國立臺灣大學電機資訊學院資訊工程學研究所

博士論文

Department of Computer Science and Information Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Doctoral Dissertation

針對高通量定序資料之可延展序列組合演算法

Scalable Assembly of High-Throughput *De Novo* Sequencing Data

陳建智

Chien-Chih Chen

指導教授：賴飛羆、陳俊良、何建明 博士

Advisor: Feipei Lai, Chuen-Liang Chen, Jan-Ming Ho, Ph.D.

中華民國 102 年 1 月

Jan, 2013

致謝

這篇論文的完成要感謝許多人的幫忙與支持，首先要感謝我碩士班及博士班的指導教授陳俊良老師。從碩士班到博士班一路走來，老師面對問題的處理方式及處事態度一直是我學習的目標。再來要感謝中研院的何建明老師，來中研院的最大的收穫就是學到面對科學應有的態度:像是發掘問題永遠比解決問題來得重要；從三個 **I** (1) <u>**I**</u>nnovation、(2) <u>**I**</u>mpact 跟 (3) <u>**I**</u>mportance 去衡量研究的價值，都是從老師身上學到的重要準則。還要感謝賴飛羆老師的協助讓我能夠順利完成學業。

此外感謝口試委員趙坤茂教授、高成炎教授、劉邦鋒教授、洪士灝教授跟林仲彥教授提出的建議跟指正，以及文鏈學長跟育榮學長提供的意見跟協助，使得論文內容更加完整。

最後感謝我的父母及家人對我的支持跟包容。僅以這篇論文獻給我的太太育妏和我的女兒昭羽。

摘要

DNA 定序技術是研究分子生物的最重要的步驟之一，用來判定 DNA 片段所代表的序列資訊。隨著次世代定序技術的發展，基因體學跟轉錄體學的研究已進入到下一個里程。然而目前的定序技術無法將基因體或轉錄體從頭到尾一次定序完成，必需將樣本切成很多短的片段，再透過序列組合演算法將基因體或轉錄體還原。因此序列組合組合演算法一直是生物資訊中的核心問題。序列組合的挑戰在於: (1)序列中的錯誤、(2)序列中有重複片段、(3)序列中每各位置被定序到的次數不平均、(4)針對大量資料時計算複雜度的問題。其中，隨著次世代定序技術所帶來快速增加的資料量，使得序列組合軟體能具備可延展運算能力來處理大量序列資料為最迫切的需求。這類的需求剛好符合雲端運算的運作模式。在雲端雲算中，使用者可以依需求透過網路向供應商去設置幾千台的電腦資源來做大量資料的平行處理。針對大量的資料，這種可延展的雲端應用通常是發展在 MapReduce 的架構下。

在本篇論文中，我們提出一個以 MapReduce 為架構的高通量序列組合演算法，稱作 CloudBrush。CloudBrush 是以雙向串圖(bidirected string graph)為基礎，主要分成兩個階段: 建構圖形(graph construction)和簡化圖形(graph simplification) 。在建構圖形的步驟，我們以一個讀序(read)當做圖上的一個節點(node)，節點跟節點間的圖邊(edge)定義為讀序跟讀序間的重複(overlap)。我們提出一個前綴延伸(prefix-and-extend)的演算法來判定兩兩讀序間的重複。在圖形簡化的步驟，我們採取常見的串圖簡化方法包括:傳遞簡約(transitive reduction)、路徑壓縮(path compression)、移除分枝結構(tips removal)和移除氣泡結構(bubble removal) 。此外我們提出一個新的串圖簡化方式:圖邊調整(edge adjustment)來移除串圖內因序列錯誤所造成的複雜結構。此簡化方式主要是利用圖形中鄰居間的序列資訊來做判斷，移除可能是錯誤造成的圖邊。

在效能評比部分，我們利用 Genome Assembly Gold-Standard Evaluation 為基準

來衡量 CloudBrush 組出的序列品質並跟其他序列組合工具做比較。實驗結果顯示我們的演算法可組出中等長度的 N50，且不易發生序列誤接(mis-assembly)的情形。

此外針對轉錄體的序列資料我們另外提出一個 T-CloudBrush 的流程。T-CloudBrush 主要是利用多重參數(multiple-*k*)來克服轉錄體序列資料深度(coverage)差異的問題。多重參數的概念主要是來自於這項觀察：在考慮序列組出的品質情況下，序列資料的深度跟序列組合演算法使用的重複區域(overlap size)參數呈現正相關的關係。實驗結果顯示 T-CloudBrush 可以增進轉錄體序列組合的品質。

綜合上述，本篇論文在可延伸的運算架構底下，探討序列組合算法所面臨的挑戰，並針對大資料的處理，序列錯誤及序列資料深度差異的問題提出可能的解法。

關鍵字： 序列組合、平行運算、基因體學、轉錄體學、生物資訊。

Abstract

DNA sequencing is one of the most important procedures in molecular biology research for determining the sequences of bases in specific DNA segments. With the development of next-generation sequencing technologies, studies on genomics and transcriptomics are moving into a new era. However, the current DNA sequencing technologies cannot be used to read entire genomes or a transcript in 1 step; instead, small sequences of 20–1000 bases are read. Thus, sequence assembly continues to be one of the central problems in bioinformatics. The challenges facing sequence assembly include the following: (1) sequencing error, (2) repeat sequences, (3) nonuniform coverage, and (4) computational complexity of processing large volumes of data. From these challenges, considering the rapid growth of data throughput delivered by next-generation sequencing technologies, there is a pressing need for sequence assembly software that can efficiently handle massive sequencing data by using scalable and on-demand computing resources. These requirements fit in with the model of cloud computing. In cloud computing, computing resources can be allocated on demand over the Internet from several thousand computers offered by vendors for analyzing data in parallel. Such cloud-computing applications are constantly being developed for large datasets and are run under the framework of MapReduce.

In this dissertation, we have proposed CloudBrush, a parallel pipeline that runs on the

MapReduce framework for *de novo* assembly of high-throughput sequencing data. CloudBrush is based on bidirected string graphs and its analysis consists of 2 main stages: graph construction and graph simplification. During graph construction, a node is defined for each nonredundant sequence read, and the edge is defined for overlap between reads. We have developed a prefix-and-extend algorithm for identifying overlaps between a pair of reads. The graph is further simplified by using conventional operations such as transitive reduction, path compression, tip removal, and bubble removal. We have also introduced a new operation, edge adjustment, for removing error topology structures in string graphs. This operation uses the sequence information of all graph neighbors for each read and eliminates the edges connecting to reads containing rare bases.

CloudBrush was evaluated against Genome Assembly Gold-Standard Evaluation (GAGE) benchmarks to compare its assembly quality with that of other assemblers. The results showed that our assemblies have a moderate N50, a low misassembly rate of misjoins, and indels. In addition, we have introduced 2 measures, precision and recall, to address the issues of faithfully aligned contigs in order to target genomes. Compared with the assembly tools used in the GAGE benchmarks, CloudBrush was found to produce contigs with high precision and recall.

We have also introduced a T-CloudBrush pipeline for transcriptome data.

T-CloudBrush uses the multiple-*k* concept to overcome the problem of nonuniform coverage of transcriptome data. This concept is based on observation of the correlation between sequencing data coverage and the overlap size used during assembly. The experiment results showed that T-CloudBrush improves the accuracy of *de novo* transcriptome assembly.

In summary, this dissertation explores the challenges facing sequence assembly under the scalable computing framework and provides possible solutions for the problems of sequencing errors, nonuniform coverage, and processing of large volumes of data.

Keywords: sequence assembly, parallel computing, genomics, transcriptomics, bioinformatics.

Table of Contents

List of Figures

List of Tables

Chapter 1

1. Introduction

With the rapid development of next-generation sequencing (NGS) technologies, the cost of DNA sequencing continues to fall rapidly, faster than the rate according to Moore's law for computing costs [50]. To keep pace with the increasing availability of sequencing data, ever more efficient or scalable fundamental sequence tools are needed.

A fundamental step in analyzing *de novo* sequencing data, in the absence of a reference genome, is genome assembly—the problem of determining the sequence of an unknown genome—with the most well-known assembly problem being the Human Genome Project. In general, the genome of an organism offers great insight into its phylogenetic history, interaction with the environment, and internal function [31]. Thus, *de novo* assembly continues to play a central role in biology for the sequencing of novel species. Figure 1 shows a generic pipeline of *de novo* sequencing projects. This dissertation focuses on the subject of *de novo* assembly.

With the evolution of genome sequencing technology, methods for assembling genomes have changed. Therefore, it is an interesting aspect to study the evolution in graph structure with the growth in read length and coverage, and thus develop algorithms that efficiently assemble next generation genomic sequence data.

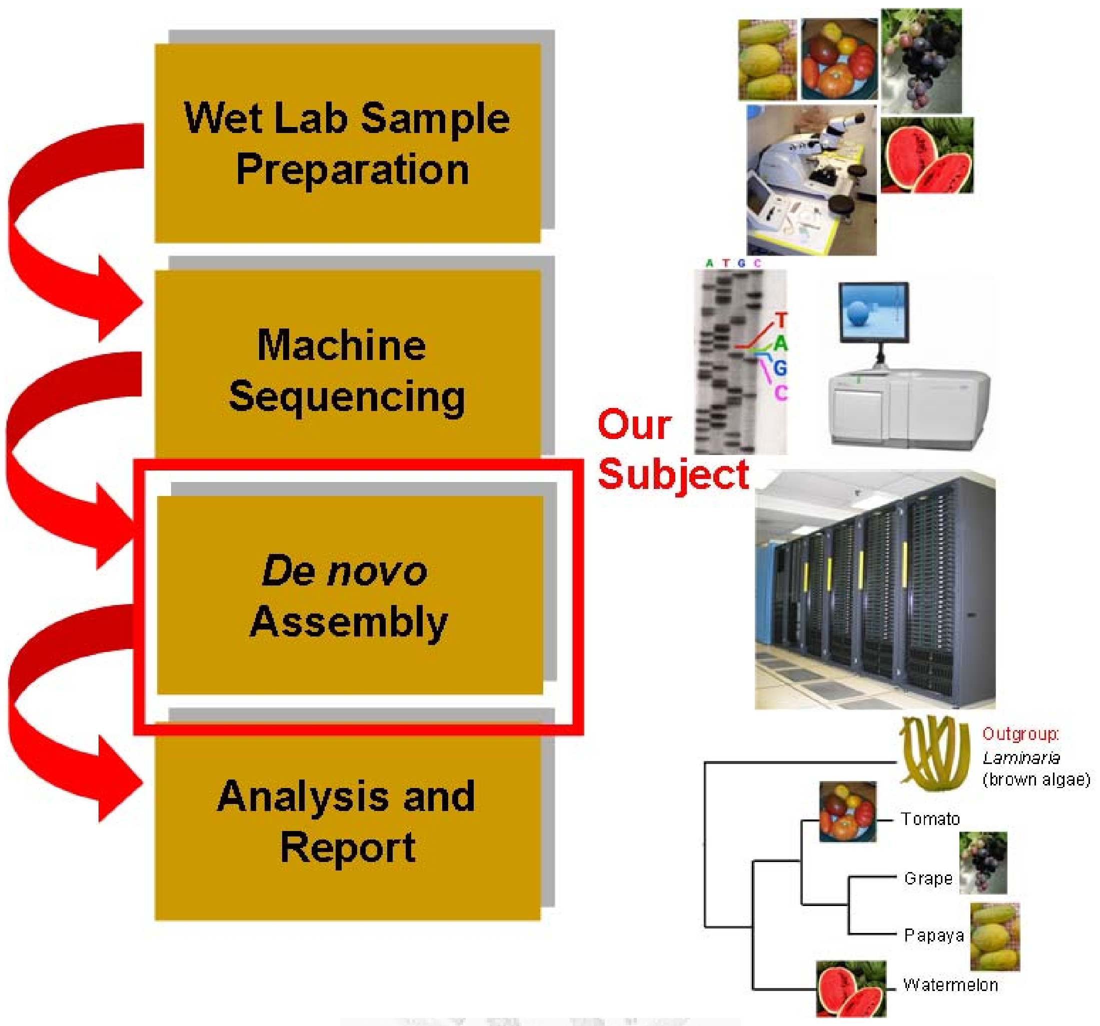


Figure 1. Generic pipeline of *de novo* sequencing projects.

1.1 Shotgun Sequencing and Assembly

The process through which scientists decode the DNA sequence of an organism is called sequencing. In 1977, Frederick Sanger developed the basic sequencing technology that is still widely used today [22]. Although genomes vary in size from millions of base pairs (bp) in bacteria to billions of base pairs in humans, the chemical reactions researchers use to decode the DNA base pairs are accurate for only about 600–700 bp at a time. To overcome this limitation, a technique called shotgun sequencing has been developed. The shotgun process involves shearing the genome of an organism into multiple small fragments. Following this, the ends of the fragments (called reads) are decoded via sequencing. In most sequencing projects, the fragment sizes are carefully controlled. This provides a link between the reads generated from the ends of the same fragment, which are called paired ends or mate pairs. Figure 2 shows an illustration of the shotgun sequencing process. Reconstructing a complete genome from a set of reads requires an assembly program.

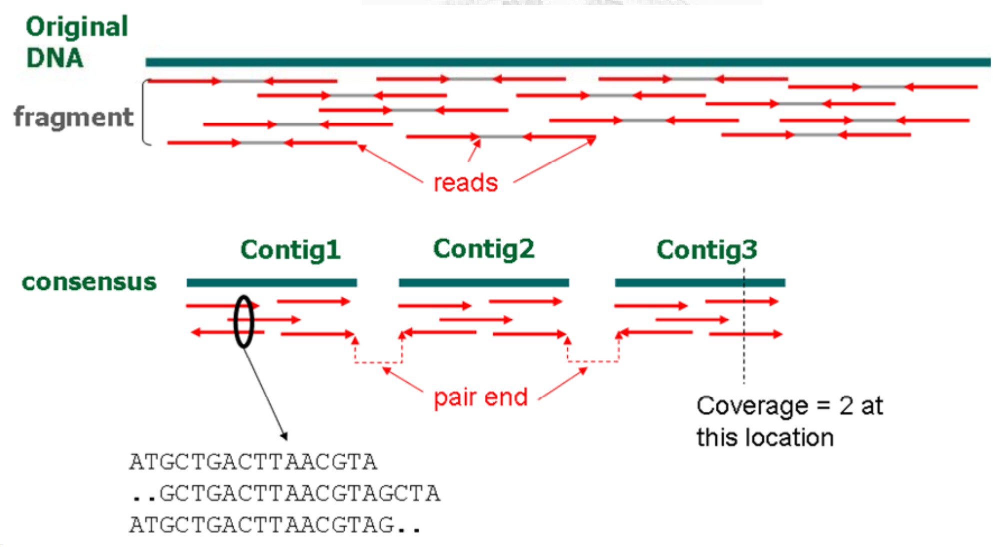


Figure 2. Illustration of *de novo* assembly.

The assembly program relies on the basic assumption that 2 reads that share the same string of letters originate from the same place in the genome (Figure 2). Using such overlaps between the reads, the assembly program can join the reads together and produce contiguous pieces of DNA (contigs). Note that shotgun sequencing can be viewed as a random sampling process, with each DNA fragment originating from a random location within the genome [30]. Thus, in the same way raindrops will eventually cover a whole sidewalk, it is possible to sequence an entire genome by oversampling the genome to ensure each position is seen. Eric Lander and Michael Waterman provide a statistical model [33] to examine the correlation between the oversampling of the genome (also called coverage) and the number of contigs that can be reconstructed by an idealized assembly program.

The high cost of Sanger's sequencing technology has long been a limiting factor for genome projects [12]. Current NGS technologies have far greater speed and much lower costs than Sanger sequencing. However, this reduction in cost comes at the expense of read lengths, with new sequencing having much shorter reads (30–400 bp). Therefore, their application has mainly been restricted to resequencing projects [13, 51], where a good reference sequence exists. For the *de novo* sequence assembly, assembling a large genome (>100 Mbp) using NGS data is a challenge. Thus, while sequencing cost is no longer a limiting factor for most *de novo* large genome projects, sequence assembly remains a major challenge.

1.2 Review of Sequencing Technology

There are many factors to consider in DNA sequencing, such as read length, bases per second, and raw accuracy. Extensive research and development has led to an exponential reduction in cost per base. Automated sequencing based on the Sanger method was developed in 1980 [49, 53]. It dominated the industry for almost 2 decades, and led to a number of monumental accomplishments, including the only finished-grade human genome sequence. Subsequent improvements in the Sanger method have increased the efficiency and accuracy by more than 3 orders of magnitude. At each step, more sophisticated DNA sequencing instruments, programs, and bioinformatics have provided more automation and higher throughput. A further revolution in sequencing began around 2005, when several massively parallel sequencing methods became available and began to produce massive throughput at far lower costs than Sanger sequencing. This allowed larger-scale production of genomic sequences.

The automated Sanger method is considered a “first-generation” technology, while newer methods are NGS technology. NGS platforms include the Genome Sequencer from Roche 454 Life Sciences (www.454.com), the Solexa Genome Analyzer from Illumina (www.illumina.com), and the SOLiD System from Applied Biosystems (www.appliedbiosystems.com). Common to all these technologies is the high degree of parallelism in the sequencing process. Millions of DNA fragments are immobilized to a surface and then sequenced simultaneously, leading to a throughput order of a magnitude higher than that achievable through Sanger sequencing. This performance comes at a price: read lengths are considerably shorter, ranging from 25–50 bp (SOLiD) to 400 bp (454 Titanium instrument). Table 1 shows a partial comparison of sequencing technology [25].

Refinements and automation have greatly improved cost effectiveness. In 1985, the cost of reading 1 single base was $10, while the same amount of money rendered 10,000 bases 20 years later [29]. This cost reduction mainly came about with the development of the 454 Sequencer, quickly followed by the Solexa/Illumina and ABI SOLiD technologies. Since then, the cost of sequencing a base has halved every 5 months, whereas, in accordance with Kryder’s law, the cost of storing a byte of data has halved every 14 months [37]. As this gap widens, the question of how to design higher throughput analysis pipelines becomes crucial [50].

Table 1. Comparison of sequencing technology

	First Generation	Next Generation		
Company	ABI	Roche	Illumina	ABI
Platform	3730xl	454 FLX Titanium	GAIIx	SOLiD-4
Sequencing Method	Sanger	Synthesis (pyrosequencing)	Synthesis	Ligation
Run time	2 h	10 h	14 d	12 d
Millions of reads/run	0.000096	1	320	> 840
Base/read	650	400	150	50
Reagent cost/Mb	\$1,500	\$12.40	\$0.12	< \$0.11
Machine cost	\$550,000	\$500,000.00	\$540,000.00	\$595,000
Error rate	0.1–1	1	≥0.1	>0.06
Error type	Substitution	Indel	Substitution	A-T bias

1.3 Review of Assembly Algorithms

1.3.1 Evolution of Assembly Algorithms

Early genome assemblers followed a simple but effective strategy in which the assembler greedily joins together the reads that are most similar to each other. This simple merging process will accurately reconstruct the simplest genomes, but it fails to do so for repetitive sequences that are longer than the read length. The problem is that the greedy algorithm cannot tell how to connect the unique sequences on either end of a repeat, and it can easily assemble together distant portions of the genome into misassembled contigs. Considering these repetitive sections in the genome, the sequence assembly problem may be modeled as a graph traversal problem. This formulation allows researchers to use techniques developed in the field of graph theory to solve the assembly problem. However, optimal path discovery is impractical because there can be an exponential number of paths between any source and sink node. Therefore, assemblers rely on heuristics and approximation algorithms to reduce the computational complexity.

