory (MB)	SOF memory (MB)	SOF memory/ SGA-Inexact memory	SOF memory/ SGA-Exact memory
A. thaliana	55	2439	2440	2791	1.1	1.1
	65	2439	2439	2959	1.2	1.2
	75	2439	2439	3079	1.3	1.3
C. elegans	55	2154	1279	2327	1.1	1.8
	65	2153	2153	2400	1.1	1.1
	75	2152	2152	2537	1.1	1.2
D. radiodurans	55	452	452	508	1.1	1.1
	65	448	449	496	1.1	1.1
	75	448	448	558	1.2	1.2
E. coli	55	258	259	325	1.3	1.3
	65	258	258	346	1.3	1.3
	75	258	258	361	1.4	1.4
D. rerio	55	6675	6675	6503	1	1
	65	6673	6673	6475	1	1
	75	6669	6670	6396	1	1

Table 4.4: Assembly quality comparison for the string graph constructed by SGA-Inexact (SGA-In) SGA-Exact (SGA-Ex) and SOF on $e.coli$ dataset and $a.thaliana$ dataset

	E. coli dataset			*A. thaliana* dataset		
	SGA-In	SGA-Ex	SOF	SGA-In	SGA-Ex	SOF
No. of contigs (>=0 bp)	705	705	705	704856	704856	707567
No. of contigs (>= 5000 bp)	117	117	117	3939	3939	3939
No. of contigs (>= 10000 bp)	93	93	93	471	471	471
No. of contigs (>= 25000 bp)	63	63	63	2	2	2
No. of contigs (>= 50000 bp)	29	29	29	0	0	0
Total length (>= 10000 bp)	4339159	4339159	4339159	5869635	5869635	5869635
Total length (>= 25000 bp)	3846153	3846153	3846153	52791	52791	52791
Total length (>= 50000 bp)	2554704	2554704	2554704	0	0	0
Largest contig	158085	158085	158085	26896	26896	26896
N50	55046	55046	55046	3062	3062	3062
N75	35706	35706	35706	1576	1576	1576

CHAPTER 5: CONCLUSION

In this thesis, we propose a novel and linear time algorithm SOF for constructing string graph. Our work takes the idea of overlap detection using FM-index from [SD10], but applies a novel approach of irreducible edge collection and transitive edge removal using the property of irreducible edge and transitive edge. We implemented our algorithm and executed the program on five real datasets for three different minimum overlap length. Experimental result shows that our program SOF is at least 2.3 times faster than the most popular string graph-based assembler SGA. We have designed SOF in such way that it does not take more memory than the memory needed to store two BWT of reads. For this reason, SOF takes almost same space as SGA. Therefore, in this thesis, we are able to show that it is possible to increase the speed of string graph construction algorithm without compromising space efficiency.

We have open sourced our implementation so that anyone can use this program to efficiently construct string graph. String graph construction is the major step in the string-graph based assembly. However there are other preprocessing steps and post processing steps required to perform complete assembly operation. SGA has already developed all the necessary preprocessing and post-processing steps. We have implemented this program as a plug-in of SGA so that users can easily access the steps provided in SGA.

Our proposed algorithm is alphabet size independent. For this reason, this algorithm is supposed to perform significantly better that other existing algorithms when sequence data has larger alphabet size. Therefore, we plan to explore the efficiency of SOF in sequence assembly of larger alphabet size. [ZYY17] has already shown the usefulness of string graph in homolog detection from metagenomic data. In that paper, authors constructed string graph from the metagenomic peptide sequence. Since alphabet size does not preclude efficiency of SOF, we believe that SOF has the potential to be the best program in metagenomic homolog search.

APPENDIX : LIST OF ABBREVIATIONS

BWT	Borrows Wheeler Transform
FSG	Fast String Graph
NGS	Next generation Sequencing
SGA	String Graph Assembler
SOF	String graph construction using Overlap Filtering

LIST OF REFERENCES

[AMS+97] Stephen F. Altschul, Thomas L. Madden, Alejandro A. Schäffer, Jinghui Zhang, Zheng Zhang, Webb Miller, and David J. Lipman. Gapped BLAST and PSI-BLAST: A new generation of protein database search programs, 1997.

[BBC14] Ilan Ben-Bassat and Benny Chor. String graph construction using incremental hashing. *Bioinformatics*, 30(24):3515–3523, 12 2014.

[BCR11] Markus J. Bauer, Anthony J. Cox, and Giovanna Rosone. Lightweight BWT construction for very large string collections. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 6661 LNCS, pages 219–231. Springer, Berlin, Heidelberg, 2011.

[BDVP+17] Paola Bonizzoni, Gianluca Della Vedova, Yuri Pirola, Marco Previtali, and Raffaella Rizzi. FSG: Fast String Graph Construction for De Novo Assembly. *Journal of computational biology*, 24(10):953–968, 2017.

