12-2-2016

A Novel Tree Structure for Pattern Matching in Biological Sequences

Anas Al-okaily
anas.al-okaily@uconn.edu

A Novel Tree Structure for Pattern Matching in Biological Sequences

Anas Al-okaily, Ph.D.

University of Connecticut, 2016

This dissertation proposes solutions for the following problems: *Approximate Pattern Matching*, *Planted Motif Search*, *Genome assembly*, and *DNA compression*.

Approximate Pattern Matching is a fundamental problem in bioinformatics and information retrieval applications, involving different matching metrics such as the Hamming distance, edit distance, and wildcard matching. The input usually is a text of length n over a fixed alphabet of length Σ, a pattern of length m, and an integer k. The task is to find the set of positions of subsequences that are at no more than k Hamming distance, edit distance, or wildcards matching with P. Many algorithms and indices have been proposed to solve the problem more efficiently, but owing to the complexities of the problem in terms of space and time, most tools have adopted heuristic approaches based on suffix tree, suffix array, or Burrows Wheeler Transform to achieve a practical implementation. This dissertation mainly proposes a novel tree structure designated as *Error Tree* (ET). The structure is designed to solve approximate pattern matching problems using less space and time. For the Hamming distance and wildcard matching, the tree structure needs $O(n\frac{log_{\Sigma}^{k}n}{k!})$ words and takes

+-$O(\frac{m^k}{k!} + occ)$($O(m + \frac{log_\Sigma^k n}{k!} + occ)$ in the average case) query time for any online or offline pattern, where occ is the number of outputs. For the edit distance, the structure requires $O(2^k n \frac{log_\Sigma^k n}{k!})$ words and $O(\frac{m^k}{k!} + 3^k occ)$($O(m + \frac{log_\Sigma^k n}{k!} + 3^k occ)$ in the average case) query time for any online or offline pattern.

The second problem, Planted Motif Search, is an important and challenging problem in many biological applications that involve the search for promoters, enhancers, locus control regions, transcription factors, and more. The *(l, d)-Planted Motif Search* is one of several variations of the problem. In this problem, there are n given sequences over alphabets of size Σ, each of length m; and two given integers, l and d. The challenge is to find a motif M of length l, where each sequence includes at least an *l-mer* at a Hamming distance of no more than d from M. As the problem requires finding approximate matches, we propose an algorithm, *ET-Motif*, which employs the ET structure to resolve the problem more efficiently in terms of time and space. The algorithm can solve the PMS problem in $O(nm^2 \sum_{j=0}^{d} \binom{l}{j})$ time and $O(nml)$ space. The time bound can be further reduced by a factor of m with $O(m \sum_{j=0}^{d} \binom{l}{j} (\Sigma - 1)^j)$ space. In the case of a balanced suffix tree where the suffix tree is built for the input sequences, the problem can be solved in $O(nm^2 \sum_{j=0}^{d} \binom{log_\Sigma nm}{j})$ time and $O(nml)$ space. Similarly, the time bound can be reduced by a factor of m using $O(m \sum_{j=0}^{d} \binom{l}{j} (\Sigma - 1)^j)$ space. Moreover, the variations of the problem namely, the Edit Distance PMS and Edited PMS (Quorum) can be solved using ET-Motif but with simple modifications and an increase in the time and space bounds. For the Edit Distance PMS

problem, the time and space bounds are increased by $O(3^d)$. For the Edited PMS problem, the increase is by a factor of $(n - q + 1)$ only for time bound.

Next, current high-throughput sequencing technologies generate millions or billions of short reads (100-1000 bases) that are sequenced from a genome that is millions or billions of bases long. The third problem, De novo Genome Assembly problem, requires assembling the original genome such that it is as long and accurate as possible. Although high quality assemblies can be obtained by assembling multiple paired-end libraries with both short and long insert sizes, the latter is costly to generate. Moreover, the recent GAGE-B study showed that a remarkably good assembly quality can be obtained for bacterial genomes using state-of-the-art assemblers run on a single short-insert library with a very high coverage. The dissertation introduces a novel *hierarchical genome assembly* (HGA) method that takes further advantage of such high coverage by independently assembling disjoint subsets of reads, combining the assemblies of the subsets, and finally re-assembling the combined contigs along with the original reads. We empirically evaluated this methodology for eight leading assemblers using seven GAGE-B bacterial datasets consisting of 100bp Illumina HiSeq and 250bp Illumina MiSeq reads with coverage ranging from 100x-∼200x. The results show that HGA results in a significant improvement in the quality of the assembly for all evaluated assemblers and datasets. Nevertheless, the problem involves a major step, which is overlapping the ends of the reads together and allowing few mismatches (*i.e.,* the Approximate Matching problem). This requires computing the overlaps between the ends of all-against-all reads. The compu-