The evolution of assembly algorithms has accompanied the development of sequencing technologies. Currently, there are 2 widely used classes of algorithms: string graph algorithms and de Bruijn graph algorithms. The string graph approach follows the overlap graph model by treating reads as nodes and assigning a edge between 2 nodes when the overlap (or intersection) between these 2 reads is larger than a cutoff length. A string graph is obtained by removing the transitive edges of the overlap graph [52]. In the overlap graph model, sequence assembly becomes the problem of identifying a path through the graph that contains all the nodes, i.e., a Hamiltonian path. This model became successful with the widespread use of the Sanger sequencing technology. Many widely used assembly programs adopted overlap graphs, such as Arachne [6], Celera Assembler [3], Phrap [7], Phusion [54], and Newbler [32]. However, with the far shorter reads and far higher coverage generated by these NGS technologies, computation of overlap became a bottleneck for the overlap graph model assembly. This is due to the considerable increase in the number of reads generated in a short-read NGS project as compared to the number generated using Sanger sequencing. For these reasons, the de Bruijn graph approach has been developed specifically to address the challenges of assembling very short reads.

Approaches using de Bruijn graphs approaches are based on early attempts to sequence genomes through a technique called sequencing by hybridization [2]. In de Bruijn graphs, instead of treating reads as nodes, each read is broken up into a collection of overlapping strings of length *k* (*k*-mers). For applications in DNA sequence assembly, de Bruijn graphs have a node for every *k*-mer observed in the sequence set and an edge between nodes if these 2 *k*-mers are observed adjacently in a read. It is easy to see that in a graph containing the information obtained from all the reads, a solution to the assembly problem corresponds to a path in the graph that uses all the edges, i.e., an Eulerian path. In late 2007 and early 2008, several next-generation de Bruijn graph assemblers were released for very short reads compatible with the Solexa technology, including VELVET [55], EULER-USR [20], ABySS [4], ALLPATHS-LG [35], and SOAPdenovo [18].

The development of assembly algorithms is tied closely to the development of sequencing technologies. Thus, research and practice have moved from the string graph approach for Sanger sequencing to the de Bruijn graph approach for NGS [12]. As short-read sequencing progressively shifts towards longer reads (>100 bp), the landscape of assembly software has had to adapt to high-coverage, longer reads. In the future, *de novo* assembly algorithms are likely to return to the string graph approach for long-read sequencing.

1.3.2 Comparison of String Graphs and de Bruijn Graphs

The main difference between string graph and de Bruijn graph algorithms is the method used to exploit the overlap information [12]. In the string graph algorithm, the identification of overlap between each pair of reads is explicit, typically by aligning all-against-all pairwise reads. In the de Bruijn algorithm, overlaps between reads are implicitly captured by decomposing all the reads into *k*-mers and simultaneously recording their neighboring relations. Figure 3 shows the differences between a string graph and a de Bruijn graph for assembly.

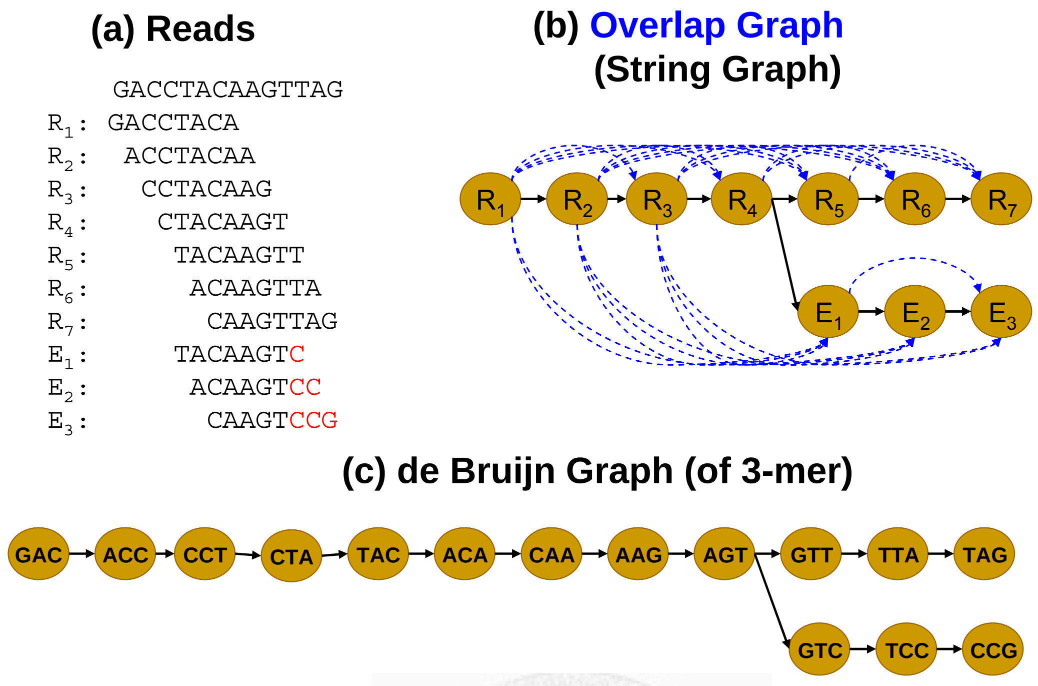


Figure 3. The difference between an overlap graph and a de Bruijn graph.

Paul Medvedev showed that both the de Bruijn graph and string graph models for genome assembly are NP-hard [14]. However, the success of both the de Bruijn and string graph models in practice indicates that by defining a more restricted graph model and suboptimal heuristics, it is possible to develop more efficient algorithms for genome assembly.

The question of whether the string graph or the de Bruijn graph approach is superior remains an open question, and the answer most likely depends on the sequencing technology and the specific genome characteristics. Only de Bruijn graph assemblers have demonstrated the ability to successfully assemble very short reads (<50 bp). For longer reads (>100 bp), string graph assemblers have been quite successful and have a much better track record overall [9]. The main drawback to the de Bruijn approach is the loss of information caused by decomposing a read into a path of *k*-mers. Typically, de Bruijn graph assemblers attempt to recover the information lost from breaking up the reads, and attempt to resolve small repeats using complicated read threading algorithms [23]. Using the string graph model of assembly can avoid this problem. Therefore, the quality of de Bruijn graph implementations in assembling long reads is lower than the quality of string graph implementations. Furthermore, when read lengths increase, it is relatively easy to increase minimum overlap size in string graph implementations; however, it is hard to increase *k*-mer size in de Bruijn graphs for several reasons, including computational limitations. Most of the current de Bruijn graph assemblers can only accept a *k*-mer size of up to 31 bp, with some of the latest versions going up to 127 bp [18, 35, 55]. The limited *k*-mer size in the de Bruijn graph approach has therefore limited its potential to use long reads to overcome repeats.

1.3.3 Common Stages of Genome Assembly

For the assembly of whole genomes, the graph construction and graph traversal is followed by sophisticated steps aimed at reconstructing larger contiguous segments of the genome being assembled. In most assemblers, there are a core set of steps needed [8]. In Figure 4, the common stages of genome assembly are introduced. The 4 major stages are (1) error correction, (2) graph construction, (3) graph traversal, and (4) scaffolding.

Due to high coverage, many error correction techniques in NGS data have been developed in recent years. As errors occur infrequently, the majority of reads in a specific position can be used to correct errors. This general idea has been implemented in most error-correction algorithms. Currently, error correction can be built in as a module in assemblers, such as is found with Velvet [55] and ALLPATHS-LG [35], or be isolated as stand-alone software, such as Quake [46] and HiTEC [36].

The scaffolding phase of assembly focuses on resolving repeats by linking the initial contigs to scaffolds. A scaffold is a collection of contigs linked by mate pairs. Mate pairs constrain the separation distance and the orientation of contigs containing mated reads. If the mate pair distances are long enough, they permit the assembler to link contigs across almost all repeats. Like error correction, stand-alone scaffolders are also available, such as Bambus [10], Opera [42], and SSPACE [48], allowing mate-pair information to be added to virtually any assembler.

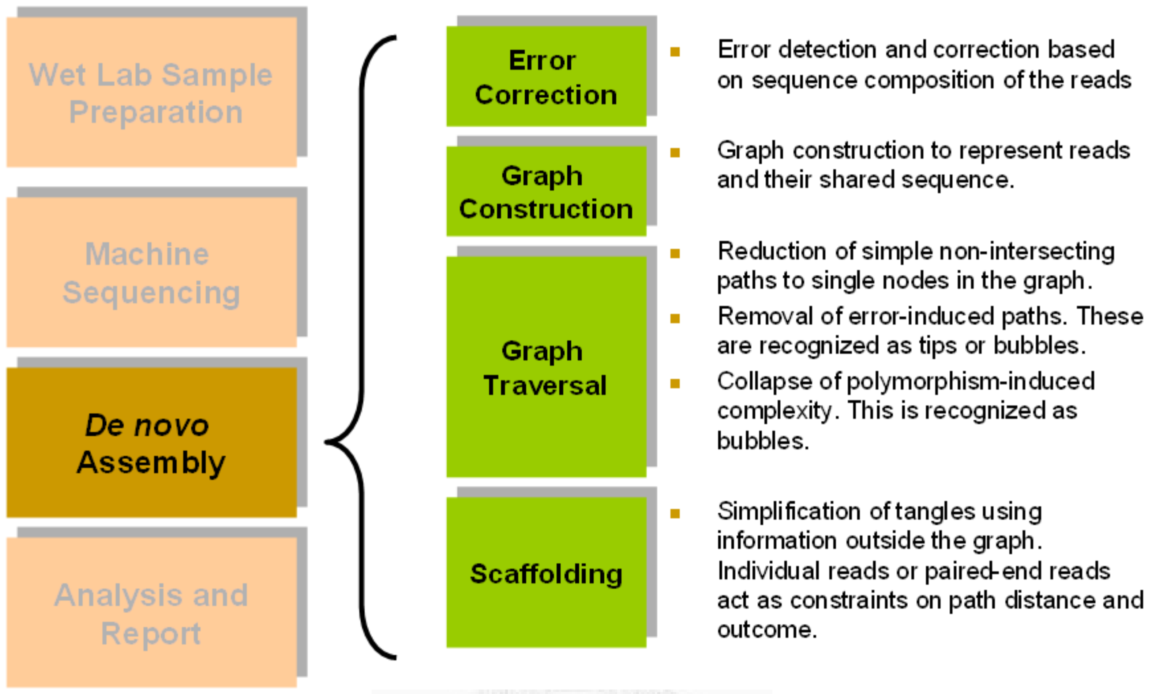

Wet Lab Sample Preparation
Machine Sequencing
De novo Assembly
Analysis and Report
Error Correction
Graph Construction
Graph Traversal
Scaffolding
Error detection and correction based on sequence composition of the reads
Graph construction to represent reads and their shared sequence.
Reduction of simple non-intersecting paths to single nodes in the graph.
Removal of error-induced paths. These are recognized as tips or bubbles.
Collapse of polymorphism-induced complexity. This is recognized as bubbles.
Simplification of tangles using information outside the graph. Individual reads or paired-end reads act as constraints on path distance and outcome.

Figure 4. Common stages of genome assembly.

1.4 Overview of this Dissertation

De novo assembly is much like a large jigsaw puzzle in that the DNA reads that shotgun sequencing produces must be assembled into a complete picture of the genome. Unlike a simple jigsaw puzzle, several factors make *de novo* assembly challenging. First, the data contain errors, some from limitations in sequencing technology and others from human mistakes during laboratory work. Second, there are repetitive sections in the genome. Similar to pieces of sky in jigsaw puzzles, reads belonging to repeats are difficult to position correctly. The magnitude of the challenge depends on the sequencing technology, because the fraction of repetitive reads depends on the read lengths themselves. In addition, with the rapid growth of sequence data, *de novo* assembly is complicated by the computational complexity of processing larger volumes of data; furthermore, non-uniform coverage of sequence data also becomes an issue for sequence assembly.

The unifying theme of this dissertation is to develop an efficient *de novo* sequence assembler. The following chapters describe possible solutions to the above challenges. As for the rapid growth of sequence data, chapter 2 proposes a MapReduce framework assembler to accommodate large datasets. Chapter 3 explores the error-correction issue of sequence assembly. Chapter 4 studies the issue of non-uniform coverage of transcriptome sequence data, and provides a feasible procedure. Finally, chapter 5 discusses the dissertation’s findings, and explains the wider implications of these findings, as well as suggesting topics for future research.

Chapter 2

2. Genome Assembly with MapReduce

2.1 Introduction

The current sequencer of the Illumina GAIIx produces reads longer than 150 bp and up to 320 Gbp data output per run. To assemble genomic data generated by such high-throughput sequencers is a big challenge even for multi-core processors with memory constraints. Most of the recent assemblers use de Bruijn graphs [4, 15, 18, 20, 35, 55] to model and manipulate the sequence reads. These assemblers have successfully assembled small genomes from short reads but have had limited success scaling to larger mammalian-sized genomes, mainly because they require memory and other computational resources that are unobtainable for most users. For larger genomes, the choice of assemblers is often limited to those that will run without crashing.

Addressing this limitation, a string graph-based *de novo* assembler, namely CloudBrush, is presented. CloudBrush uses the Hadoop/MapReduce distributed computing framework to enable *de novo* assembly of large genomes. MapReduce [38] is a distributed data processing model and execution environment that runs on large clusters of commodity machines. Hadoop is an open-source project for distributed computing. It consists of subprojects, including MapReduce, distributed file systems, and several others [34]. Hadoop is known for its scalability to handle petabyte-scale data and tolerance of hardware failures. It is becoming the *de facto* standard for large-scale data analysis, especially in “cloud computing” environments where computing resources are rented on demand, thus allowing data to be analyzed in parallel.

2.2 Methods

2.2.1 Distributed Graph Processing in MapReduce

Genome assembly has been modeled as a graph-theoretic problem. Graph models of particular interests include de Bruijn and string graphs in either directed or bidirected forms. Here, a bidirected string graph was used to model the genome assembly problem.

In a bidirected string graph, nodes represent reads, and edges represent the overlaps between reads. To model the double-stranded nature of DNA, a read can be interpreted in either forward or reverse-complement directions. For each edge that represents an ordered pair of nodes with overlapping reads, 4 possible types exist according to the directions of the 2 reads: forward-forward, reverse-reverse, forward-reverse, and reverse-forward. The type attribute is incorporated into each edge of the bidirected string graph. It is noteworthy that traversing the bidirected string graph should follow a consistent rule, i.e., the directions of in-links and out-links of the same node should be consistent. In other words, the read of a specific node can only be interpreted in a unique direction in one path of traversal.

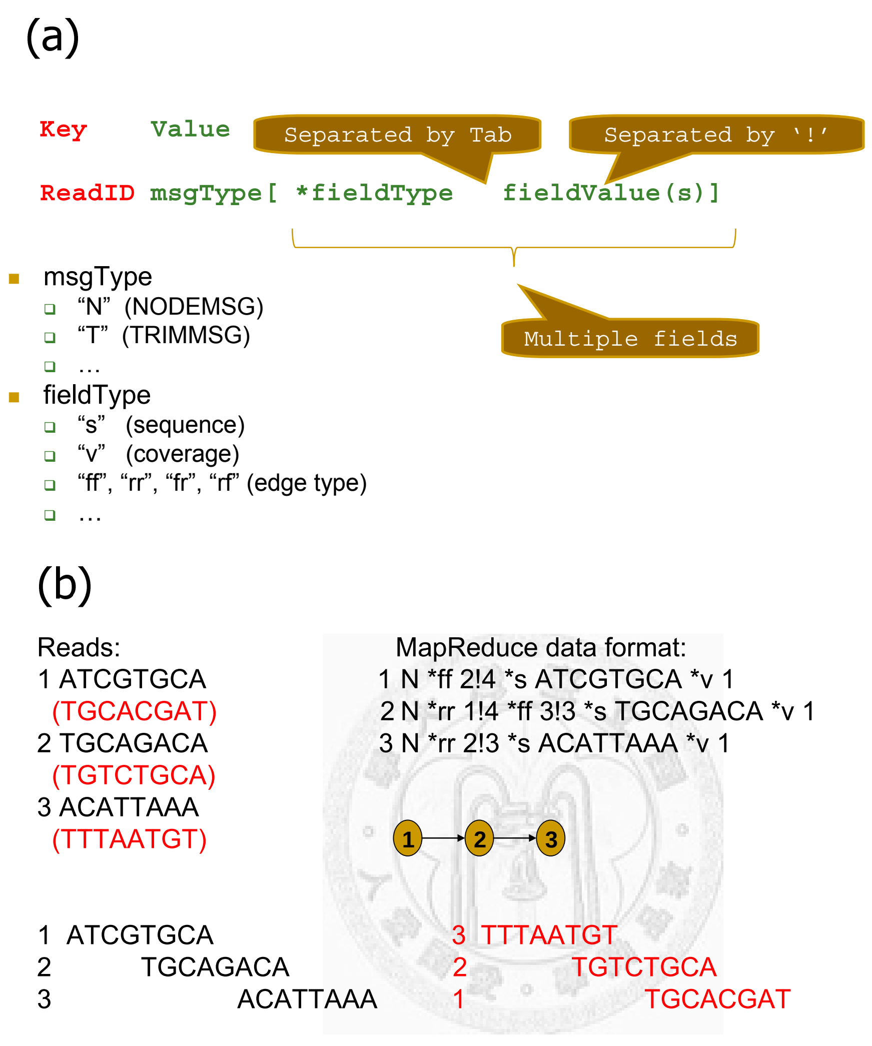


Figure 5. The data format of a bidirected string graph in MapReduce

The MapReduce framework [16] uses *key-value* pairs as the only data type to distribute the computations. To manipulate a bidirected string graph in MapReduce, a *node adjacency list* was used to represent the graph (which stores *node id* – the identifier of a node – as the *key*), and *node data structure* was used to represent the *value*. *Node data structure* contains features of the node and is represented by several types of defined messages. Each type of message may contain multiple fields, and each field has its type and value as shown in Figure 5(a). For example, the message of a node consists of fields, including sequence, coverage, and 1 of the 4 edge types. Using the Contrail assembler [15] and its data structure, the overlap and string graphs were modeled with an extension to record nodal overlap lengths. Figure 5(b) shows an example of a bidirected string graph in MapReduce. Note that each node can be represented not only its forward sequence but also its reverse-complement sequence as shown in the red sequence in Figure 5(b).

In MapReduce, a basic unit of computations is usually localized to a node's internal state and its neighbors in the graph. The results of computations on a node are output as *values*, each *keyed* with the identification of a neighbor node. Conceptually, this process can be thought of as "passing" the results of computation along out-links. In the reducer, the algorithm receives all partial results having the same destination *node id* and performs the computation. Subsequently, the data structure corresponding to each node is updated and written back to distributed file systems.

2.2.2 CloudBrush: String Graph Assembly Using MapReduce

The core of the string graph assembly algorithm can be divided into 2 parts: graph construction and graph traversal. The most critical step to construct a string graph is computing the overlap between all pairs of reads. For high-throughput sequence data, this step requires redesigning to make it computationally feasible. To find all pairs of read-read overlaps, a prefix-and-extend strategy is presented that speeds up construction of the string graph in the MapReduce framework.

As for graph traversal, it is inherently a sequential operation. However, traversing a chain of one-in one-out nodes can be formulated as a list rank problem, a well-studied parallel problem. A graph traversal problem can be formulated as a graph simplification problem, which manipulates graph structure transforming each node into the one-in one-out format without any loss of information. By incorporating the solution of list rank problems in parallel [47], it is possible to efficiently implement a parallel graph traversal algorithm in the MapReduce framework.