[BLA13] BLAST. BLAST Basic Local Alignment Search Tool, 2013.

[Blo02] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 2002.

[BVP+16] Paola Bonizzoni, Gianluca Della Vedova, Yuri Pirola, Marco Previtali, and Raffaella Rizzi. LSG: An External-Memory Tool to Compute String Graphs for Next-Generation Sequencing Data Assembly. *Journal of Computational Biology*, 23(3):137–149, 3 2016.

[BW94] M Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, 1994.

[CR12] Rayan Chikhi and Guillaume Rizk. Space-efficient and exact de Bruijn graph representation based on a bloom filter. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2012.

[DIS+16] Rosalinda D'Amore, Umer Zeeshan Ijaz, Melanie Schirmer, John G. Kenny, Richard Gregory, Alistair C. Darby, Migun Shakya, Mircea Podar, Christopher Quince, and Neil Hall. A comprehensive benchmarking study of protocols and sequencing platforms for 16S rRNA community profiling. *BMC Genomics*, 2016.

[FM05] Paolo Ferragina and Giovanni Manzini. Indexing Compressed Text. *Journal of the ACM*, 52(4):552–581, 2005.

[GK12] Giorgio Gonnella and Stefan Kurtz. Readjoiner: a fast and memory efficient string graph-based sequence assembler. *BMC Bioinformatics*, 13(1):82, 5 2012.

[GMM16] Sara Goodwin, John D. McPherson, and W. Richard McCombie. Coming of age: Ten years of next-generation sequencing technologies, 2016.

[GSVT13] Alexey Gurevich, Vladislav Saveliev, Nikolay Vyahhi, and Glenn Tesler. QUAST: Quality assessment tool for genome assemblies. *Bioinformatics*, 29(8):1072–1075, 4 2013.

[HFF+08] David Hernandez, Patrice François, Laurent Farinelli, Magne Østerås, and Jacques Schrenzel. De novo bacterial genome sequencing: Millions of very short reads assembled on a desktop computer. *Genome Research*, 18(5):802–809, 5 2008.

[HR19] Nikolay Harutyunyan and Dirk Riehle. User Experience Design in Software Product Lines. volume 6, pages 7503–7512, 2019.

[KK10] Martin Kircher and Janet Kelso. High-throughput DNA sequencing - Concepts and limitations, 2010.

[KR10] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 2010.

[LLL$^+$14] Lin Liu, Yinhu Li, Siliang Li, Ni Hu, Yimin He, Ray Pong, Danni Lin, Lihua Lu, and Maggie Law. Comparison of next-generation sequencing systems. In *The Role of Bioinformatics in Agriculture*. 2014.

[Mor] S.M. Iqbal Morshed. SOF Github repository : https://github.com/iqbalmorshed/sof/.

[Mye05] Eugene W. Myers. The fragment assembly string graph. *Bioinformatics*, 21(SUPPL. 2):ii79–ii85, 9 2005.

[PTW02] Pavel A Pevzner, Haixu Tang, and Michael S Waterman. An Eulerian path approach to DNA fragment assembly. *Proceedings of the National Academy of Sciences*, 98(17):9748–9753, 2002.

[SD10] Jared T Simpson and Richard Durbin. Efficient construction of an assembly string graph using the FM-index. *Bioinformatics*, 26(12):367–373, 2010.

[SD12] Jared T Simpson and Richard Durbin. Efficient de novo assembly of large genomes using compressed data structures. *Genome Research*, 22(3):549–556, 2012.

[SN18] Jang Il Sohn and Jin Wu Nam. The present and future of de novo whole-genome assembly. *Briefings in Bioinformatics*, 19(1):23–40, 10 2018.

[SNC06] F. Sanger, S. Nicklen, and A. R. Coulson. DNA sequencing with chain-terminating inhibitors. *Proceedings of the National Academy of Sciences*, 2006.

[SSK13] Kamil Salikhov, Gustavo Sacomoto, and Gregory Kucherov. Using cascading bloom filters to improve the memory usage for de Brujin graphs. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2013.

[ZYY15a] C. Zhong, Y. Yang, and S. Yooseph. GRASP: Guided Reference-based Assembly of Short Peptides. *Nucleic Acids Research*, 43(3):e18–e18, 2 2015.

[ZYY15b] Cuncong Zhong, Youngik Yang, and Shibu Yooseph. GRASPx: Efficient homolog-search of short-peptide metagenome database through simultaneous alignment and assembly. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 9096(Suppl 8):442–443, 2015.

[ZYY17] Cuncong Zhong, Youngik Yang, and Shibu Yooseph. GRASP2: Fast and memory-efficient gene-centric assembly and homolog search. In *2017 IEEE 7th International Conference on Computational Advances in Bio and Medical Sciences (ICCABS)*, pages 1–1. IEEE, 10 2017.