tation of such overlaps is an intensive step. Use of the aforementioned ET structure is suggested for this process in order to speed up the step.

Lastly, owing to the significant amount of DNA data being generated by Next-Generation-Sequencing machines for genomes of lengths ranging from megabases to gigabases, there is an increasing need to compress such type of data to reduce storage-space and speed up transmission-time. Huffman encoding that incorporates the characteristics of DNA sequences proves to better compress DNA data. This dissertation provides different implementations that are centered on the selection of frequent repeats so that the Huffman tree can be forced to be skewed, in addition to the construction of multiple Huffman trees when encoding. The implementations demonstrate improvements on the compression ratios for five genomes with lengths ranging from 5-50 Mbp, compared with the standard Huffman tree algorithm. Hence, the dissertation suggests an improvement on all DNA sequence compression algorithms that employ the conventional Huffman encoding. Moreover, approximate repeats can be compressed to further improve the results by encoding the Hamming or edit distance between these repeats. However, computing such distances requires additional cost in both time and space. These costs can be reduced by using the ET structure.

A Novel Tree Structure for Pattern Matching in Biological Sequences

Anas Al-okaily

B.S., Jordan University of Science and Technology, 2005

M.S., University of Technology, Sydney, 2009

A Dissertation

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Doctor of Philosophy

at the

University of Connecticut

2016

APPROVAL PAGE

Doctor of Philosophy Dissertation

A Novel Tree Structure for Pattern Matching in Biological Sequences

Presented by

Anas Al-okaily, B.S., M.S.,

Major Advisor

Chun-Hsi Huang

Associate Advisor

Sanguthevar Rajasekaran

Associate Advisor

Don Sheehy

University of Connecticut

2016

To those who are or will be suffering from diseases.

To my mother, Ameena, who always wanted me to be a physician, but I just hate to physically deal with bodies, sufferance, and death.

To my father, Ali, who always expected to see me at the top somewhere, but I just want to seek and achieve this with setting and experiencing my own goal, methodology, and standards.

ACKNOWLEDGEMENTS

I would like to greatly acknowledge and express my extremest appreciation to my parents, brothers, and sisters for their ongoing support.

I would like to acknowledge and thank greatly Prof. Ghassan Issa for his believe in my academic abilities; Petra University, and in particular Awni Shaker and Prof. Adnan Badran, for the partial financial support; Prof. Reda Ammar for his support and collaboration; Prof. Swapna Gokhale for her support and collaboration; Lois Timms-Ferrara for her faith, offering, and kind and nice mentoring; Prof. Yufeng Wu for his mentoring and collaboration; Eng. John Fikiet for his assistant and collaboration; Prof. Ion Mandoiu for his mentoring and collaboration; Prof. Sheida Nabavi, Prof. Mukul Bansal, and Prof. Thomas Peters for their support and collaboration; and my friends James Lindsay, Lance Fiondella, Saad Quader, Sherif Tolba, Jiangwen Sun, and Husam Melhem for their kind support and assistance.

A special thank and appreciation to my wife Hanadi Al Tbeishat for her support, faith, and assistance.

I would like to acknowledge and thank so much Prof. Sanguthevar Rajasekaran and Prof. Pramod Srivastava for their faith, great mentoring, and collaboration; Prof. Don Sheehy for his assistance and collaboration; and Prof. Chun-Hsi Huang for his great, kind, and professional advising which led to the completion of this dissertation.

Finally, I would like to thank all the staff in University of Connecticut for providing a great academic environment and assisting us as students in every possible way.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction: Approximate Pattern Matching in Biological Sequences

Amidst the growing trend of internet-based searches, information retrieval, data-mining applications, and bioinformatics research, there is a great need to resolve the problem of determining whether a given pattern occurs as an exact, approximate, or wildcard match in a given database. The pattern is usually small in size (such as words or sentences), while the database is much larger (such as web documents, genomes, and books).