The pipeline of CloudBrush is summarized as follows. First, a string graph is constructed in 4 steps: retaining non-redundant reads as nodes, finding overlaps between reads, edge adjustment, and removing redundant transitive edges. Second, several techniques are used to simplify the graph, including path compression, tip removal, bubble removal, and edge adjustment. Figure 6 shows the workflow of CloudBrush.

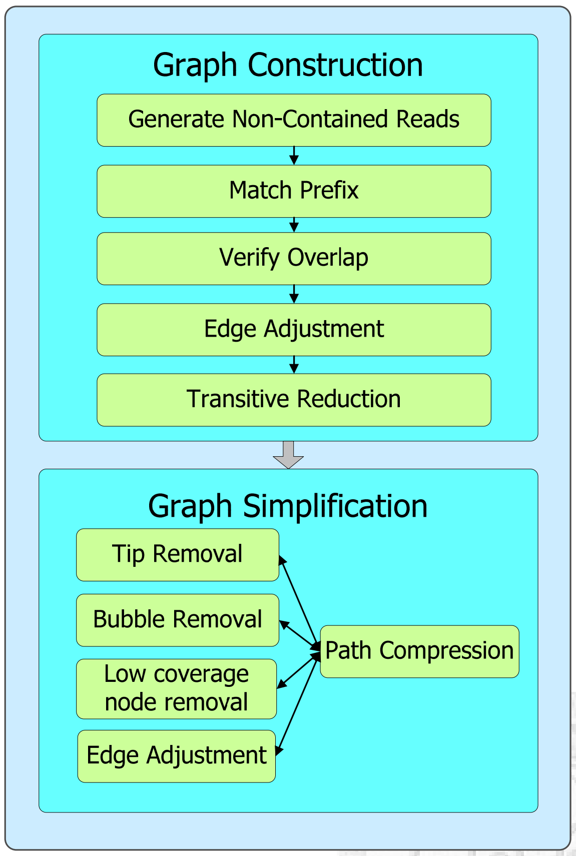


Figure 6. Workflow of CloudBrush assembler.

2.2.2.1 Graph Construction in MapReduce

Retaining non-redundant reads as nodes

Since a sequence read may have several redundant copies in the dataset due to oversampling in machine sequencing, the first step in graph construction is to merge redundant copies of the same read into a single node. This can be accomplished by constructing a prefix tree as Edena does. A distributed prefix tree for MapReduce based on Edena's prefix-tree approach was implemented [19]. In the mappers, each read of each strand was divided into a *w*-mer prefix *u* with the remaining suffix *v*, and prefix *u* was used as the *key* to distribute its ID with suffix *v* and orientation to reducers, where *w* is a parameter specified by users and each read is associated with a unique ID. A reducer, which receives information of reads with the same *key*, then builds a prefix tree of the reads' suffixes. After traversing the prefix trees, the reducer then merges identical reads and keeps a record of the number of identical copies for further processing. Examples of such processing include computing the coverage of a node at a later stage.

Finding pairwise overlaps between reads

Read-read overlaps are basic clues to connect reads to contigs. However, finding overlaps is often the most computationally intensive step in string graph-based assemblies. An algorithm was developed to run the prefix-and-extend strategy for MapReduce. The prefix-and-extend strategy is similar to the seed-and-extend strategy [28]. However, the seed, i.e., a common substring of 2 reads, must start at position 1 of 1 of the reads. The seed is denoted by *brush* in this dissertation. The strategy consists of 2 phases, i.e., the prefix and the extend phases. In the prefix phase, a pair of reads is reported if the prefix of one of the reads exactly matches a substring of the other at the given seed length. The pair is then said to have a *brush*. In the extend phase, pairs of reads with a *brush* are further examined starting from the *brush*. If the match extends to one end of the second read, then an edge containing the 2 nodes of the 2 reads is created as shown in Figure 7.

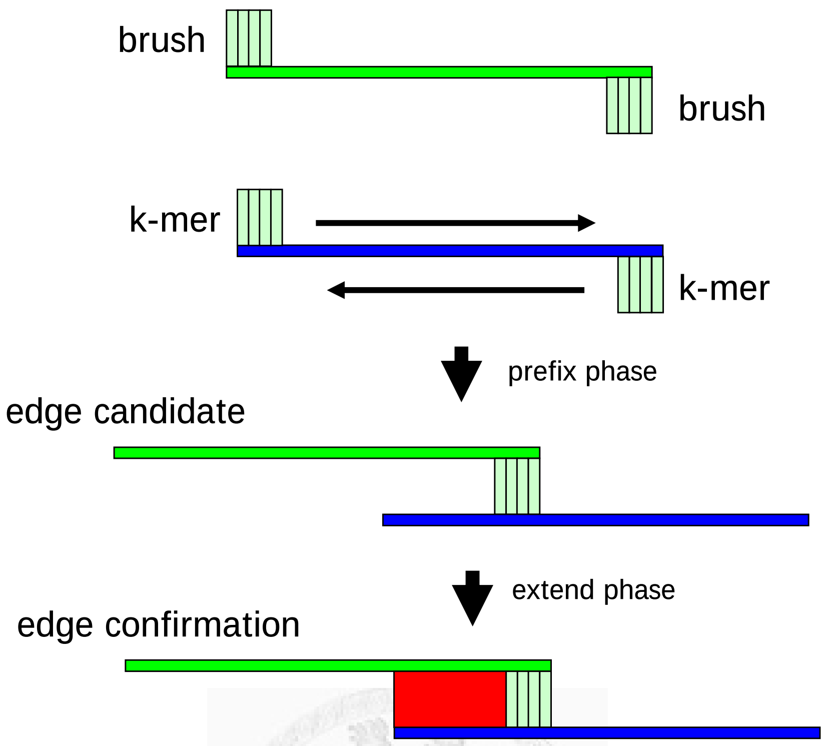


Figure 7. Illustration of the prefix-and-extend strategy.

The prefix and extend phases are implemented as 2 MapReduce procedures. In the *prefix* procedure, for each read *R*, a tuple is output for every *k*-mer subsequence of each strand of *R*, such that the *key* of the tuple is set as the *k*-mer subsequence, and the *value* of the tuple contains the node ID of *R*, as well as the orientation and the position of the *k*-mer subsequence, where *k* is a user-defined parameter. Then, each reducer receives all reads with the same *k*-mer subsequence. Following this, reads at position 1 are labeled as brush reads, and the other reads are labeled as suffix reads. For each brush read R_i, a tuple is output for each node containing R_i with the node ID as *key* and its candidate edges. In the *extend* procedure, each candidate edge is tested by extending the brush match into full alignment as shown in Figure 7. If the match extends to 1 end of the second read, then an edge containing the 2 nodes of the 2 reads is created.

Edge adjustment algorithm

After finding overlaps as edges, the edge adjustment (EA) algorithm is used on the graph structure. To perform the EA algorithm in MapReduce framework, a pass of the neighbors' edges for each node R_i is performed, such that R_i knows all the neighboring nodes in the reducer. Once a node has all its neighbors' information, the EA algorithm can easily compute the consensus sequence from the neighbors' content and thus execute the algorithm. Note that in the MapReduce framework, each node can compute its own consensus sequence in parallel. More detail regarding the EA algorithm is given in chapter 3.

Reducing transitive edges

After the EA algorithm, the graph still has unnecessary edges caused by oversampling in the sequencing. Consider 2 paths of consecutively overlapping nodes $R_1 \rightarrow R_2 \rightarrow R_3$ and $R_1 \rightarrow R_3$. The path $R_1 \rightarrow R_3$ is transitive because it spells the same sequence as $R_1 \rightarrow R_2 \rightarrow R_3$ but ignoring the middle node R_b.

To perform the transitive reduction in MapReduce framework, a pass of the neighbors' edges for each node R_i is performed such that R_i knows all the neighboring nodes in the reducer. In contrast to a de Bruijn graph, the overlap size information is attached in the edge of the bidirected string graph. Thus, it is possible to sort neighbors by overlap size and remove transitive edges in order. Using Figure 8 as an example, R_2

and R_3 are R_1's neighbor. From the viewpoint of R_1, the nodes R_2 and R_3 are checked for any overlaps between each other. Once R_3 overlaps with R_2, R_3 is treated as a transitive edge of R_1, and will be removed from R_1's adjacency list. Note that once a transitive edge is removed, its symmetric edge is also removed to maintain the structure of the bidirected string graph.

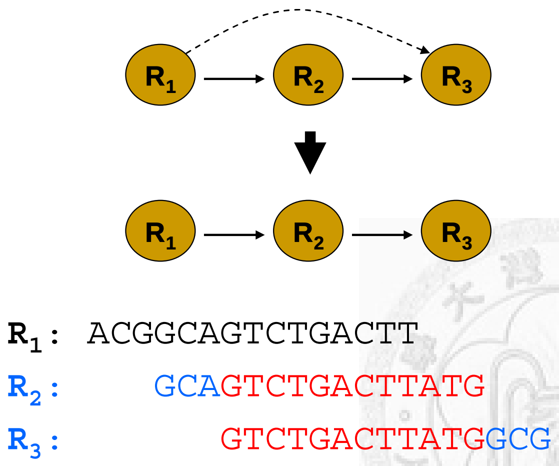


Figure 8. Illustration of transitive reduction.

2.2.2.2 Graph Simplification in MapReduce

Once the graph is constructed, several techniques can be used to simplify the graph, including path compression, tip removal, bubble removal, and low coverage node removal. This MapReduce implementation of path compression, tip removal, bubble removal, and low coverage node removal are similar to Contrail's implementation [15]. However, the novel implementation described by this dissertation differs in that it has an additional field of overlap size for the data structure of message passing between nodes tailed for the string graphs.

Path compression

Path compression is used to merge a chain of nodes (each having a single in-link and a single out-link along a specific strand direction) into a single node. Figure 9 shows an illustration of path compression.

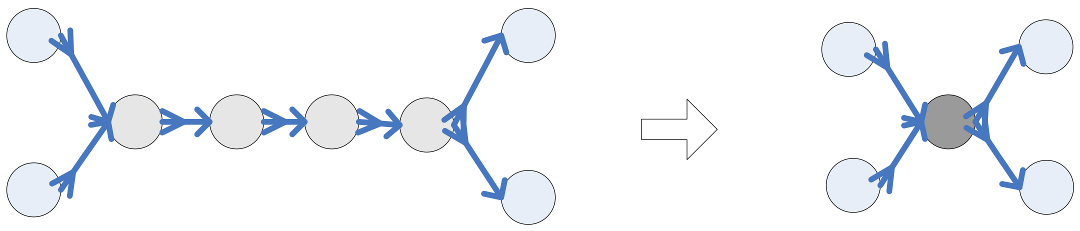


Figure 9. Illustration of path compression.

Contrail uses a randomized parallel list ranking algorithm [47] to efficiently compress simple chains of length *S* in O(*log(S)*) rounds. The algorithm randomly assigns a head or tail tag to each compressible node, which means that each node so assigned has only a single out-link for a single direction. The path compression is an iterative algorithm. For each iteration, the algorithm merges each pair of nodes so that 1 has a head tag and the other has a tail tag, and they are linked together. Until the number of compressible nodes decreases below a defined threshold (the default being 1024), MapReduce procedures assign them all to the same reducer, and merge them in 1 round.

Tip removal and bubble removal

Read errors distort the graph structure by creating tips and bubbles. Tips are generally due to errors at the end of reads. Bubbles are created by errors within reads or by nearby tips presenting spurious overlaps [55]. After path compression, tips and bubbles are easily recognized with local graph structures. Figure 10 and Figure 11 illustrate how tips and bubbles are removed. Nodes with only one out edge and lengths less than a defined threshold are treated as tips. To remove tips in the MapReduce framework, node id is output as the *key* and *node data structure* as *value* to pass graph structure to reducer and output as a *key-value* pair for each tip. The *key* contains the node ID of a tip's neighbor, and *value* is a tip's node ID. After shuffle and sort, reducers will receive *keys* corresponding to the node ID and its neighbors, which are treated as tips. This node can then remove tips in reduce stage.

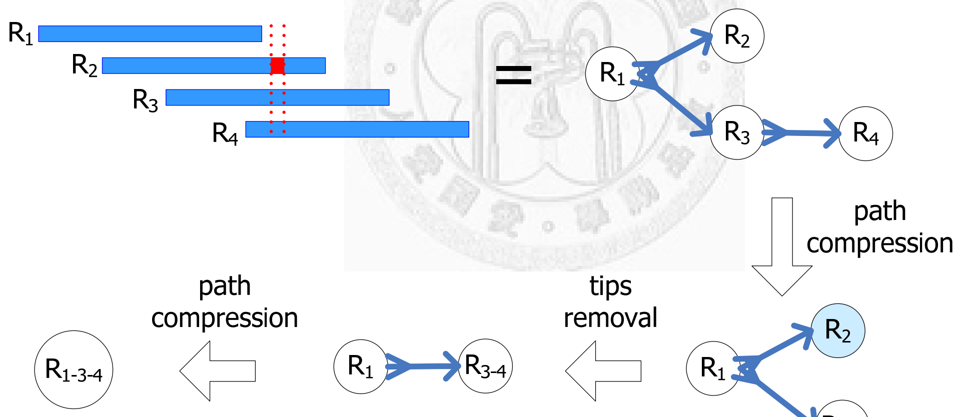


Figure 10. Illustration of tips removal.

In contrast, bubbles are shaped from a set of nodes that has only a single in-link and a single out-link, such as nodes R_2 and R_3 in Figure 11. These nodes are adjacent to the same neighbor for both the in-link and the out-link. Thus, nodes that have one in-link and one out-link are treated as potential bubbles. To ensure the consistency of a potential bubble's neighbor, one side neighbor with a large node ID is defined as major neighbor and the other side's neighbor with a small node ID as minor neighbor. To remove bubbles in the MapReduce framework, two MapReduce procedures are used. In the first procedure, the graph structure is passed to the reducer, and a *key-value* pair for each potential bubble is output. The *key* is the node ID of its major neighbor, and *value* is a potential bubble's information, which includes node feature and its minor neighbor's node ID. After shuffle and sort, reducers will receive *keys* corresponding to node ID and its neighbors, which are potential bubbles. Using Figure 11 as an example, R_2 and R_3 are potential bubbles, and R_4 is their major neighbor. Thus, the information from R_2 and R_3 will be output to the same machine with R_4 in the reducer. Since R_2 and R_3 have the same minor neighbor R_1. R_2 and R_3 are merged into a single node in the reducer when their sequences are similar. Once bubbles have been merged, the link between the major neighbor (R_4) and the merged node (R_2) is removed. Note that a second procedure is used to maintain the bidirected string graph structure's consistency. To do this, a bubble message is stored, which includes information about the minor neighbor and the node removed to the *node data structure* of the major neighbor. In the second procedure, the bubble message from the node is popped, and a *key-value* pair for each bubble message is output. The *key* is the minor neighbor's node ID, and the *value* is the removed node's ID. Thus, it is possible to remove the link between the minor neighbor and the removed node in the reducer of the second procedure. Note that tip removal and bubble removal lead to additional simple chains of nodes that can be further merged by path compression.

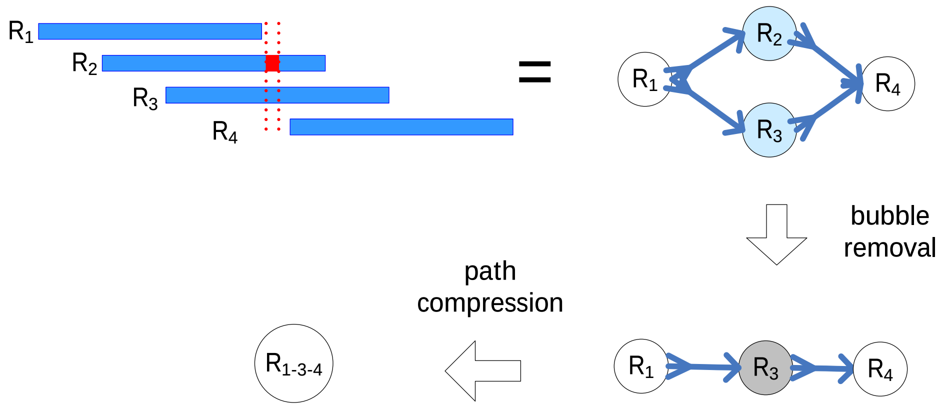


Figure 11. The illustration of bubble removal.

Edge adjustment in graph simplification

To further simplify the graph, the EA algorithm is reused in graph simplification. However, this time, it is only performed on *unique* nodes. A *unique* node is a node whose sequence is present exactly once in the genome. In general, a unique node should have a single in-link and a single out-link; however, sequencing error may cause branching on the unique node. Using the EA algorithm on the unique node can fix this problem.

The A-statistic [52] is then applied to detect the *unique* node. The formula of the A-statistic is as follows:

$$A(\Delta, r) = \Delta \times (n / G) - r \times \ln 2, \tag{2.1}$$

where Δ is contig length, r is the number of reads that comprise this contig, n is the number of total reads, and *G* is the genome size. Although the size of the genome *G* is unknown, the expected *k*-mer frequency can be computed in all reads in the MapReduce framework. The expected *k*-mer frequency is equal to $(read\_length - k + 1) \times n / G$, thus the information of *n/G* can be obtained via the expected *k*-mer frequency. The A-statistic can then be rewritten as follows:

$$A(\Delta, c, f) = \Delta \times f /(read\_length - k + 1) - (\Delta \times c / read\_length) \times \ln 2 \,, \tag{2.2}$$

where *c* is the coverage of contig and *f* is the expected *k*-mer frequency. According to the formula, the A-statistic is computed for each node in the string graphs. The EA algorithm is performed on the nodes with A-statistics >10. Note that A-statistics >10 mean that the nodes are likely located in unique regions [52]. The EA algorithm can be performed iteratively until there are no further neighbors of the *unique* node to be removed.

2.3 Results and Discussion

2.3.1 Comparison with Other Tools Using GAGE Benchmarks

To give a comprehensive comparison, GAGE [27] can be used as a benchmark to evaluate CloudBrush and compare with 8 assemblers evaluated in GAGE. Table 2 and Table 3 summarize the validation results for the 2 genomes: *Staphylococcus aureus* and *Rhodobacter sphaeroides*.

Table 2. Evaluation of *S. aureus* (genome size 2,872,915 bp).

Assembler	Num	N50 (kb)	N50 corr. (kb)	Indel >5 bp	Misjoins
ABySS	302	29.2	24.8	9	5
ALLPATHS-LG	60	96.7	66.2	12	4
Bambus2	109	50.2	16.7	164	13
MSR-CA	94	59.2	48.2	10	12
SGA	1252	4	4	2	4
SOAPdenovo	107	288.2	62.7	31	17
Velvet	162	48.4	41.5	14	14
CloudBrush	527	9.7	9.5	2	10

Table 3. Evaluation of *R. sphaeroides* (genome size 4,603,060 bp)

Assembler	Num	N50 (kb)	N50 corr. (kb)	Indel >5 bp	Misjoins
ABySS	1915	5.9	4.2	34	21
ALLPATHS-LG	204	42.5	34.4	37	6
Bambus2	177	93.2	12.8	363	5
CABOG	322	20.2	17.9	24	10
MSR-CA	395	22.1	19.1	32	10
SGA	3066	4.5	2.9	4	4
SOAPdenovo	204	131.7	14.3	406	8
Velvet	583	15.7	14.5	27	8
CloudBrush	661	12.8	12.7	10	2

As described elsewhere [27], a more aggressive assembler is prone to creating more segmental indels, as it strives to maximize the length of its contigs, while a conservative assembler minimizes errors at the expense of contig size. From Table 2 and Table 3, it can be seen that SGA's assemblies have the fewest errors of misjoints and indels >5 bp but have the shortest N50. CloudBrush produced the second fewest errors and led to longer N50. This shows that CloudBrush is a conservative assembler but is capable of assembling longer contigs.