The exact matching problem is the simplest form among pattern-matching problems, while approximate and wildcard matching are more complicated. For the exact matching problem, a tree structure was proposed by [91] and improved later by [53, 89], which led to an optimal solution of a linear structure and linear query time.

Approximate matching involves two metrics: *Hamming distance* and *edit distance* (also known as *Levenstein distance*). The Hamming distance between two strings is the minimal number of substitution operations required to transform the first string into the second one. The edit distance is the minimal number of substitution, insertion, or deletion operations required to transform the first string into the second. On the other hand, wildcard matching occurs when the pattern has a wildcard, also known as "do not care" character, represented

by Φ, that can match with any other character in the alphabet set. No linear solution (in structure size and query time) has yet been found for these matching metrics.

The dissertation proposes a novel tree structure, named as *Error Tree* (*ET*) [3], that aims to solve the approximate pattern problems and wildcard matching. The description of ET is detailed in Chapter 2. Briefly, the tree structure is based on the suffix tree with additional hash tables. Moreover, the space and time bounds for building ET are proved to be better when compared to any other structure, also its design is easily extendible. Lastly, ET contributes to better resolutions for several bioinformatics problems: alignments of Next-Generation sequencing reads, motif search, genome assembly, and compression of DNA sequences, which are addressed in this dissertation.

The first problem is the alignments of Next-Generation sequencing reads. The problem can take different forms; the most common form is the read-to-genome alignment. Given a genome of n bases of a finite set of size Σ (four for DNA sequences), a read $P = p_1p_2...p_m$, and an integer k. The goal is to find the set of positions of the subsequences in the text that are at k Hamming or edit distance or k wildcard matching with P.

The lengths of genomes range from millions of bases (such as bacterial genomes) to billions of bases (*e.g.,* mammalian genomes). The number of reads is usually in either millions or billions depending on the genome's length and the desired convergence of the sequencing. The length of the reads ranges from 100-1000 bases, where the expected error (including mismatches, insertions, or deletions) rate in each read is 2-5%.

Naively, the alignment of reads can be solved by matching them with subsequences at each position in the genome. However, as the length of genomes is long and the number of

reads is very large, this solution is not efficient in terms of time complexity. An alternative approach, which is currently the most common, is to index the genome and then align each read using that index. This requires more space (needed to build the index) but is time-efficient.

Most algorithms that introduce a practical query time require an index with a space that is polynomial to the genome length or exponential to the parameter k. Therefore, the current alignment tools use linear indices that are oriented to solve the exact matching problem such as suffix tree [89], suffix array (enhanced) [1], and FM index [28]; then, apply heuristic methods such as seed-extend in order to obtain practical space and time solutions.

The ET structure can deterministically solve the reads alignment problem by constructing a tree structure, where the needed space is efficient and the design is easily extendible. For the Hamming distance and wildcard matching, the cost of constructing ET is $O(n\frac{log_{\Sigma}^{k}n}{k!})$ words, with a query time of $O(\frac{m^k}{k!} + occ)(O(m + \frac{log_{\Sigma}^{k}n}{k!} + occ)$ in the average case) for any online or offline pattern, where occ is the number of occurrences to be reported. For the edit distance, a construction space of $O(2^k n\frac{log_{\Sigma}^{k}n}{k!})$ words is required, where the query time is $O(\frac{m^k}{k!} + 3^k occ)$ ($O(m + \frac{log_{\Sigma}^{k}n}{k!} + 3^k occ$) in the average case) for any online or offline pattern.

Although reducing the construction space of the ET structure is left for future work, a practical alignment tool using the ET structure for large genomes is still realizable. This can be achieved by utilizing the fact that the average error rate in a read 100 bases long is 2% ($k = 2$). Thereby, the alignments for a Hamming distance of two can be computed by constructing the ET for $k = 1$ (costing $O(nlog_{\Sigma}n)$ words); then, the alignments of the 1-neighbors of a read are determined. The *1*-neighbors of a read r that is of length m are

the sequences that are at a Hamming distance of 1 from r, *i.e.,* $\sum_{j=0}^{1}(\Sigma-1)\binom{m}{j} = 3m$. Therefore, the query time will be $O(m^2 + occ)$ ($O(m + log_{\Sigma}^{2}n + occ$ in the average case).

The second application that is addressed in this dissertation is the motif search problem. This is an important and challenging problem in biology and bioinformatics. A motif is a substring that occurs in each sequence of a set of DNA or protein sequences, where the substrings can differ in some number of mutations. The problem is also known in the literature as the *closest substring problem.* In terms of biological applications, finding such a motif can lead to finding promoters, enhancers, locus control regions, transcription factors, and more.

The problem involves several formulations, the most prominent one is the *Planted Motif Search (PMS).* Given n sequences of length m each, l as the length of the motif of interest, and d as the number of mutations allowed; the *(l, d)*-PMS problem is to find a motif M that has at least one substring of length l (*l-mer*) in each sequence, where the *Hamming distance* between M and each such substring is less than or equal to d.

The solution to the *PMS* problem can be *exact* or *approximate.* An exact solution guarantees the discovery of all motifs in the input sequences, whereas the approximate solution does not. For both types of solutions, there are two main approaches: pattern-driven and sample-driven. The motif using the pattern-driven approach can be found by first selecting a candidate motif (selected from the input sequences), and then searching for its validity. By contrast, the solution using the sample-driven approach is to find the positions of the motif's instances in the input sequences. A motif's instance is defined as the sequence that is at a Hamming distance of a given value of k from the motif.

Clearly, the main complexity of the PMS problem is computing the approximate pattern-matching. For this, an algorithm *ET-Motif* [6], described in Chapter 3, employs the ET structure for a better resolution of the *PMS* problem. The algorithm takes $O(nm^2 \sum_{j=0}^{d} \binom{l}{j})$ time and $O(nml)$ space. Moreover, the time bound can be further reduced by a factor of m with $O(m \sum_{j=0}^{d} \binom{l}{j}(\Sigma - 1)^j)$ space. In case the suffix tree built for the input sequences is balanced, the problem can be solved in $O(nm^2 \sum_{j=0}^{d} \binom{log_{\Sigma} nm}{j})$ time and $O(nml)$ space. Moreover, the time bound can be further reduced by a factor of m but using $O(m \sum_{j=0}^{d} \binom{l}{j}(\Sigma - 1)^j)$ space.

The third problem, the *de novo* genome assembly problem, is one of the fundamental challenges in bioinformatics. Beginning from short reads that are sequenced for a genome, the task is to assemble the genome such that it is as long as possible using these reads. On its face, the problem appears simple, but it is actually quite complex owing to sequencing errors, repeats in the genomes that are longer than the reads, and non-uniform sequencing coverage.

Many approaches have been proposed to solve the problem. Some of the main approaches are: greedy algorithms, *overlap graph*, and *de Bruijn graph*. Furthermore, the *divide-and-conquer* method can be employed to produce better assembly results [4]. Chapter 4 describes an implementation of this method and improvements in the results of assemblies for bacterial genomes.

All of these approaches need to compute the matching between each pair of reads, whether as an end-to-end alignment in order to disregard duplicate reads, or matching the ends of the reads to find an overlap among them. Due to the sequencing errors in the reads,

the matching process must be approximate. Finding the approximate matching between the ends of the reads is an extensive process and requires matching of all-against-all reads. Further, it is more complex to compute the approximate matching than the exact one. As the ET structure contributes in resolving the approximate matching problem, it can be used to reduce the costs of such computations. Since the overlap graph is a more memory-efficient solution than the de Bruijn graph [67], this dissertation recommends the use of the former with the ET structure for future research.

Lastly, this dissertation addresses the need for compressing the increasing amount of DNA data being generated by Next-Generation-Sequencing machines. Data compression is a classical problem that has been comprehensively discussed in the literature. The compression of data allows a reduction in costs associated with the storage and transmission bandwidth. There are two types of compression: *lossy* and *lossless*. Lossy compression does not guarantee an exact restoration of the original data due to unnecessary content or live transmission, whereas lossless compression does.

Many specialized compression algorithms that target DNA sequences have been proposed recently due to particular characteristics of DNA sequences. The first characteristic is the fact that DNA sequences contain only four letters, which is a relatively small alphabet size compared to the English alphabet for instance. Secondly, DNA sequences contain a relatively high percentage of repeats, which in turn carry significant biological implications.