2.3.2 Runtime Analysis

To evaluate the performance of CloudBrush's approach, CloudBrush analysis was performed on 3 different sizes of Hadoop clusters using machines leased from Hicloud (http://hicloud.hinet.net/). The 3 clusters consisted of 20, 50, and 80 nodes, respectively. Each node had 2 core CPU (roughly 2 GHz) and 4 GB of RAM. A 6 GB dataset consisting of 33.8 million read pairs was used as the benchmark to analyze the CloudBrush's runtime. CloudBrush is counted separately in 2 phases: Graph Construction and Graph Simplification. In Figure 12, it is observed that Graph Construction is CloudBrush's main bottleneck, with 20, 50, or 80 nodes. However, with the increase in the number of nodes, the computation time of Graph Construction decreases considerably, while the runtime of Graph Construction decreases slightly. These experiments show that Graph Construction tends to scale well in MapReduce.

Computation of Graph Simplification does not do so well.

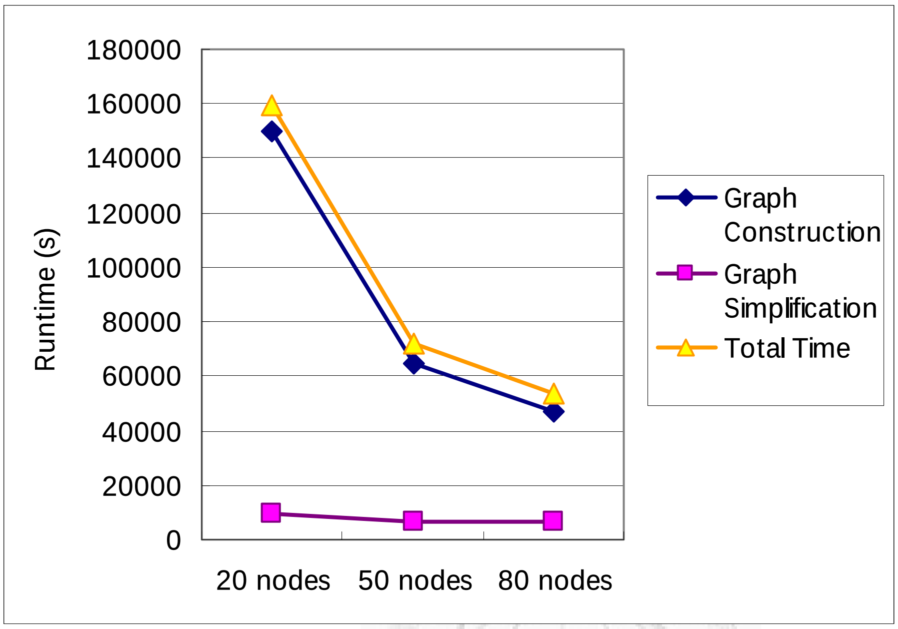


Figure 12. Runtime analysis of Dataset D3 (*C. elegans*) by CloudBrush.

Chapter 3

3. Error Correction

3.1 Introduction

Error correction is one of the most important steps in genome assembly, often taking much longer than the assembly itself. High-quality data can produce dramatic differences in the results. For example, the evaluation results of GAGE [27] show the contig sizes after error correction often increased dramatically, as much as 30-fold. This highlights the critical importance of data quality to a good assembly. There are generally 2 ways to correct for errors in genome assembly.

The first method is based on read alignment. Initially, this finds all the overlapped reads by doing multiple alignments. Reads that contain an error in a specific position can be corrected by using the majority of the reads that have this base correctly. This idea has been implemented in most error-correction algorithms [57] and can be extended to the concept of the *k*-mer frequency spectrum. The method of *k*-mer frequency spectrum counts the frequencies of all *k*-mers in the reads dataset, and then divides them into 2 types: trusted *k*-mers and untrusted *k*-mers. The process of error correction is to modify reads with minimal change that will transform all the untrusted *k*-mers into trusted *k*-mers.

The second method is based on the topological features of the graph. In using the graph-based assembly approach, sequencing errors may generate complex structures in the graph. For example, sequencing errors at the end of reads may create tips in the graph, and sequencing errors within long reads may create bubbles in the graph. By simplifying the graph (removing tips and smoothing bubbles), the number of errors can be reduced, allowing the assembly of longer contigs.

In this chapter, structural defects in a string graph as caused by sequencing error are addressed, and an EA algorithm is proposed to resolve the problem.

3.2 Methods

3.2.1 Correct Sequencing Error by String Graph

3.2.1.1 Structural Defects in a String Graph

Tips and bubbles are well-defined problems with a solution making use of the topological features of the graph as described elsewhere [15, 55]. Some errors, however, create more complex structures that cannot be readily identified from the topology of the graph. In this dissertation, these structures are referred to as “structural defects.” A well-known structural defect is the chimerical link problem. Figure 13(a) displays an example of chimerical links caused by sequencing error in string graphs.

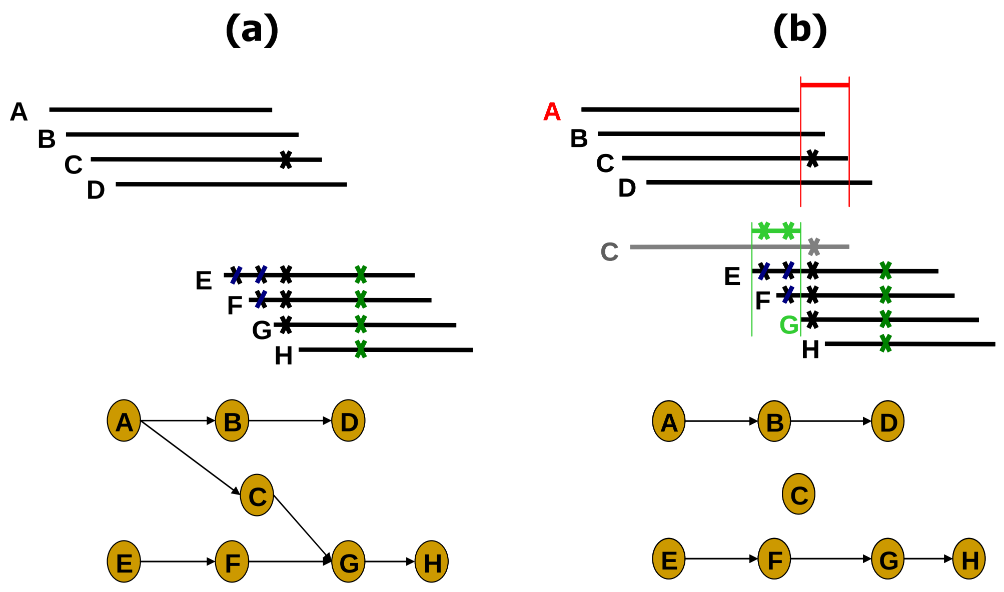


Figure 13. Chimerical link structure problem solved by edge adjustment.

In this instance, the chimerical link is caused by false overlap between node C and node G. In addition to sequencing errors, repeat regions also cause structural defects in a string graph, as seen, for example, in the well-known “frayed rope” pattern [8]. Furthermore, repeats shorter than the read lengths may also complicate processing in string graphs, for example, if a short repeat exists in reads D, E, F, I, J, L, and M, where C, D, E, and F are reads from a specific region in the genome, while I, J, L, and M are reads from another region in the same genome (Figure 14(a)). It is noteworthy that in the string graph, the edge between nodes D and L is labeled a “branch structure,” which may lead an assembly algorithm to report an erroneous contig.

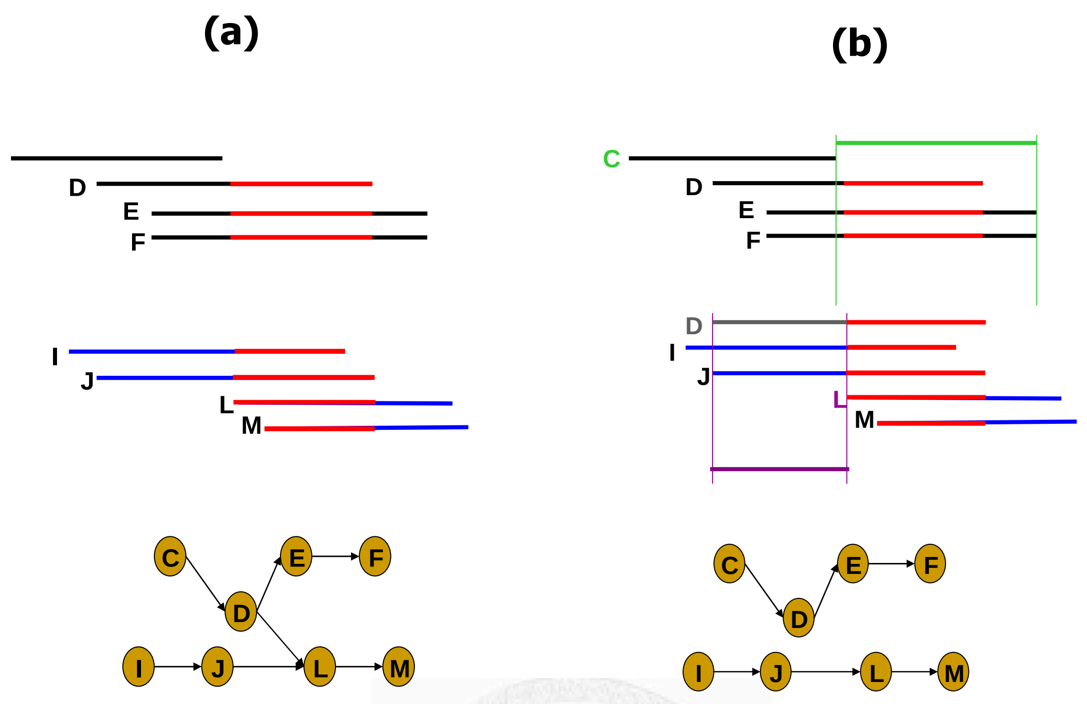


Figure 14. Branch structure problem solved by edge adjustment.

In addition to false overlaps, missing overlaps also introduce structural defects into the string graph, for example, the formation of a braid structure caused by sequencing errors appearing in continuous reads (Figure 15(a)). In this instance, 2 missing overlaps forbid the adjacent reads from being merged together; node B lost an overlap link to node C, and node D lost an overlap link to node E (Figure 15(a)). Similar to the chimerical link problem, it is challenging to use topological features of the graph to deal with braid structures.

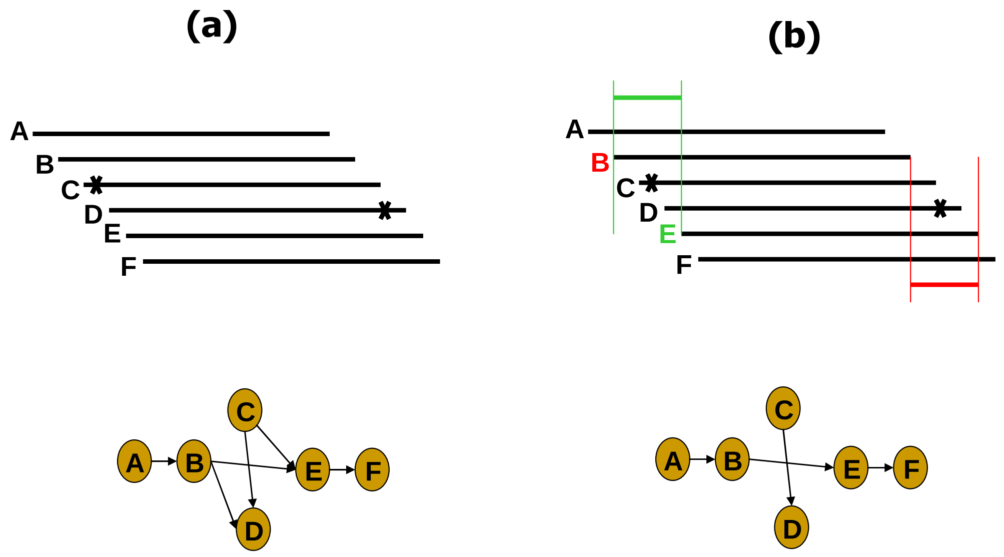


Figure 15. Braid structure problem solved by edge adjustment.

3.2.1.2 Edge Adjustment Algorithm

The EA algorithm presented in this dissertation fixes structural defects in string graphs. For a node *n* in the string graph *G*, the EA algorithm adjusts edges of *n* by examining neighbors of *n* to decide whether each neighbor has sequencing errors or not. Figure 16 shows the pseudocode of the EA algorithm in sequential versions. Figure 17 shows the pseudocode of the EA algorithm in the MapReduce version.

```
Require: overlap graph G = (N, E)
for all nodes n ∈ N do
    construct Position Weight Matrix (PWM) of the forward neighbors
    ConsensusSequence ← ComputeConsensusSequence(PWM)
    if ConsensusSequence != ∅ then
        for all forward neighbors u ∈ n.AdjacencyList do
            if !consistent(u.sequence, ConsensusSequence) then
                remove the edge between u and n
            end if
        end for
    end if
    construct Position Weight Matrix (PWM) of the reverse neighbors
    ConsensusSequence ← ComputeConsensusSequence(PWM)
    if ConsensusSequence != ∅ then
        for all reverse neighbors w ∈ n.AdjacencyList do
            if !consistent(w.sequence, ConsensusSequence) then
                remove the edge between w and n
            end if
        end for
    end if
end for
```

```
function ComputeConsensusSequenc(parameters: matrix)
    ConsensusSequence ← ∅
    for each column ci of matrix do // from i = 1 to matrix.length
        if the ratio of letter 'A' in ci > 0.6 then
            ConsensusSequence ← ConsensusSequence + "A"
        else if the ratio of letter 'T' in ci > 0.6 then
            ConsensusSequence ← ConsensusSequence + "T"
        else if the ratio of letter 'C' in ci > 0.6 then
            ConsensusSequence ← ConsensusSequence + "C"
        else if the ratio of letter 'G' in ci > 0.6 then
            ConsensusSequence ← ConsensusSequence + "G"
        else
            ConsensusSequence ← ConsensusSequence + "N"
        end if
    end for
    if (N's ratio in ConsensusSequence > 10%) then
        return ∅
    else
        return ConsensusSequence
end function
```

Figure 16. The pseudocode of the EA algorithm in sequential versions.

```
class Mapper
    method Map (nid n, node N)
      Emit (nid n, NODE_MSG N)      // Pass along graph structure
          for all nodeid m ∈ N.AdjacencyList do
                Emit (nid m, NBR_MSG N)  //Emit information to neighbor
            end for

class Reducer
    method Reduce(nid m, [MSG1 N1, MSG2 N2, …])
        M ← ∅
        for all MSGi Ni ∈ [MSG1 N1, MSG2 N2, …] do
            if IsNode(MSGi) then
                M ← Ni    // Recover graph structure
            else if IsForwardNeighbor(MSGi)
                add Ni to Forward Position Weight Matrix (FPWM)
            else if IsReverseNeighbor(MSGi)
                add Ni to Reverse Position Weight Matrix (RPWM)
            end if
        end for
        FCS ← ComputeConsensusSequence(FPWM)
        for all forward neighbors u ∈ M.AdjacencyList do
            if u.sequence is not consistent with FCS then
                remove u from M.AdjacencyList
            end if
        end for
        RCS ← ComputeConsensusSequence(RPWM)
        for all reverse neighbors w ∈ M.AdjacencyList do
            if w.sequence is not consistent with RCS then
                remove w from M.AdjacencyList
            end if
        end for
        Emit(nid m; node M)
```

Figure 17. The pseudocode of the EA algorithm in the MapReduce version.

Note that the NGS reads are of the same length. Thus, neighbors of *n* may be divided into 2 groups, i.e., forward neighbors and reverse neighbors. A forward neighbor of *n* overlaps with the suffix *n*, while a reverse neighbor of *n* overlaps with the prefix *n*. To construct node *n*'s position weight matrix (PWM) of its neighbors in 1 of the 2 directions, the reads of the neighbors to *n* are aligned. Then, the subsequences of each read ranging from the end of node *n* to the end of the second-last neighbor are used to define PWM of *n*. A consensus sequence of neighbors can be obtained by computing the PWM of the neighbors. PWM has 4 rows corresponding to A, T, C, and G, respectively. An element of PWM in column *i* is the number of occurrences of β at position *i*, where $\beta \in \{A,T,C,G\}$. The consensus sequence of these subsequences may then be defined as follows:

$$Consensus_i = \begin{Bmatrix} \beta_i \mid \beta_i / S_i > 0.6 \\ 'N' \mid \forall \beta_i / S_i \le 0.6 \end{Bmatrix}, \tag{3.1}$$

where *i* represents the position in the consensus sequence corresponding to the column position in PWM; $\beta \in \{A,T,C,G\}$; β_i is the number of occurrences of β at position *i*; and S_i is the sum of the occurrences of letters at position *i*. The letter "N" at position *i* of the consensus sequence is used if for every letter in {A,T,C,G}, $\beta_i / S_i \le 0.6$. Note that, if the percentage of "N" in the consensus sequence is greater than 10%, this consensus sequence is rejected by the EA algorithm, and all neighbors in the specific direction are retained. Otherwise, the consensus sequence is used to detect sequencing errors in each neighbors *n'* of *n* by comparing the subsequence of *n'* with the consensus sequence. The edge *(n, n')* is removed if the subsequence of *n'* is found to be inconsistent with the consensus sequence. Our approach defined that the subsequence of *n'* is consistent with the consensus sequence if every character of the subsequence is equal to the character, except character “N” on the consensus sequence at the same position. Note that for each node of the string graph, the EA algorithm generates a consensus sequence for each direction to perform the consistency check and to remove edges that are inconsistent with the consensus sequence. In an illustration of an EA algorithm, read 1 has 3 neighboring reads: 2, 3, and 4 (Figure 18). The range of the PWM exists from the end of read 1 to the end of read 3. Since read 2 has a character “A,” which is different from the first character “T” of the consensus sequence (Figure 18), the edge between read 1 and read 2 will be removed. Next, the following examples were used to illustrate the reduction of structural defects in a string graph by using the EA algorithm.

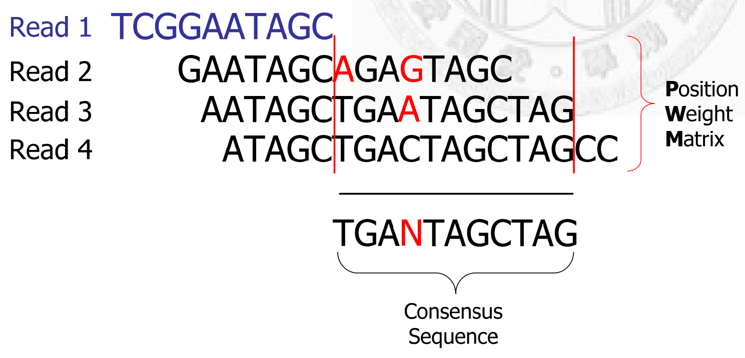


Figure 18. Illustration of the position weight matrix.

One example each of a chimerical link problem, a branch structure problem, and a braid problem, which were solved with the EA algorithm are displayed (Figures 13(b)–15(b)). To solve the chimerical link problem, the EA algorithm generates a consensus sequence for read A (shown in red) from the neighboring reads B, C, and D

(Figure 13(b)). Since read C has 1 character that is different from the consensus sequence, the overlap link between reads A and C will be removed. By contrast, the EA algorithm generates a consensus sequence for read G (shown in green) from the neighboring reads C, E, and F (Figure 13(b)). Thus, the overlap link between reads C and G will be removed in a similar manner.