Several compression techniques have been applied by DNA-specialized compressors; one such example is Huffman encoding, which is an old compression method but one of the most common algorithms in data compression. The algorithm functions by encoding high-

frequency symbols with shorter bit-codes and low-frequency symbols with longer ones. A specialized tree structure, called Huffman tree, is built for this purpose.

Due to the uniform distribution of the bases in DNA sequences, Huffman encoding is not the most efficient method to compress such sequences. However, a new method, called *MUHTL* [5], shows how Huffman encoding can be better employed to improve the compression of DNA sequences. Implementation of the MUHTL method can improve the compression ratios of several small-to-medium genomes. The central idea of this method is to select frequent repeats so as to force a skewed Huffman tree. It also aims to construct multiple Huffman trees when encoding. The MUHTL method is described in Chapter 5.

Compression of *approximate repeats* has been reported to improve the compression results of DNA sequences [17, 16]. These repeats can be found by computing the approximate pattern-matching between the subsequences in the genome. As the ET algorithm can contribute to such a computation, the ET algorithm is intended to be included in the future version of the *MUHTL* method in order to find those repeats and eventually improve the compression results.

Chapter 2

Error Tree: A Tree Structure for Hamming and Edit Distances and Wildcard Matching

This chapter considers the following problems. Broadly speaking, the main focus is on problem 2.

Problem 1: Dictionary Indexing

Inputs: A dictionary of N strings, where each string s is of length m symbols of a finite set of size Σ and $\sum_{i=1}^{N} |s_i| = n$, a pattern $P = p_1 p_2 ... p_m$, and an integer k.

Outputs: The set of strings in the dictionary that are at a Hamming distance of less than or equal to k (*k-Hamming distance*), edit distance of less than or equal to k (*k-edit distance*), or less than or equal to k wildcard matching (*k-wildcard matching*) with P.

Problem 2: Text Indexing

Inputs: A text of n symbols of a finite set of size Σ, a pattern $P = p_1 p_2 ... p_m$, and an integer k.

Outputs: The set of positions of the subsequences in the text that are at k-Hamming distance, k-edit distance, or k-wildcard matching with P.

Note that problem 1 is a simple case of problem 2. To simplify the presentation, we assumed that all the strings in the dictionary are of the same length.

A novel tree structure, *Error Tree* (ET), is proposed to solve the aforementioned problems. The design is to construct a tree structure that is based on the suffix tree with additional hash tables. Then, the pattern can be queried using the tree structure. The space bound for building the ET is efficient and its design is easily extendible. For the Hamming distance and wildcard matching, the cost of constructing the ET is $O(n\frac{log_{\Sigma}^{k}n}{k!})$ words with a query time of $O(\frac{m^k}{k!} + occ)(O(m + \frac{log_{\Sigma}^{k}n}{k!} + occ)$ in the average case) for any online or offline pattern, where occ is the number of occurrences to be reported. For the edit distance, a construction space of $O(2^k n\frac{log_{\Sigma}^{k}n}{k!})$ words are required, and $O(\frac{m^k}{k!} + 3^k occ)$ ($O(m + \frac{log_{\Sigma}^{k}n}{k!} + 3^k occ$) in the average case) of query time for any online or offline pattern.

2.1 Background and related work

Many algorithms have been recently proposed that index the text or the dictionary initially to allow for faster querying. Table 2.1 provides a comparison of most of these recent algorithms. Some algorithms were designed to solve the k-Hamming distance, k-edit distance, wildcard matching, or all of them. For the Hamming or edit distance, the data structure of [21] can output the k-edit distance in $O(3^k m^{k+1} + occ)$ time, where the construction of the structure requires $O(nlog^k n)$ words of space in the average case and takes $O(KN\Sigma)$ time, where N is the number of nodes in the index. An algorithm with upper space bound was proposed by [88], where the space complexity of the index structure is $O(n^{1+\epsilon})$ words for any constant $\epsilon > 0$, but with a query time of $O(m + loglogn + occ)$ for both the k-Hamming and k-edit distances.

Some algorithms solve both distances using a lower structure space and an upper query

Table 2.1: Comparison of the known results of the Text Indexing problem for the different matching problems and a bounded k. Hamming distance (HD), edit distance (ED), and wildcard matching (WM).

	Authors	Construction Space	Query Time	
HD	[22]	$O(n\frac{(c_1 logn)^k}{k!})$ words	$O(m + \frac{(c_2 logn)^k loglogn}{k!} + occ)$	$c_1, c_2 > 1$
	[88]	$O(n^{1+\epsilon})$	$O(m + loglogn + occ)$	$\epsilon > 0$, same for ED
ED	[22]	$O(n\frac{(c_3 logn)^k}{k!})$ words	$O(m + \frac{(c_4 logn)^k loglogn}{k!} + 3^k occ)$	$c_3, c_4 > 1$
	[43]	$O(n\sqrt{logn})$ bits	$O(\Sigma^k m^k(k + loglogn) + occ)$	
	[15]	$O(n)$,	$O(m + log^{k(k+1)} nloglogn + occ)$	
		$O(n)$	$O(log^{\epsilon} n(\Sigma^k m^k(k + loglogn) + occ))$	$0 < \epsilon \leq 1$.
	[21]	$O(nlog^k n)$ words	$O(3^k m^{k+1} + occ)$, average case	$O(KN\Sigma)$ time, N: number of index's nodes
	[39]	$O(nlogn)$ bits	$(\Sigma^k m^k max(k, logn) + occ)$	
WM	[22]	$O(n\frac{(k+logn)^k}{k!})$ words	$O(m\frac{2^k loglogn}{k!} + occ)$	
	[12]	$O(nlognlog_{\beta}^{k-1} n)$ words	$O(m + \beta^k loglogn + occ)$	$2 \leq \beta \leq \Sigma$
	[45]	$O(nlog^k nlog\Sigma)$ bits	$O(m + 2^k logn + occ)$	

time. Among these algorithms, a linear space index $O(n)$ was presented by [15], but with $O(m + log^{k(k+1)} nloglogn + occ)$ query time. Moreover, a data structure of $O(n\sqrt{logn})$ bits was proposed by [43], which requires $O(\Sigma^k m^k(k + loglogn) + occ)$ query time; by using $O(n)$ bits of space, the query time can be $O(log^{\epsilon} n(\Sigma^k m^k(k + loglogn) + occ))$, where $0 < \epsilon \leq 1$. By using more space, [39] showed an index structure that requires $O(nlogn)$ bits and takes $(\Sigma^k m^k max(k, logn) + occ)$ query time; the index could also be reduced to $O(n)$ by increasing the query time by a factor of $O(logn)$.

For k-wildcard matching, there is a slight reduction in both time and space costs over the Hamming and edit distances. Many algorithms introduced structures for solving this matching problem such as [12, 45]. The algorithm that was introduced by [12] could generalize the structure of [22] and reduce the space to $O(nlognlog_{\beta}^{k-1} n)$ words; however, the query time increased to $O(m + \beta^k loglogn + occ)$, where $2 \leq \beta \leq \Sigma$. A structure with less space was presented in [45], where the required space was $O(nlog^k nlog\Sigma)$ bits; however, with a slight increase in the query time, $O(m + 2^k logn + occ)$.

Data structures with different bounds for the approximate matching problems were presented by [22]. These include a data structure that requires $O(n\frac{(c_1 logn)^k}{k!})$ words and allows the k-Hamming distance to be found in $O(m+\frac{(c_2 logn)^k loglogn}{k!}+occ)$ time, and also a structure of $O(n\frac{(c_3 logn)^k}{k!})$ words that allows the k-edit distance to be found in $O(m+\frac{(c_4 logn)^k loglogn}{k!}+3^k occ)$ time, where $c_1, c_2, c_3, c_4 > 1$ are constants. For wildcard matching, their data structure requires $O(n\frac{(k+logn)^k}{k!})$ words, which solves the problem in $O(m\frac{2^k loglogn}{k!}+occ)$ time. The design of these structures is mainly based on the centroid path decomposition. The construction of subtrees, which corresponds to the operations of substitution, insertion, deletion, and wildcarding is performed by using *LCA tree* [37] and a new data structure called *longest common prefix*.