To solve the branch structure problem, the EA algorithm generates a consensus sequence for read L from the neighboring reads D, I, and J (Figure 14(b)). Therefore, read D differs from the consensus sequence, which is primarily represented by reads I and J. The overlap link between reads D and L are removed.

To solve the braid structure problem, in which the errors in reads C and D complicate the graph structure, the EA algorithm removes the overlap link between reads C and E, and between reads B and D (Figure 15(b)). Thus, reads C and D are isolated from the main graph, and no braid structure exists.

3.2.2 Correct Sequencing Error by Read Stack

Although the EA algorithm can reduce the effect of sequencing errors in string graph assembly, there are some limitations of graph-based error correction. Some error patterns and datasets may cause string graph construction to lack sufficient information to create edges, such as low coverage region of data, or the error occurs in the prefix or suffix region of reads. In these cases, using the error correction before creating the graph is a better choice.

In the review of GAGE, the error-correction model of ALLPATHS-LG—called a read stack (RS) algorithm—showed a substantial improvement in sequencing assembly. To explore the reason for its effectiveness, an overview of the RS algorithm is given in this section.

The RS algorithm can be divided into 3 phases: (1) the recommendation phase, (2) the correction phase, and (3) the screening phase. In the recommendation phase, a stack of reads is built by aligning the same *k*-mer contained by those reads. Once the reads stack is built, the column of reads stack is computed to give correction recommendations. In the correction phase, it collects the information of correction recommendation and makes decisions about which base should be corrected. In the screening phase, reads that contain unique *k*-mers (after error correction) are discarded.

Two modules are used to build read stacks: PreCorrect and FindError. The FindErrors module uses a contiguous *k*-mer, whereas PreCorrect uses a split *k*-mer (a (*k* + 1)-mer with an unspecified central base). PreCorrect corrects only the central column in the stack. It is run before FindErrors and can correct some errors not found by FindErrors. Figure 19 illustrates the construction of FindError and PreCorrect. The correction recommendation was created on the basis of frequencies and quality scores of base calls in columns of the stack, such as the central column in PreCorrect and left and right columns in FindError (Figure 19). For each column, the quality scores are summed separately for each of the 4 potential calls. The base call with the highest quality score sum is declared the winner. Any other base call with no more than 1 call of quality 20 or more and quality score sum less than one-quarter that of the winner is declared a loser call. If there is a winner call, a correction recommendation is issued for each of the loser calls. The corrections are made only if all of the various correction recommendations for the same base agree with each other. Note that there is one heuristic used in the RS algorithm that makes it more accurate than other error-correction algorithms. Some base locations are marked as "confirmation" if they belong to winner calls and other bases in this column are loser calls. A confirmed base location will not be error-corrected even if it is marked in another stack as in need of correction. This conservative mechanism is designed to prevent false positives in the error correction that occur in repeat regions.

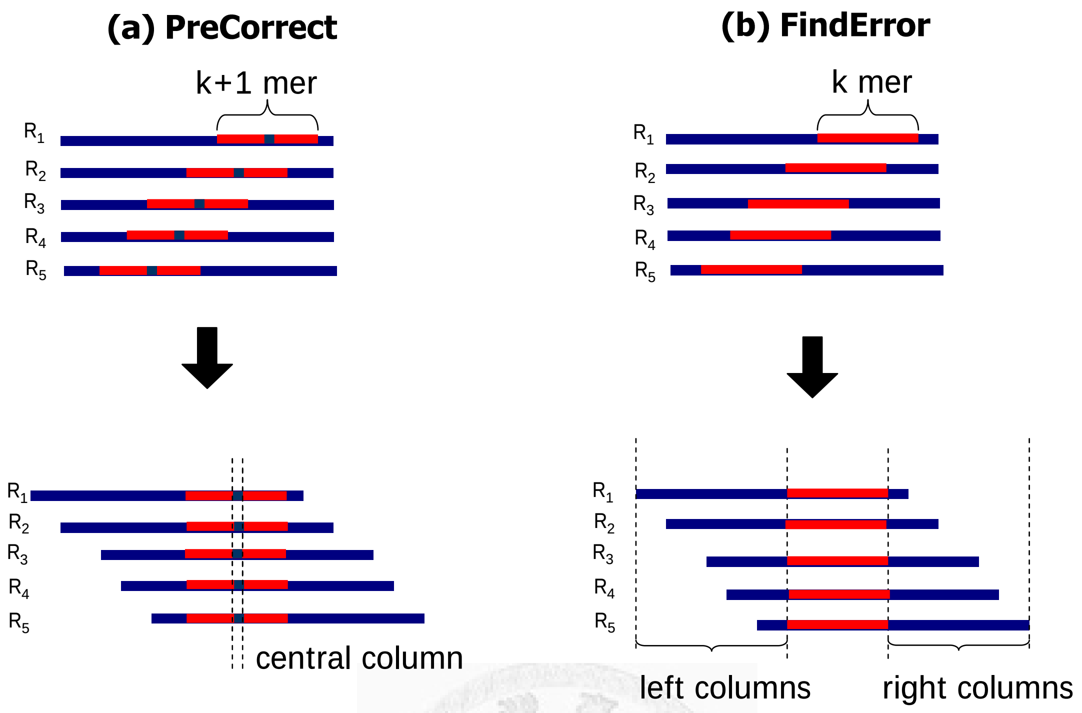


Figure 19. Illustration of read stack construction.

The RS algorithm is useful for error correction. Unfortunately, the construction of read stacks makes it very costly to apply to NGS data, and the execution time is usually longer than whole assembly time. However, the construction of read stack is easy to be paralleled. A MapReduce version of the RS algorithm as a preprocessor is provided in Appendix D.

3.3 Results and Discussion

3.3.1 Analysis of Edge Adjustment

Simulated datasets generated from the *E. coli* genome were prepared to evaluate the effectiveness of the EA algorithm. In other words, the position of each read on the target genome and thus the positions of sequencing errors on the read are also present in the dataset. The overlap graph of the dataset was subsequently constructed by creating a node to present each read and an edge between each pair of reads if a node had a sequence overlap with a size no smaller than an integer *k*. Two attributes are associated with each edge of the overlap graph from the simulated data. In the first attribute, if the positions of the 2 reads overlap with each other on the genome, then the overlapping region is designated as a *true* edge; otherwise, it is designated as a *false* edge. The second attribute is used to denote whether any sequencing error exists on the 2 reads of the edge. Therefore, it is now possible to classify edges of the overlap graph into 4 classes according to these 2 attributes. Class I denotes the subset of *true* edges without sequencing errors; class II denotes the subset of *true* edges with sequencing errors; class III denotes the subset of *false* edges with sequencing errors; and class IV denotes the subset of *false* edges without sequencing errors. It is noteworthy that class I edges are most desired to improve the quality of data for subsequent stages of sequence assembly. By contrast, class III edges are chimerical edges; class II edges contain sequencing errors; and class IV edges contain reads that intersect repeats. Edges of classes II, III, and IV may introduce errors or structural defects into the latter stages of sequence assembly. Therefore, it is the design goal of the EA algorithm to minimize the number of class II, III, and IV edges and to maximize the number of class I edges.

To test the effectiveness of the EA algorithm, 4 sets of simulated data were generated.

In the first and second sets, 36-bp reads were generated at a constant coverage of 100×, and single base errors were inserted at rates of 0.5% and 1%, respectively. In the third and fourth sets, 150-bp reads were generated at a constant coverage of 200×, and single base errors were inserted at rates of 0.5% and 1%, respectively. Table 4 shows the number of edges of the overlap graphs before and after performing the EA algorithm. Most of the edges removed by the EA algorithm were class II edges (i.e., possessing sequencing errors). It was also observed that the EA algorithm was quite effective in removing class III (chimerical) edges for the 2 150-bp datasets and satisfactory in removing the class III edges for the 2 36-bp datasets. By contrast, only about 20% of the class IV edges (i.e., those containing reads that intersect repeats) are removed by the EA algorithm.

Table 4. Edge analysis of the overlap graph before and after edge adjustment

Simulated *E. coli* Dataset	Edge Type	# of edges before Edge Adjustment	# of edges after Edge Adjustment
100 × 36 bp 0.5% error dataset	Class I	92829732	92754696 [99.92%]
	Class II	14519426	322510 [2.22%]
	Class III	252762	118542 [46.90%]
	Class IV	377856	294110 [77.84%]
100 × 36 bp 1% error dataset	Class I	76439532	76364264 [99.90%]
	Class II	24836446	749900 [3.02%]
	Class III	358432	76162 [21.25%]
	Class IV	132412	92834 [70.11%]
200 × 150 bp 0.5% error dataset	Class I	115230002	115163888 [99.94%]
	Class II	74214420	438274 [0.59%]
	Class III	1347100	51988 [3.86%]
	Class IV	403836	322746 [79.92%]
200 × 150 bp 1% error dataset	Class I	32604042	32580388 [99.93%]
	Class II	53758272	554020 [1.03%]
	Class III	1422472	57494 [4.04%]
	Class IV	256952	225124 [87.61%]

A braid index was used to provide an approximate measure of the number of braid structures in a set *S* of reads. To acquire the braid index, the overlap graph $G^o(S)$ of *S* was constructed. Following this, a simplified string graph $G^s(S)$ of S was constructed. This string graph was derived from $G^o(S)$ by removing contained reads, transitive edges, and concatenating “one-in one-out” nodes. For each node v of $G^o(S)$, its neighborhood was examined for a pair of vertices, u_1 and u_2, and an additional vertex v', such that the following properties exist: (1) (u_1, u_2) is not an edge of $G^o(S)$; (2) u_1 and u_2 form a consensus when both are aligned to v; (3) (v, v') is not an edge of $G^o(S)$; and (4) v and v' form a consensus when aligned to u_1 and u_2. The braid index is then defined as the number of tuples (v, v', u_1, u_2) satisfying the aforementioned 4 properties. Table 5 shows the braid indices of the simplified string graphs of the 4 datasets with and without the application of the EA algorithm. It was observed that datasets with a larger sequencing error have a larger braid index and may therefore possess more complicated braid structures. By contrast, the EA algorithm has also been shown to be effective in removing braid structures.

Table 5. Analysis of simplified string graphs with and without edge adjustment

Simulated data	Graph feature	Without edge adjustment	With edge adjustment
100 × 36 bp 0.5% error dataset	# of node	2502312	1572470
	# of edge	2220162	26079
	braid index	342736	750
100 × 36 bp 1% error dataset	# of node	4418943	2964253
	# of edge	4051264	46649
	braid index	873835	802
200 × 150 bp 0.5% error dataset	# of node	3839687	2680727
	# of edge	6618017	7739
	braid index	1750824	242
200 × 150 bp 1% error dataset	# of node	5085964	4245557
	# of edge	8501560	16767
	braid index	2350695	413

3.3.2 Evaluation of Assembly Accuracy

Hypothetically, a perfect assembly result produces nothing but subsquences of the reference sequences. In particular, rearrangements do not exist in any contig. To distinguish superior assembly results from those containing collapsed repetitive regions or rearrangements, a strict measurement scheme known as precision and recall was designed. The precision and recall scheme focuses on the quality of the contigs. A contig must be aligned along its whole length with a base similarity of at least 95% in order to be considered valid. The union of all the valid contig areas in the references was treated as a true positive, and the recall was defined using the following formula:

$$\text{Recall} = \frac{\text{number of true positive bases in reference}}{\text{total length of reference sequence}}. \tag{3.2}$$

Similarly, the union of all the valid contigs areas on the side of contigs was treated as a true positive in contigs, and precision was defined using the following formula:

$$\text{Precision} = \frac{\text{number of true positive bases in contigs}}{\text{total length of contigs}}. \tag{3.3}$$

Importantly, only contigs with length ≥ 200 bp were evaluated.

Three real and 2 simulated datasets were used to test CloudBrush and the other assemblers. The first real dataset was a set of short-read data from an *E. coli* library (NCBI Short Read Archive, accession no. SRX000429) consisting of 20.8 million 36-bp reads. The second real dataset was released by Illumina, which included 12 million paired-end 150-bp reads. This dataset contains sequences from a well-characterized *E. coli* strain, the K-12 MG1655 library, sequenced on an Illumina MiSeq platform. For the 2 real datasets, the first half of reads were used to evaluate assemblers, and their coverage was 81× and 197×, respectively. The 36-bp and 150-bp datasets were denoted by D1 and D2, respectively. Furthermore, *Caenorhabditis elegans* sequence reads (strain N2) were downloaded from the NCBI SRA (accession no. SRX026594). This comprised the D3 dataset and consisted of 33.8 M read pairs sequenced using the Illumina Genome Analyzer II and a constant coverage of 67×. The 2 simulated datasets were generated at random from the *E. coli* K-12 genome using 36-bp reads with 100× coverage and 1% mismatch errors, and 100-bp reads with 200× coverage and 1% mismatch errors.

Assembly was performed on these datasets using Edena [12], Velvet [4], Contrail [10], and CloudBrush assemblers. Edena was the first string graph-based assembler for data of short reads. Velvet was one of the first de Bruijn graph-based assemblers for short reads, and it is often used as a standard tool for assembling small- to medium-sized genomes. Contrail is the first de Bruijn graph-based assembler to use the MapReduce framework. Each assembler is required to set the parameter *k*, i.e., the minimum length of overlap for 2 contigs to form a longer contig. Considering the relationship between parameter *k* and coverage [20], the following were used: $k = 21$ on dataset D1 and 100× simulated data, $k = 75$ on dataset D2 and 200× simulated data, and $k = 51$ on dataset D3. Importantly, pair-end information was not used in this experiment.

Figure 20 shows the precision and recall of contigs with different length thresholds on the 2 simulated datasets of *E. coli* genome with a 1% error rate and datasets D1 and D2. CloudBrush was observed to outperform the other assemblers for the 2 simulated datasets; the other assemblers generated more mis-assembly contigs when reads become longer from 36 bp to 150 bp (Figures 20(a) and 20(b)). For the D1 and D2 datasets, CloudBrush had a similar performance for precision and recall, leading the other assemblers (Figures 20(c) and 20(d)). Longer reads and a larger error rate may generate more complex structure defects. CloudBrush may have a greater ability to handle complicated graph structures by using the EA algorithm.

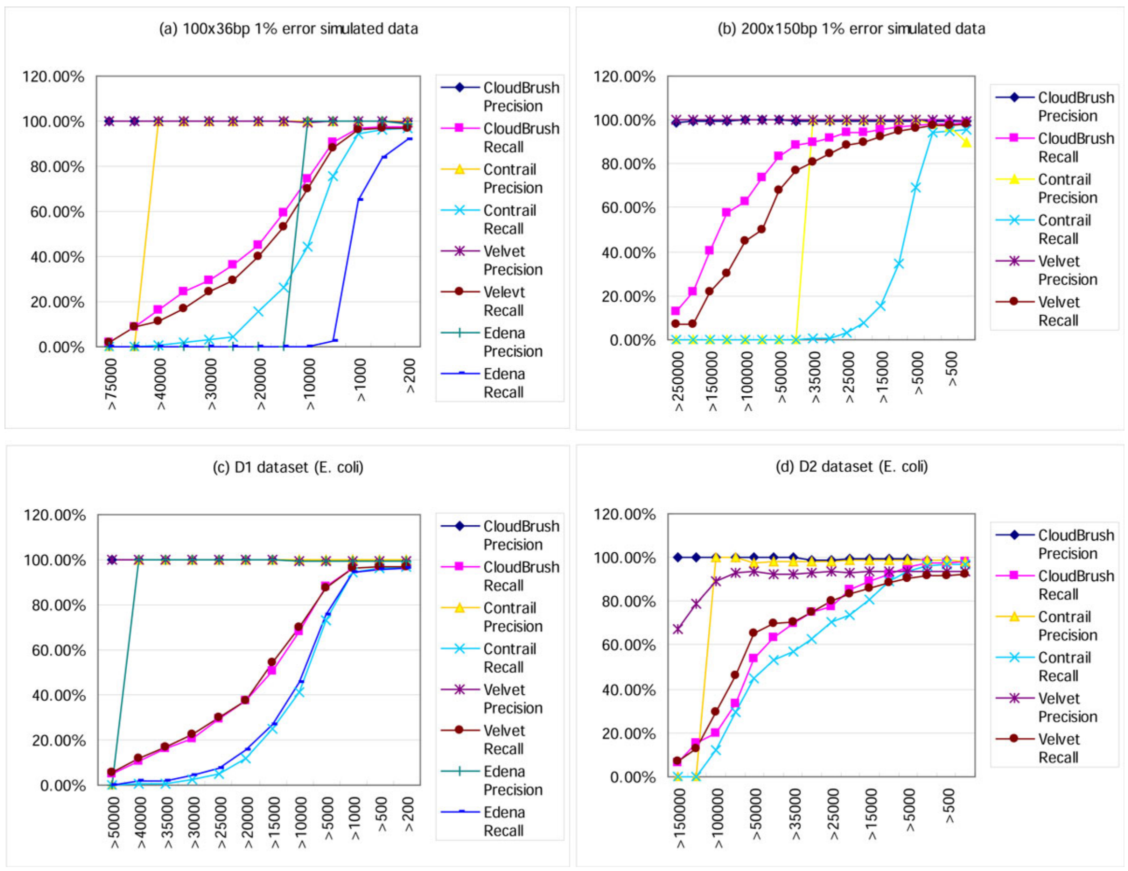


Figure 20. The variation of precision and recall with different lower bounds of length on simulated data and datasets D1 and D2.

A number of different evaluation criteria (summarized in Tables 6 and 7) were considered. It is noteworthy that CloudBrush and Contrail ran on a cluster with 150 nodes each having 2 core CPU and 4 GB of RAM; while Edena and Velvet ran on a single machine with 16 core CPU and 128 GB of RAM. Besides, Edena failed to work on datasets D2 and D3 in longer read data; therefore, no results were generated. Furthermore, precision and recall had to be computed by parsing the result of MegaBLAST [21].

Table 6. Evaluation of assemblies of the simulated dataset (100×, 36 bp, 1% error) and dataset D1 with CloudBrush, Contrail, Velvet, and Edena

Dataset	Assembler	# of contigs[1]	N50	Largest contig size	Precision	Recall	# of valid contigs[1]	# of invalid contigs[1]	Runtime (s)
100×, 36 bp, 1% error	CloudBrush	447	17907	95387	99.79%	97.51%	420	27	6218
	Contrail	906	8982	40066	99.72%	96.76%	858	48	5499
	Velvet	507	15632	100501	99.68%	96.95%	498	9	590
	Edena	4012	1436	11264	98.84%	91.85%	3868	144	2524
D1 dataset	CloudBrush	521	15149	66832	99.26%	97.10%	481	40	5555
	Contrail	930	8605	40066	99.73%	96.81%	886	44	4789
	Velvet	505	15862	73042	99.62%	96.90%	494	11	452
	Edena	889	9045	44942	99.18%	96.34%	823	66	1401

[1] Contigs with lengths >200 bp are counted.

Table 7. Evaluation of assemblies of the simulated dataset (200× 150 bp, 1% error) and datasets D2 and D3 with CloudBrush, Contrail, and Velvet

Dataset	Assembler	# of contigs[1]	N50	Largest contig size	Prec -ision	Recall	# of valid contigs[1]	# of invalid contigs[1]	Runtime (s)
200×, 150 bp, 1% error	CloudBrush	229	112531	327245	99.20%	96.00%	152	77	10616
	Contrail	2540	7554	36335	90.12%	95.92%	957	1583	15823
	Velvet	209	78642	327101	99.63%	98.10%	168	41	1317
D2 dataset	CloudBrush	361	52961	156592	98.10%	98.15%	230	131	8622
	Contrail	300	43609	124089	98.47%	96.98%	250	50	7200
	Velvet	189	71764	174184	93.60%	92.20%	164	25	927
D3 dataset	CloudBrush	37064	8880	114585	93.65%	92.41%	24603	10387	48603
	Contrail	31870	8274	105244	96.99%	90.89%	25236	6116	44619
	Velvet	23565	10847	106863	95.55%	89.01%	20187	2838	13963

[1] Contigs with lengths > 200 bp are counted.