2.2 Preliminaries

The length of string s is $len(s)$. Substring $s[x:y]$ is the substring from position x to position y. The ith suffix of string s is denoted as $suff(s,i)$. For a list l, $l[i]$ is the item at index i, $l[-1]$ is the item at the last index, and $l[-2]$ is the item before the last item in l, and so forth.

2.3 Dictionary Indexing Algorithm

To explain the steps of the algorithm, the steps for $k = 1$ are shown first. Then, the general design for $k \geq 2$ is explained.

2.3.1 k = 1 Case

The algorithm involves two stages: construction of a tree and searching for the strings that are at a one Hamming distance from P.

Construction stage:

1. A *generalized suffix tree* (GST) needs to be built for the strings in the dictionary [89]. Therefore, by the definition of the suffix tree, a leaf node for each suffix of the strings in the dictionary is constructed. The cost is $O(n)$ words.

2. All leaves and internal nodes in the suffix tree are assigned a unique key. The cost is $O(n)$ words.

   Definition 1: A suffix tree with a unique key identifying each leaf and internal node is called a *keyed suffix tree* (KST).

   Definition 2: For a string s and a given KST, the function *All Visited Nodes*, denoted as *AVN(s)*, returns a list of the nodes′ keys and the edges′ lengths along with their order, which results from traversing s in the KST.

   Corollary 1: In the case of $k = 1$ and because in problem 1 there exists leaves for all suffixes of the strings in the dictionary, AVN(s)[-1] can be found for any suffix s of any string in the dictionary in a constant time. This is owing to the fact that AVN(s)[-1] is the key of the last visited node after traversing the suffix s in the KST, which must be a leaf node. Therefore, by hashing all the leaves, AVN(s)[-1] can be returned in a constant time without traversing s in the KST.

Corollary 2: Given a KST and two strings, s_1 and s_2, where len(s_1) = len(s_2) and s_1 is in the KST, if the Hamming distance between s_1 and s_2 is 1 and a mismatch occurs at a position x that is not the last position, then AVN(suff(s_1, x+1)) = AVN(suff(s_2, x+1)). Similarly, AVN(suff(s_1, x+1))[-1] = AVN(suff(s_2, x+1))[-1].

3. Construct a compact tree for all the strings in the dictionary; $O(n)$ time and space.

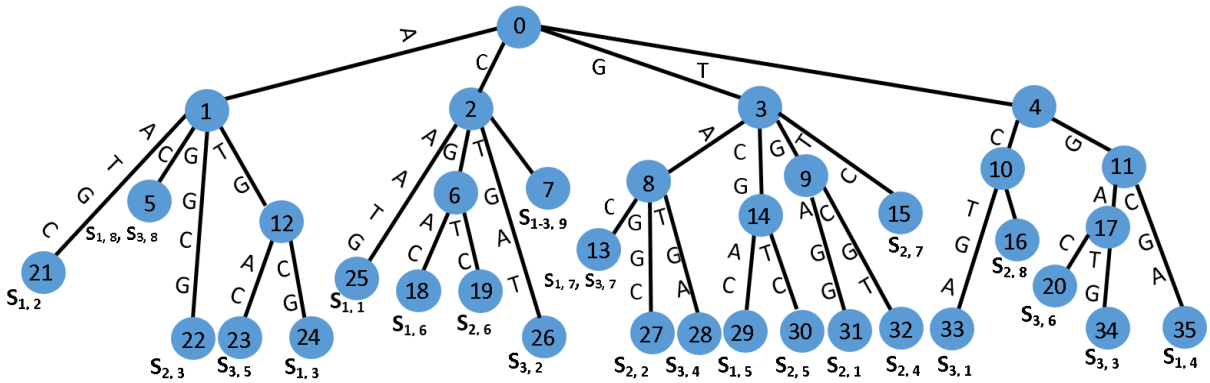
4. For each internal node v, a hash table I_1 is initialized. Let L be the set of all the leaves of the subtree rooted at v, and assuming v at level (symbol depth) i. Then, for all l in L, first choose any string s labeled at l, and then add to I_1 a tuple of $(AVN(suff(s, i+1)[-1], l)$. Therefore:

```
For each internal node v in the tree:
     initialize a hash table I_1
     i = get_level(v)
     L = get_desc_leaves(v).
     For l in L:
         choose a string s labeled at l
           v.I_1.add(AVN(suff(s, i+1)[-1], l)
```

Definition 3: Without loss of generality, such a tree structure is called *1*-ET because it is constructed to find one mismatch.