Chapter 4

4. *De Novo* Assembly of Transcriptome Data

4.1 Introduction

Recently, research on *de novo* assembly has focused on both error removal and repeat resolution for genomic sequences, whereas only a few studies shed light on *de novo* transcriptome assembly [21, 26, 43]. However, *de novo* transcriptome assembly offers a unique opportunity to study the metabolic states of organisms [21] and provides an alternative path to study non-model organisms [5], and thus, it is a desirable and challenging approach. The main difference between genome assembly and transcriptome assembly is the variation in coverage. For example, in a genome assembly project, short reads are randomly sampled from a genome, and thus, the coverage is anticipated to be uniformly distributed on the genome. On the other hand, the distribution of short reads in a transcriptome analysis project is highly dependent on gene expression levels, and the abundance of expressed genes exhibit a power-law distribution [56]. While the coverage is related to the key parameter *k* (or *k*-mer size) for de Bruijn graph approaches [55] or minimum overlap size for string graph approaches, it seems that a single run of an assembly program would not be sufficient for *de novo* transcriptome sequencing data. In this chapter, the relationships between sequencing error rate, coverage, and parameter *k* (minimal overlap size) have been studied using simulated data. Accordingly, a transcriptome assembly procedure for *de novo* assembly of whole-transcriptome sequencing data is proposed. The primary innovation outlined in this chapter is to utilize the relationship between minimal overlap size and the coverage of sequence data. The performance and practicability of the proposed procedure is demonstrated by using a simulated transcriptome dataset of mice.

4.2 Results

4.2.1 On the Relationship Between Coverage and Optimum *k*

For genome assembly projects, it has been shown that the parameter *k* of de Bruijn graph approaches affects assembly results and is related to coverage [55]. Because the coverage of transcripts is correlated with expression levels and is thus varied, it is necessary to study the relationship between coverage and *k*s that optimize assembly. To do this, an experiment on 2 synthetic transcriptome datasets of mice was conducted, one of which was error-free and the other, with a sequencing error rate ~0.3%. The synthetic dataset of 80 million pair-ended 36-bp reads was randomly sampled from 26,332 transcripts of mice collected from the NCBI RefSeq database [39]. To mimic the varied coverage of transcriptome shotgun sequencing data, the number of reads of each transcript was proportional to the number of expressed sequence tags (ESTs) multiplied by the length of the transcript, where the EST numbers were computed according to the NCBI dbEST database. Most transcripts have low coverage, and the variation of coverage is large, extending from 1 to 4,266 (see Figure 21). Moreover, the distribution of the coverage is a power-law distribution and is similar to the experimental data of the whole transcriptome shotgun sequencing for HeLa [45]. Each transcript was separately assembled by Velvet with different parameter *k*s, and a *k* value was classified as optimum for a transcript if the transcript could be consistently aligned with 95% or more of the contig length [23]. Figures 22a and 22b show the relationship between the optimum *k* and coverage for the error-free dataset and the dataset with an error-rate of ~0.3%, respectively. Here, a red-green heat map is used to indicate the degree of optimization: a green cell represents a higher ratio of transcripts that are assembled well (achieving 95% consistent alignment with contig lengths), and a red cell represents a lower ratio of these transcripts. From these figures, 2 phenomena were observed. First, the upper left corners of Figures 22a and 22b are red, which means that optimum *k*s of transcripts with lower coverage are distributed on smaller values. In contrast, the lower right corner of Figure 22b is red, which implies that the optimum *k*s of transcripts with higher coverage are distributed on larger values when there is a sequencing error rate.

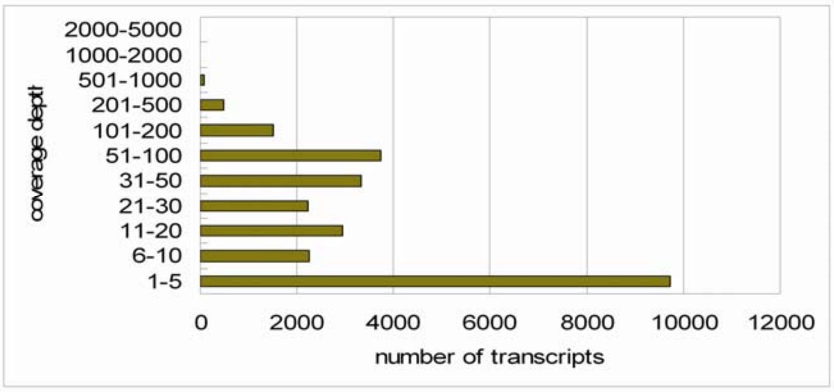


Figure 21. Histogram of the coverage (expression levels) of the 26,332 transcripts of mice.

Since the meaning of *k* can be treated as the minimum length of overlap for 2 short reads to form a longer contig, the lower coverage implies less chance to have an overlap longer than or equal to *k*. This, in turn, implies shorter contigs, and furthermore explains why a smaller *k* is more suitable for transcripts with lower coverage. The Lander-Waterman model [33] explains this first phenomenon. In this model, the expected number of contigs in a genome assembly project is $(c*G/L)e^{-(1-(k/L))c}$, where ***G*** is the genome length; ***L*** is the read length; ***c*** is the coverage; and ***k*** is the minimum length required for the detection of an overlap. Taking every transcript to be the genome in the Lander-Waterman model, this formula was then used to estimate the relationship between optimum *k*s and coverage, where a *k* value was classified as optimum for a transcript if the expected number of contigs is less than or equal to 1. Figure 22c summarizes the relationship between optimum *k*s and coverage. As can be seen, the results shown in Figure 22c are similar to the results shown in Figures 22a and 22c.

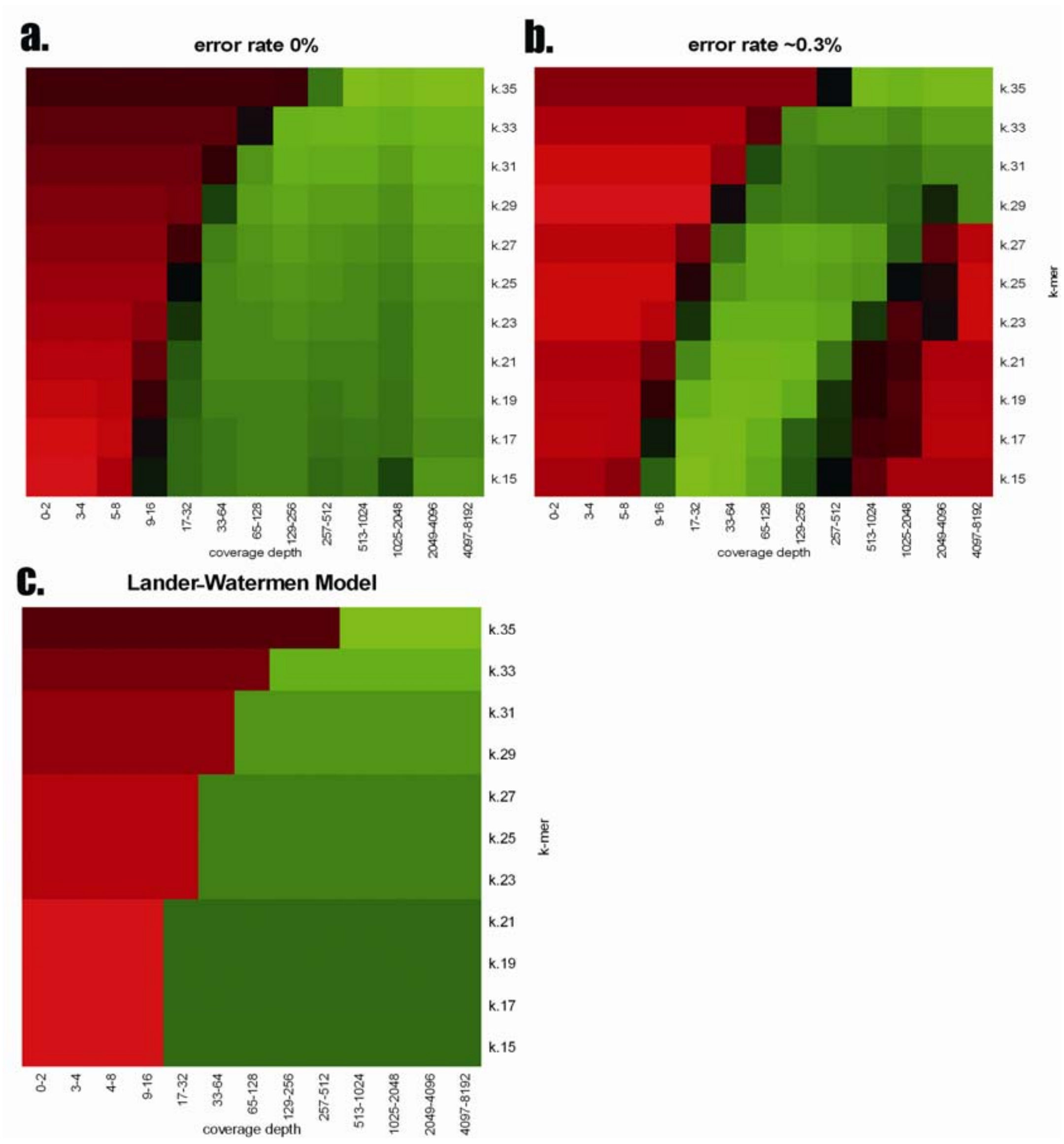


Figure 22. The relationship between optimum k and coverage. Green cells represent high ratios indicating high completeness of transcripts. Red cells represent low ratios indicating low completeness of transcripts.

For the second phenomena, it was thought that sequencing errors would result in structural defects in the underlying de Bruijn graphs. Although de Bruijn graph approaches have been designed to handle most of such undesirable cases, the sequencing error rate times and higher coverage mean more error-called bases, which implies more chances to produce structural defects that would not be resolved. In addition, a smaller *k* would give more chances to produce structural defects than would a larger *k*. Thus, using a larger *k* to assemble very high coverage data is a practical approach when there is a sequencing error rate.

4.2.2 The Effect of Sequencing Error Rate

Because a sequencing error rate of 0.3% is commonly seen in the control lane of the Illumina Solexa sequencer [20], it is possible that the sequencing error rate might increase for non-control lanes. To see the crosstalks among coverage, sequencing error rate, and optimum *k*, mouse transcripts were arbitrarily picked to generate simulated datasets with coverage 2×, 4×, 8×, 16×, … till 16384×. In addition, errors were simulated with average rates of 0%, 0.3%, 0.6%, 0.9%, ... till 2.4% for every coverage. Error rates that were slightly increased from start to end in reads were applied. For the average error rate of 0.3%, the error rate at the first nucleotide is 0.2%. This increases 0.005% for every subsequent nucleotide. Similarly, for average error rates 0.6%, 0.9%, ... till 2.4%, the error rates start with 0.5%, 0.8%, ..., and 2.3%, respectively. Figure 23 shows results of 1 simulated transcript, which demonstrate a consistent trend with Figure 22b. With the increased error rate, the range of optimum *k*s of each coverage narrows, and a positive correlation between coverage and optimum *k*s becomes noticeable. It should be noticed that for all datasets with sequencing errors, no *k* remains optimal for most tested coverage.

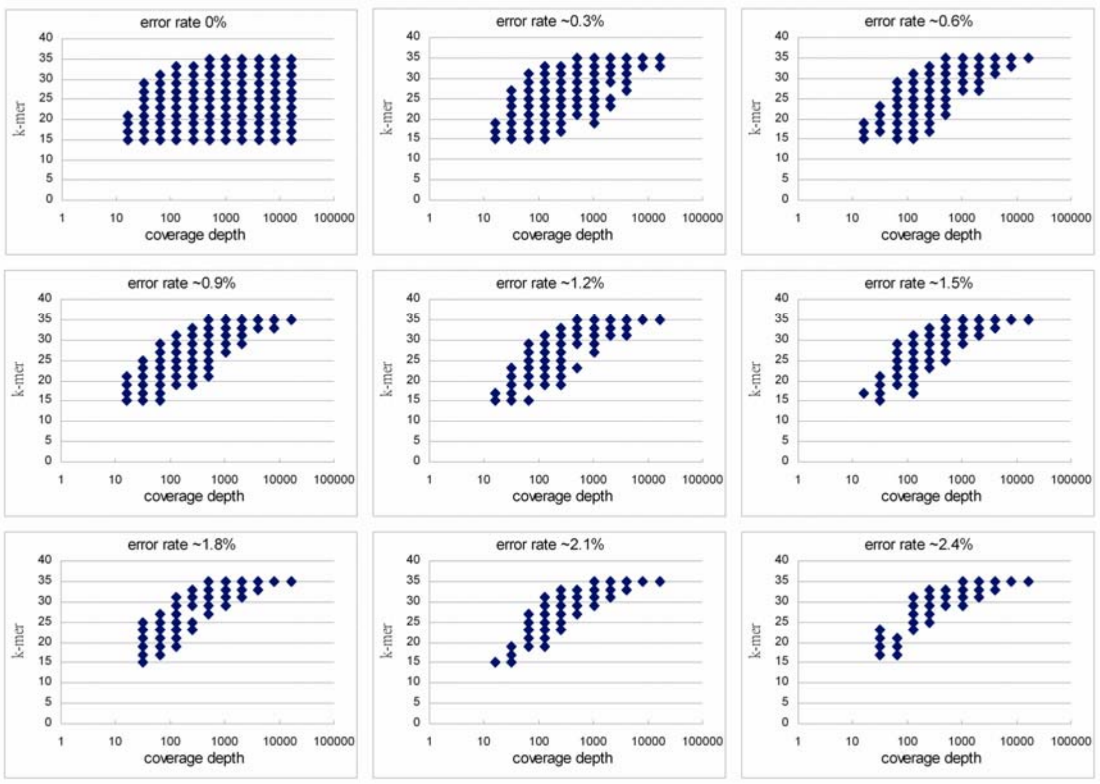


Figure 23. The relationship between optimum *k*s and coverage for one transcriptome sequence for different error rates.

4.2.3 T-CloudBrush: Multiple Overlap Size of CloudBrush

Choosing an appropriate parameter *k* for de Bruijn graph approaches is a practical issue for short-read sequence data. From the above experiments, it can be seen that coverage affects the distribution of optimum *k*s, and that no *k* is optimal for all coverage if there are sequencing errors. Thus, the issue of choosing *k* becomes tricky, especially for *de novo* transcriptome assembly, where data of different coverage are mixed in 1 sample. However, utilizing the correlation between coverage and optimum *k*s, it is possible to merge the results of different parameter *k*s together and produce a more accurate assembly of transcriptome sequencing data. Similarly, the same benefit also appears in the string graph approach.

To this end, T-CloudBrush is proposed, which integrates existing assembly programs to deal with the varying coverage of transcriptome shotgun sequencing data. The T-CloudBrush procedure is based on 2 observations: (1) A larger *overlap size* (*k*) is suitable for higher coverage data, while a smaller *overlap size* (*k*) is suitable for lower coverage data, and (2) the assembly result of an optimum *overlap size* is similar to that of an adjacent optimum *overlap size* across a range of coverage. Figure 24 gives an overview of the T-CloudBrush procedure.

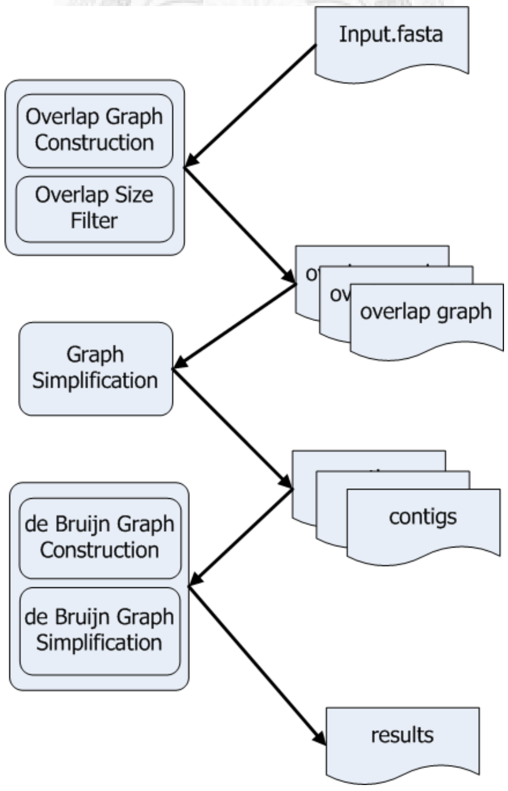

Input.fasta
Overlap Graph Construction
Overlap Size Filter
overlap graph
Graph Simplification
contigs
de Bruijn Graph Construction
de Bruijn Graph Simplification
results

Figure 24. Overview of T-CloudBrush procedure.

The T-CloudBrush procedure contains 2 stages. For the initial stage, reads were assembled using the string graph approach. In a string graph approach, an overlap size filter was added between graph construction and graph simplification. Using this mechanism, it is possible to easily construct the assembled result with different *overlap sizes* without reconstructing the overlap graph. Since the coverage affects the selection of the parameter *k* (minimal overlap size), all applicable *k*s were applied to assemble reads. Twenty-one was selected as the smallest *k* and nine-tenths of read length as the upper-bound of *k*. Then, different assembled results were obtained for the same input dataset. Due to the variation of coverage and applying multiple *k*s, some results have better performance for transcripts of higher coverage, while others are suitable for transcripts of lower coverage. For the second stage, the de Bruijn graph approach was used to assemble those results again using larger *k* (larger than read length). This was done in order to join them together, because there could be duplications and overlaps between results with different *k*s. Using the de Bruijn graph approach as a merge tool can ensure duplicated contigs are merged together. Note that using *k* larger than read length can maintain the read coherency in a de Bruijn graph [52].

4.2.4 Comparing T-CloudBrush with Existing Tools

To evaluate the performance of T-CloudBrush, the simulated dataset of the entire mouse transcriptome with a sequencing error rate of 0.3% as a benchmark was used. In the experiment, all *k*s in CloudBrush were compared with T-CloudBrush. Table 8 shows the results on the simulated data obtained from CloudBrush with *k* ranging from 21 to 31, and the results obtained from T-CloudBrush. T-CloudBrush achieved the best precision and recall measurement. Compared with CloudBrush, T-CloudBrush improved both precision and recall, reduced the number of contigs (longer than or equal to 200 bps) from 29,569 to 28,888, and extended the average size of contigs from 1349.08 to 1502.84. This shows that T-CloudBrush filled many of the gaps between highly fragmented contigs. Furthermore, the precision rate was greater than 90%, which implies that the quality of resulting contigs is reliable.

T-CloudBrush was also compared with Trinity, a state-of-the-art transcriptome assembler. Figure 25 shows the precision and recall of contigs with different thresholds of lengths on simulated mouse data using T-CloudBrush and Trinity. From this figure, it can be seen that T-CloudBrush outperformed Trinity on this dataset.

Table 8. Evaluation for assemblies of simulated mouse data with T-CloudBrush as compared to CloudBrush using different *k*-mer

k-mer parameter	21	23	25	27	29	31	T-CloudBrush
Precision	93.43%	92.17%	92.23%	91.99%	91.86%	89.02%	93.15%
Recall	20.10%	60.32%	58.18%	55.13%	50.29%	42.22%	61.88%
# of contigs ≥200 bp	31163	30115	29569	30190	30981	29300	28888
mean size (bp)	1403.89	1399.1	1349.08	1217.17	1025.34	787.6	1502.84
largest contig	86226	81929	81929	81794	47318	15910	81751

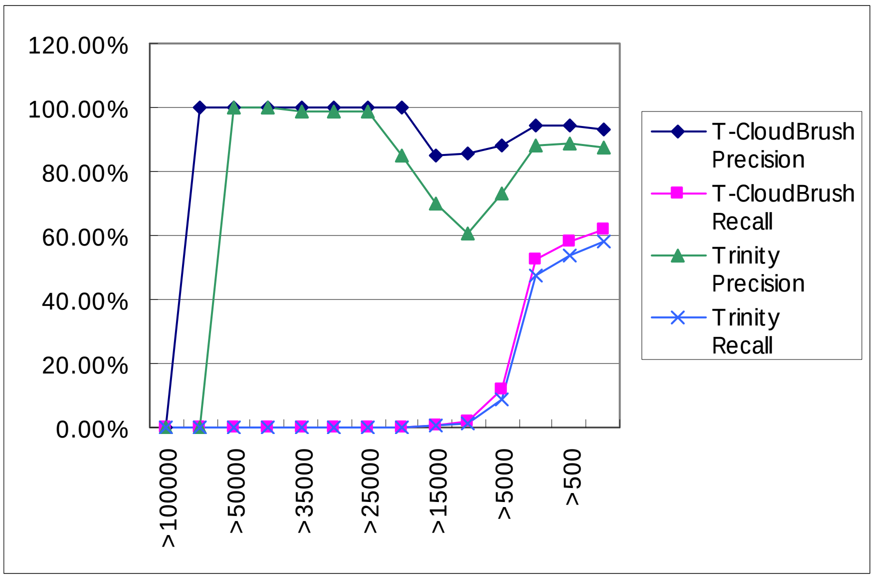


Figure 25. The variation of precision and recall with different lower bounds of length on simulated mouse data using T-CloudBrush and Trinity.

4.3 Discussion

In these experiments, it was found that the optimal overlap size of graph-based assemblers is positively correlated with the coverage of the sequence data. This phenomenon occurs because lengths of overlap between reads are highly dependent on the coverage and error rate. The lower the coverage, the lower the probability of having a longer overlap between reads. Thus, selecting a shorter minimum overlap as a criterion for assembling reads is appropriate for low coverage data. On the other hand, the higher coverage may amplify the occurrence of sequencing errors in reads. Therefore, selecting a longer minimum overlap as a criterion for assembling reads should filter out the noise for high coverage data.

As for transcriptome sequencing data, coverage is associated with expression levels. Because the expression levels exhibit a power-law distribution, choosing an appropriate overlap size for graph-based assemblers becomes a problematic issue. However, these problems can be solved by taking all of the possible overlap sizes into account. These experiments show that by merging the results of different overlap sizes obtained by graph-based assemblers, better performance for *de novo* transcriptome assembly is obtained.

5. Conclusion and Future Research

De novo assembly remains the greatest challenge for DNA sequencing, and there are specific problems for NGS, which produces high-coverage sequencing data. The problems include (1) large volumes of data, (2) sequencing error, (3) repeats, and (4) non-uniform coverage. This dissertation provides a possible solution for the abovementioned problems.

Regarding large volumes of data, a distributed assembly program based on string graphs and the MapReduce cloud computing framework is implemented. The method was evaluated against the GAGE benchmarks set by Salzberg et al [27] to compare its assembly quality with other *de novo* assembly tools. The results show that the proposed assemblies have moderate N50 and a low misassembly rate of misjoints and indels.

As for sequencing errors, the structure of string graphs in the context of high-coverage sequencing data was analyzed. Preliminary studies show that the underlying string graph used to model the intersection of reads in high-coverage data becomes too complicated for previously described assembly algorithms to handle. Thus, several types of structural defects were identified in the string graph approach. The proposed algorithms could detect the structural defects by examining neighboring reads of a specific read for sequencing errors and to adjust edges of the string graph if necessary.

To solve the non-uniform coverage problem, the relationships between read overlap size, coverage, and error rate were studied using simulated data. Based on these discovered relationships, a *de novo* transcriptome assembly procedure was developed, and its performance was demonstrated on a simulated dataset of mice.

The next target is to incorporate the scaffolding issue and mate-pair analysis into the MapReduce pipeline in order to resolve the repeat problem.

Appendix A: List of Publications

Journal Publications

Yu-Jung Chang, **Chien-Chih Chen**, Chuen-Liang Chen and Jan-Ming Ho, "A *De Novo* Next Generation Genomic Sequence Assembler Based on String Graph and MapReduce Cloud Computing Framework," BMC Genomics (2012) Volume 13 Supplement 7, S28. (Chang and Chen contributed equally to this paper)

Chien-Chih Chen, Kai-Hsiang Yang, Chuen-Liang Chen and Jan-Ming Ho, "BibPro: A Citation Parser Based on Sequence Alignment," IEEE Transactions on Knowledge and Data Engineering, volume 24, number 2, pages 236-250, January 2012.

Chien-Chih Chen, Wen-Dar Lin, Yu-Jung Chang, Chuen-Liang Chen, and Jan Ming Ho, "Enhancing *de novo* transcriptome assembly by incorporating multiple overlap sizes," ISRN Bioinformatics, 2012. (Chen and Lin contributed equally to this paper)

Conference Papers

Yu-Jung Chang, **Chien-Chih Chen**, Chuen-Liang Chen, and Jan-Ming Ho, "*De Novo* Assembly of High-Throughput Sequencing Data with Cloud Computing and New Operations on String Graphs," Proceedings 5th International Conference on Cloud Computing, IEEE CLOUD 2012, IEEE Hawaii, USA. (Chang and Chen contributed equally to this paper)

Yu-Jung Chang, **Chien-Chih Chen**, Chuen-Liang Chen and Jan-Ming Ho, "CloudBrush: A String Graph Approach of *De Novo* Assembly for High-Throughput Sequencing Data with Cloud Computing," Proceedings the 10th Asia Pacific Bioinformatics Conference, pages 1, IEEE, Melbourne Australia.

Chien-Chih Chen, Kai-Hsiang Yang and Jan-Ming Ho, "BibPro: A Citation Parser Based on Sequence Alignment Techniques," the IEEE 22nd International Conference on Advanced Information Networking and Applications (AINA), March 2008.

Appendix B: CloudBrush Manual

Introduction

CloudBrush is a *de novo* genome assembler based on the string graph and MapReduce framework.

System requirement

To use CloudBrush on a private Hadoop cluster, CloudBrush should be installed on the namenode machine of the working Hadoop cluster.

Installation

Download CloudBrush.jar

```
> wget http://cloudbrush.iis.sinica.edu.tw/download/CloudBrush.jar
```

Usage:

The first step is converting .fasta file into .sfq file. (e.g. E_coli.fastq as input file)

```
e.g.
> wget http://cloudbrush.iis.sinica.edu.tw/download/Fastq2Sfq.class
> java Fastq2Sfq E_coli.fastq E_coli.sfq
```

The second step is uploading data into hdfs.

```
e.g.
hadoop fs –put E_coli.sfq input
```

After the upload is finished, start CloudBrush by executing:

```
hadoop jar CloudBrush.jar [-asm dir] [-reads dir] [-readlen readlen] [-k k] [options]
e.g.
> hadoop jar CloudBrush.jar –reads input –asm out –k 21 –readlen 36
```

Download the results from hdfs:

```
e.g.
> hadoop fs –cat output/* > result.fasta
```

The following table describes all the properties of a CloudBrush configuration in detail:

General Options:

Parameter	**Description**	**Required**	**Default**
-asm <asmdir>	output directory	yes	-
-reads <readsdir>	input directory	yes	-
-readlen <bp>	read length	yes	-
-k <bp>	minimun overlap size to build overlap graph (half of read length is a possible choice)	yes	-
-kmercov <coverage>	average coverage of *k*-mer (used to determine unique node and repeat node)	no	30
-slots <slots>	Hadoop slots to use	no	50

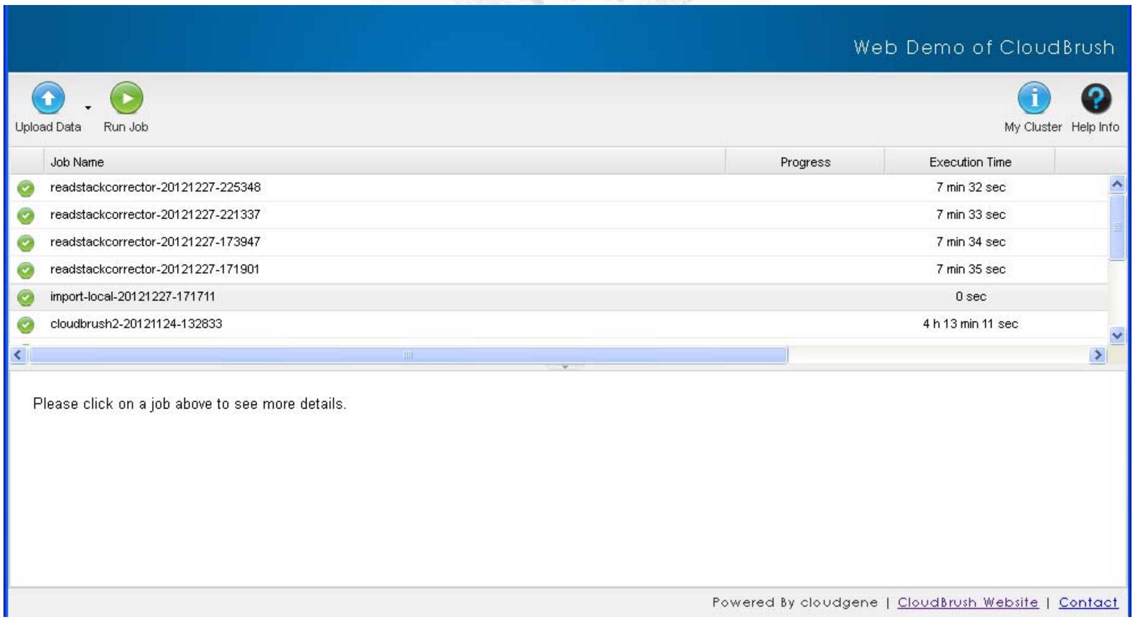
Advanced Options:

-kmerup <coverage>	threshold to build overlap graph (prevents node from having too many degrees)	no	200
-kmerlow <coveage>	threshold to build overlap graph (prevents chimerical edge in the beginning)	no	1
-maj <ratio>	majority of position weight matrix	no	0.6f
-N <ratio>	ratio of N character in consensus sequence	no	0.1f
-tiplen <len>	threshold to detect tip structure	no	10*readlen
-bubblelen <len>	threshold to detect bubble structure (max bubble length)	no	4*readlen-2*k-1
-bubbleerrate <len>	threshold to detect bubble structure (max bubble error rate)	no	0.05f
-lowcov <coverage>	threshold to detect low coverage node (coverage of node)	no	1
-lowcovlen <len>	threshold to detect low coverage node (node length)	no	2*readlen

Appendix C: CloudBrush Web Demo User Guide

Building a Hadoop cluster to run a distributed NGS analysis program, like CloudBrush, is usually not a trivial work for biologist. To demonstrate the performance of CloudBrush, we build a web demo site that provide a graphical user interface to execute CloudBrush. The web demo site is http://cloudbrush.iis.sinica.edu.tw:8082.

Figure A1 shows the main interface, which is structured as follows: the toolbar on the top shows the buttons of Run Job and Upload. The upper panel displays all executed jobs, and the lower panel shows job details like the execution time, result files, parameters, and status of the currently selected job.



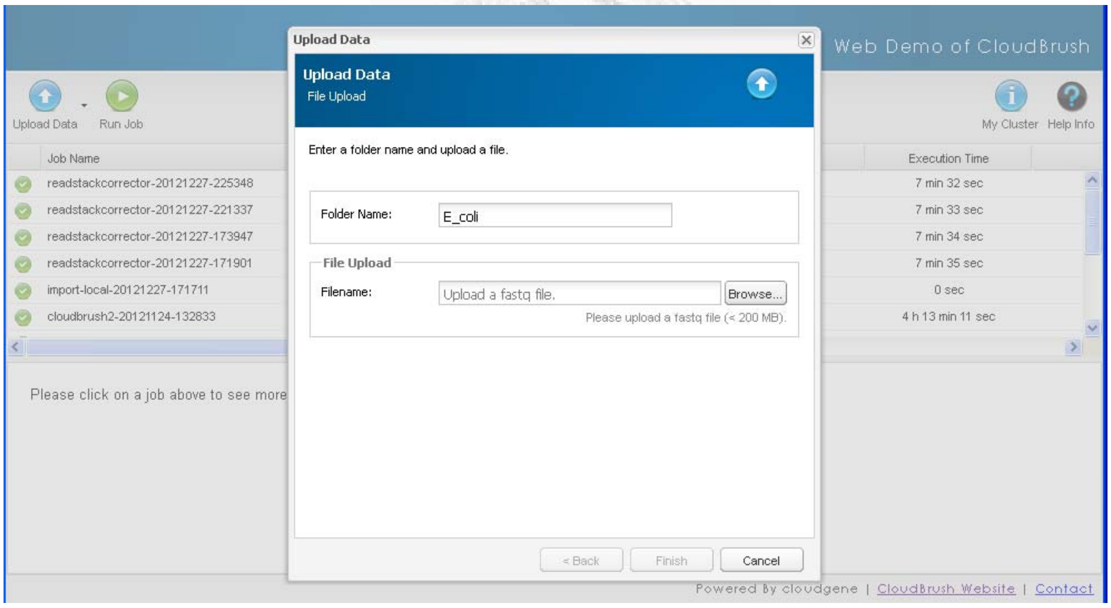
	Job Name	Progress	Execution Time
✓	readstackcorrector-20121227-225348		7 min 32 sec
✓	readstackcorrector-20121227-221337		7 min 33 sec
✓	readstackcorrector-20121227-173947		7 min 34 sec
✓	readstackcorrector-20121227-171901		7 min 35 sec
✓	import-local-20121227-171711		0 sec
✓	cloudbrush2-20121124-132833		4 h 13 min 11 sec

Please click on a job above to see more details.

Figure A1. The main interface.

Upload Data

First, you need to upload your input files (fastq format) into the HDFS Filesystem on your cluster. This can be done by clicking on the Upload Data button, whereby the source of your data has to be selected (see Figure A2). The input of CloudBrush or ReadStackCorrector is a HDFS directory; thus, you can upload multiple files in the same directory. Note that each read must have a unique name in a fastq file, and each file must be less than 200 MB.



Job Name	Execution Time
readstackcorrector-20121227-225348	7 min 32 sec
readstackcorrector-20121227-221337	7 min 33 sec
readstackcorrector-20121227-173947	7 min 34 sec
readstackcorrector-20121227-171901	7 min 35 sec
import-local-20121227-171711	0 sec
cloudbrush2-20121124-132833	4 h 13 min 11 sec

Please click on a job above to see more

Enter a folder name and upload a file.

Folder Name: E_coli

File Upload

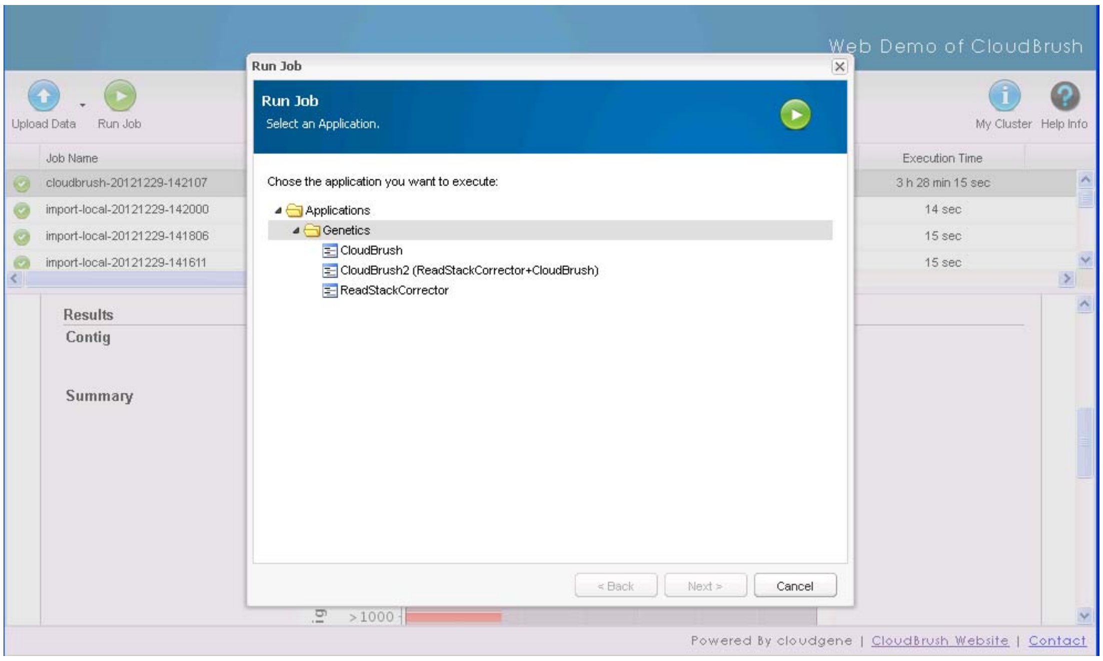
Filename: Upload a fastq file.

Please upload a fastq file (< 200 MB).

Figure A2. The upload interface.

Run Job

After the upload is finished, a job can be submitted by clicking on the button Run Job. There are 3 types of jobs that can be executed, as shown in Figure A3.



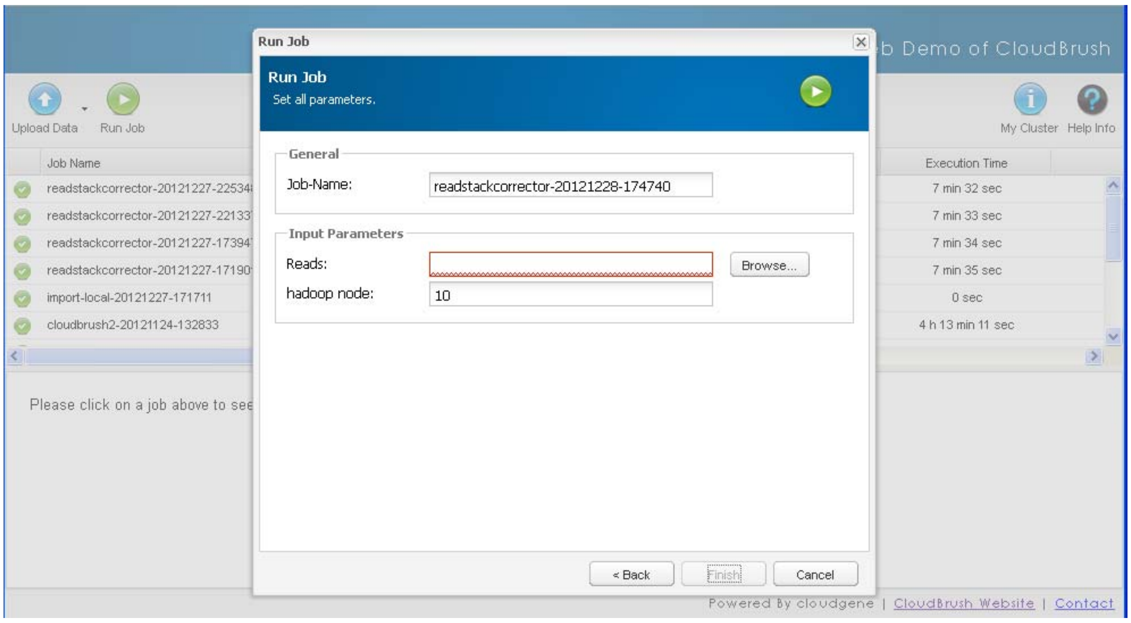
Chose the application you want to execute:

Job Name	Execution Time
cloudbrush-20121229-142107	3 h 28 min 15 sec
import-local-20121229-142000	14 sec
import-local-20121229-141806	15 sec
import-local-20121229-141611	15 sec

Figure A3. The job selection interface.

[1] ReadStackCorrector

ReadStackCorrector is an error correction tool. It can be used as a preprocessor of CloudBrush. To execute ReadStackCorrector, you should specify the input directory (browse from HDFS) as shown in Figure A4. Then, the job can be submitted by clicking on the button Finish. Once the job is finished, the resultant file can be downloaded directly via the web interface (see Figure A5). Note that the output of ReadStackCorrector can be used as the input of CloudBrush, which is under the folder **/My workspace/output/{Default Job Name}/output/**.



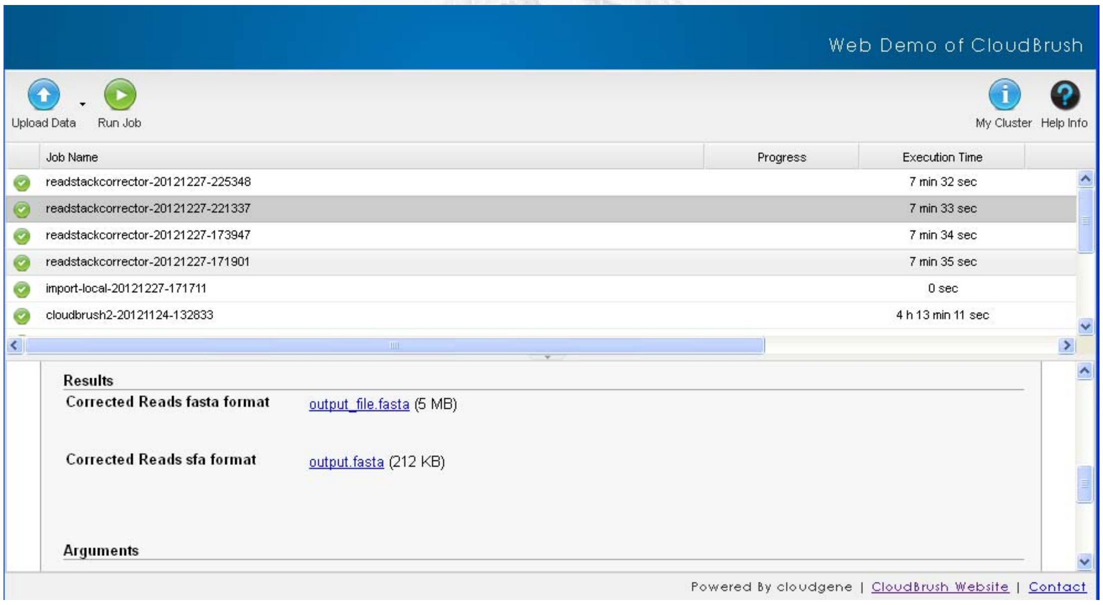
Job-Name: readstackcorrector-20121228-174740

Reads:

hadoop node: 10

Job Name	Execution Time
readstackcorrector-20121227-22534	7 min 32 sec
readstackcorrector-20121227-22133	7 min 33 sec
readstackcorrector-20121227-17394	7 min 34 sec
readstackcorrector-20121227-17190	7 min 35 sec
import-local-20121227-171711	0 sec
cloudbrush2-20121124-132833	4 h 13 min 11 sec

Figure A4. The interface of ReadStackCorrector.



Job Name	Progress	Execution Time
readstackcorrector-20121227-225348		7 min 32 sec
readstackcorrector-20121227-221337		7 min 33 sec
readstackcorrector-20121227-173947		7 min 34 sec
readstackcorrector-20121227-171901		7 min 35 sec
import-local-20121227-171711		0 sec
cloudbrush2-20121124-132833		4 h 13 min 11 sec

Results

Corrected Reads fasta format output_file.fasta (5 MB)

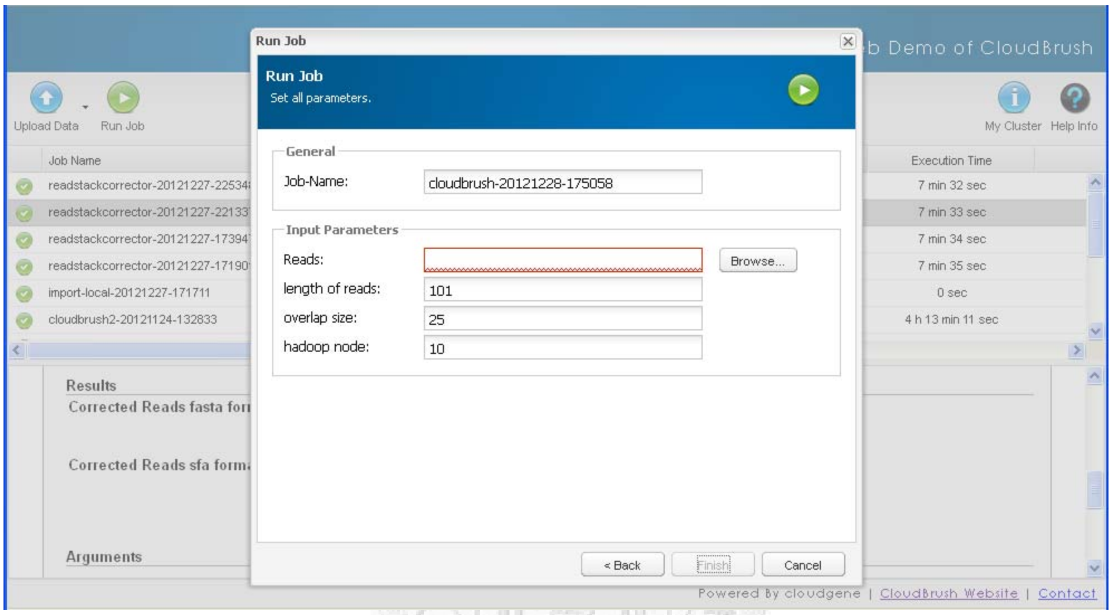
Corrected Reads sfa format output.fasta (212 KB)

Arguments

Figure A5. The result of ReadStackCorrector.

[2] CloudBrush

CloudBrush is the core program of sequence assembly. To execute CloudBrush, the input directory (uploaded in the upload data step or the output directory of **ReadStackCorrector**) and the program-specific parameters should be specified, after which the job can be submitted by clicking on the button Finish (see Figure A6). Once the job is finished, the result file can be downloaded directly via the web interface (see Figure A7).



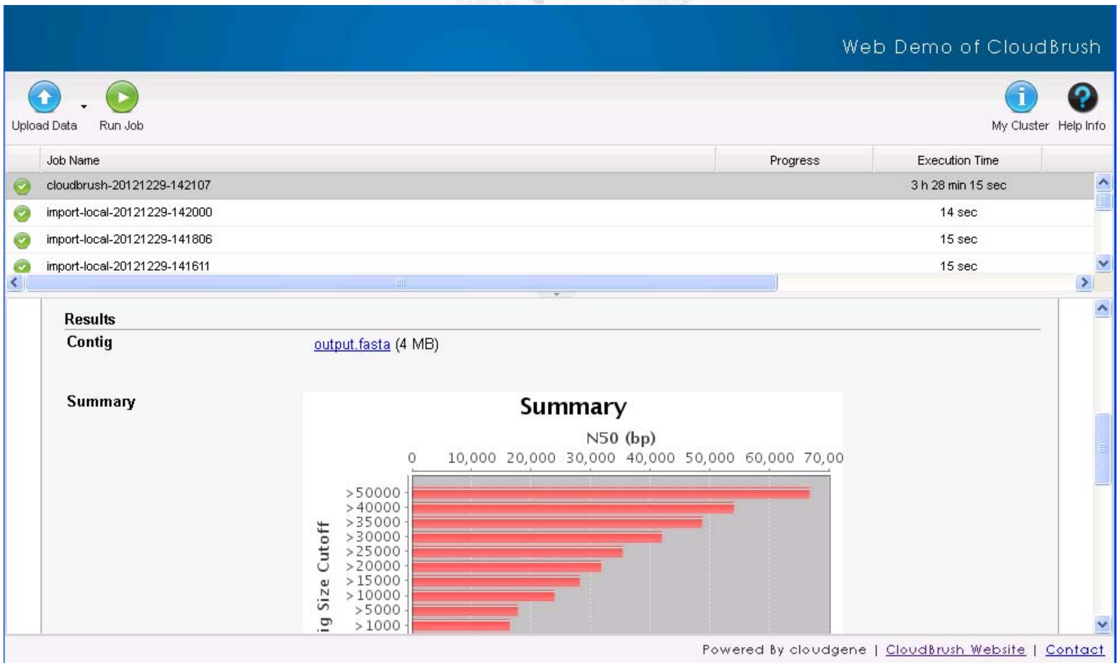
Job Name	Execution Time
readstackcorrector-20121227-22534	7 min 32 sec
readstackcorrector-20121227-22133	7 min 33 sec
readstackcorrector-20121227-17394	7 min 34 sec
readstackcorrector-20121227-17190	7 min 35 sec
import-local-20121227-171711	0 sec
cloudbrush2-20121124-132833	4 h 13 min 11 sec

Results
Corrected Reads fasta for
Corrected Reads sfa form
Arguments

General
Job-Name: cloudbrush-20121228-175058

Input Parameters
Reads:
length of reads: 101
overlap size: 25
hadoop node: 10

Figure A6. The interface of CloudBrush.



Job Name	Progress	Execution Time
cloudbrush-20121229-142107		3 h 28 min 15 sec
import-local-20121229-142000		14 sec
import-local-20121229-141806		15 sec
import-local-20121229-141611		15 sec

Results
Contig output.fasta (4 MB)
Summary

Figure A7. The results page of CloudBrush.

[3] CloudBrush2 (ReadStackCorrector + CloudBrush)

Cloudbrush2 is a pipeline to concatenate ReadStackCorrector and CloudBrush. To execute this pipeline, the input directory and the program-specific parameters should be specified, which is similar to the operation of CloudBrush.

Appendix D: Source Code

The complete source code of ReadStackCorrector and CloudBrush can be downloaded freely under an Apache License 2.0 at the following addresses:

https://github.com/ice91/ReadStackCorrector

https://github.com/ice91/CloudBrush

Bibliography

1. Zhang, Z., Schwartz, S., Wagner, L. & Miller, W. A greedy algorithm for aligning DNA sequences. *J. Comput. Biol.* **7**, 203-214 (2000).

2. Idury, R. M. & Waterman, M. S. A new algorithm for DNA sequence assembly. *J. Comput. Biol.* **2**, 291-306 (1995).

3. Myers, E. W. *et al.* A whole-genome assembly of Drosophila. *Science* **287**, 2196-2204 (2000).

4. Simpson, J. T. *et al.* ABySS: a parallel assembler for short read sequence data. *Genome Res.* **19**, 1117-1123 (2009).

5. Collins, L. J., Biggs, P. J., Voelckel, C. & Joly, S. An approach to transcriptome analysis of non-model organisms using short-read sequences. *Genome Inform* **21**, 3-14 (2008).

6. Batzoglou, S. *et al.* ARACHNE: a whole-genome shotgun assembler. *Genome Res.* **12**, 177-189 (2002).

7. de la Bastide, M. & McCombie, W. R. Assembling genomic DNA sequences with PHRAP. *Curr Protoc Bioinformatics* **Chapter 11**, Unit11.4 (2007).

8. Miller, J. R., Koren, S. & Sutton, G. Assembly algorithms for next-generation sequencing data. *Genomics* **95**, 315-327 (2010).

9. Schatz, M. C., Delcher, A. L. & Salzberg, S. L. Assembly of large genomes using

second-generation sequencing. *Genome Res.* **20**, 1165-1173 (2010).

10. Koren, S., Treangen, T. J. & Pop, M. Bambus 2: scaffolding metagenomes. *Bioinformatics* **27**, 2964-2971 (2011).

11. Pop, M. & Salzberg, S. L. Bioinformatics challenges of new sequencing technology. *Trends Genet.* **24**, 142-149 (2008).

12. Li, Z. *et al.* Comparison of the two major classes of assembly algorithms: overlap-layout-consensus and de-bruijn-graph. *Brief Funct. Genomics* **11**, 25-37 (2012).

13. Xia, Q. *et al.* Complete resequencing of 40 genomes reveals domestication events and genes in silkworm (Bombyx). *Science* **326**, 433-436 (2009).

14. Medvedev, P., Georgiou, K., Myers, G. & Brudno, M. Computability of models for sequence assembly. *Algorithms in Bioinformatics* 289-301 (2007).

15. M. C. Schatz, D. Sommer, D. R. Kelley, and M. Pop. Contrail: Assembly of large genomes using cloud computing. at <http://contrail-bio.sourceforge.net.>

16. Lin, J. & Dyer, C. Data-intensive text processing with MapReduce. *Synthesis Lectures on Human Language Technologies* **3**, 1-177 (2010).

17. Robertson, G. *et al. De novo* assembly and analysis of RNA-seq data. *Nat. Methods* **7**, 909-912 (2010).

18. Li, R. *et al. De novo* assembly of human genomes with massively parallel short read

sequencing. *Genome Res.* **20**, 265-272 (2010).

19. Hernandez, D., François, P., Farinelli, L., Osterås, M. & Schrenzel, J. *De novo* bacterial genome sequencing: millions of very short reads assembled on a desktop computer. *Genome Res.* **18**, 802-809 (2008).
20. Chaisson, M. J., Brinza, D. & Pevzner, P. A. *De novo* fragment assembly with short mate-paired reads: Does the read length matter? *Genome Res.* **19**, 336-346 (2009).
21. Birol, I. *et al. De novo* transcriptome assembly with ABySS. *Bioinformatics* **25**, 2872-2877 (2009).
22. Sanger, F., Nicklen, S. & Coulson, A. R. DNA sequencing with chain-terminating inhibitors. 1977. *Biotechnology* **24**, 104-108 (1992).
23. Simpson, J. T. & Durbin, R. Efficient *de novo* assembly of large genomes using compressed data structures. *Genome Res.* **22**, 549-556 (2012).
24. Chen, C. C., Lin, W. D., Chang, Y. J., Chen, C. L. & Ho, J. M. Enhancing *de novo* transcriptome assembly by incorporating multiple overlap sizes. *ISRN Bioinformatics* **2012**, (2012).
25. Glenn, T. C. Field guide to next-generation DNA sequencers. *Mol. Ecol. Resour.* **11**, 759-769 (2011).
26. Grabherr, M. G. *et al.* Full-length transcriptome assembly from RNA-Seq data without a reference genome. *Nat. Biotechnol.* **29**, 644-652 (2011).

27. Salzberg, S. L. *et al.* GAGE: A critical evaluation of genome assemblies and assembly algorithms. *Genome Res.* **22**, 557-567 (2012).

28. Altschul, S. F. *et al.* Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Res.* **25**, 3389-3402 (1997).

29. Pettersson, E., Lundeberg, J. & Ahmadian, A. Generations of sequencing technologies. *Genomics* **93**, 105-111 (2009).

30. Pop, M. Genome assembly reborn: Recent computational challenges. *Brief Bioinform* **10**, 354–366 (2009).

31. Paul Medvedev Genome Graphs. (2010).

32. Margulies, M. *et al.* Genome sequencing in microfabricated high-density picolitre reactors. *Nature* **437**, 376-380 (2005).

33. Lander, E. S. & Waterman, M. S. Genomic mapping by fingerprinting random clones: a mathematical analysis. *Genomics* **2**, 231-239 (1988).

34. White, T. *Hadoop: The definitive guide*. (Yahoo Press: 2010).

35. Gnerre, S. *et al.* High-quality draft assemblies of mammalian genomes from massively parallel sequence data. *Proc. Natl. Acad. Sci. USA* **108**, 1513-1518 (2011).

36. Ilie, L., Fazayeli, F. & Ilie, S. HiTEC: accurate error correction in high-throughput sequencing data. *Bioinformatics* **27**, 295-302 (2011).

37. Walter, C. Kryder's law. *Sci. Am.* **293**, 32–33 (2005).

38. Dean, J. & Ghemawat, S. MapReduce: Simplified data processing on large clusters. *Communications of the ACM* **51**, 107-113 (2008).

39. Pruitt, K. D., Tatusova, T. & Maglott, D. R. NCBI reference sequences (RefSeq): a curated non-redundant sequence database of genomes, transcripts and proteins. *Nucleic Acids Res.* **35**, D61-65 (2007).

40. Shendure, J. & Ji, H. Next-generation DNA sequencing. *Nat. Biotechnol.* **26**, 1135-1145 (2008).

41. Mardis, E. R. Next-generation DNA sequencing methods. *Annu. Rev. Genomics Hum. Genet.* **9**, 387-402 (2008).

42. Gao, S., Sung, W.-K. & Nagarajan, N. Opera: reconstructing optimal genomic scaffolds with high-throughput paired-end sequences. *J. Comput. Biol.* **18**, 1681-1691 (2011).

43. Jackson, B. G., Schnable, P. S. & Aluru, S. Parallel short sequence assembly of transcriptomes. *BMC Bioinformatics* **10 Suppl 1**, S14 (2009).

44. Nagarajan, N. & Pop, M. Parametric complexity of sequence assembly: theory and applications to next generation sequencing. *J. Comput. Biol.* **16**, 897-908 (2009).

45. Morin, R. *et al.* Profiling the HeLa S3 transcriptome using randomly primed cDNA and massively parallel short-read sequencing. *BioTechniques* **45**, 81-94 (2008).

46. Kelley, D. R., Schatz, M. C. & Salzberg, S. L. Quake: quality-aware detection and correction of sequencing errors. *Genome Biol.* **11**, R116 (2010).

47. Vishkin, U. Randomized speed-ups in parallel computation. *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing* 230-239 (1984).

48. Boetzer, M., Henkel, C. V., Jansen, H. J., Butler, D. & Pirovano, W. Scaffolding pre-assembled contigs using SSPACE. *Bioinformatics* **27**, 578-579 (2011).

49. Metzker, M. L. Sequencing technologies—the next generation. *Nat. Rev. Genet.* **11**, 31-46 (2009).

50. Stein, L. D. The case for cloud computing in genome informatics. *Genome Biol.* **11**, 207 (2010).

51. Wang, J. *et al.* The diploid genome sequence of an Asian individual. *Nature* **456**, 60-65 (2008).

52. Myers, E. W. The fragment assembly string graph. *Bioinformatics* **21 Suppl 2**, ii79-85 (2005).

53. Mardis, E. R. The impact of next-generation sequencing technology on genetics. *Trends Genet.* **24**, 133-141 (2008).

54. Mullikin, J. C. & Ning, Z. The phusion assembler. *Genome Res.* **13**, 81-90 (2003).

55. Zerbino, D. R. & Birney, E. Velvet: algorithms for *de novo* short read assembly using de Bruijn graphs. *Genome Res.* **18**, 821-829 (2008).

56. Furusawa, C. & Kaneko, K. Zipf's law in gene expression. *Phys. Rev. Lett.* **90**, 088102 (2003).

57. Yang X., Chockalingam S. P., Aluru S. A survey of error-correction methods for next-generation sequencing. *Brief Bioinform*. (2012).