Fig. 2.1 shows an example of a *1*-ET structure for the sequences CAATGCGAC, GGAGGCGTC, and TCTGATGAC.

Lemma 1: The above tree structure uses a space of $O(Nlog_{\Sigma}N)$ words.



For ***1-ET***, at each node there is a hash table I_1. For instance:

Node(0): {($S_{1,2}$, (24, 1)), ($S_{2,3}$, (32, 1)), ($S_{3,5}$, (20)), ($S_{1,3}$, (35, 1)), ($S_{1,1}$, (21, 1)), ($S_{3,2}$, (34, 1)), ($S_{2,2}$, (22, 1)), …, ($S_{3,1}$, (26, 1)), ($S_{3,3}$, (28, 1)), ($S_{1,4}$, (29, 1))}. Node(1): {($S_{1,2}$, (32, 2)), ($S_{2,3}$, (14)), ($S_{3,5}$, (13)), ($S_{1,3}$, (14))}. Node(9): {($S_{2,1}$, (9)), ($S_{2,4}$, (15, 1))}. Node(14): {($S_{1,5}$, (7)), ($S_{2,5}$, (7))}.

For ***2-ET***, at each node there is a hash table I_2. For instance:

Node(0): {($S_{1,2}$, (31, 2)), ($S_{1,2}$, (32, 2)), ($S_{1,2}$, (1, (14, 1))), ..., ($S_{3,5}$, (13)), ($S_{3,5}$, (4, 5)), ($S_{3,5}$, (11, 7)), …, ($S_{1,4}$, (18, 1)), ($S_{1,4}$, (3, 8)), ($S_{1,4}$, (14, 1))}. Node(2): {($S_{1,1}$, 11), ($S_{3,2}$, (12, 1)) *(note this for positions 4 and 5)*, ($S_{1,1}$, 1, 3), ($S_{3,2}$, 3, 4), ($S_{1,1}$, (12, 1) *(note this for positions 3 and 4)*), ($S_{3,2}$, 8)}. Node(11): {($S_{3,3}$, 3), ($S_{3,3}$, 4), ($S_{1,4}$, 1), ($S_{1,4}$, 3)}. Node(6): {-}.

Fig. 2.1: This tree structure is the *GST* for the sequences {CAATGCGAC, GGAGGCGTC, and TCTGATGAC}, after truncating all the paths of depth greater than $l = 5$. At the top of the *GST*, *1-ET* and *2-ET* structures are constructed. Each node and leaf is assigned a unique key. Samples of hash tables, I_1 and I_2, are computed and shown for the corresponding nodes. *Node(x)* implies the node with key x.

Proof: Step 4 is performed for $O(N)$ internal nodes, and at each node it is bounded to the number of descendant leaves. If *1*-ET is unbalanced, then step 4 will not be performed for the leaves under a branch that is on a heavy path (the path of most descendant leaves). Namely, there are O(Σ) branches at each node; all descendant leaves of the branch on a heavy path will be excluded in step 4 and treated in the query stage as an edge. This means that the balanced tree will be the worst-case scenario. Then, at each level of the $O(log_{\Sigma}N)$, N keys will be stored in the hash tables of the nodes that are at that level. Therefore, the bound will be *O(N*log_{Σ}*N)* words of space.

Query stage

The first step is to add all the suffixes of the pattern P to the KST and then compute AVN(.)[-1] for each suffix. Therefore, the results will be a list, R, that must equal {AVN(suff(P, 1))[-1], AVN(suff(P, 2)[-1]),..., AVN(suff(P, m))[-1]}$_m$. Secondly, traverse P in *1*-ET as follows. If the walking (or traversing) is on an edge and the next symbol in P matches with the next symbol on the edge, then continue the walk as an exact match. If the next symbol in P does not match with the next symbol at level j, it means that a mismatch has been encountered. In that case, jump over the next symbol (since the walking is on an edge) and continue the walk as an exact match until a leaf node (if any) is reached. Then, output the strings that are labeled at that leaf as a 1-Hamming distance at position j. Now, if the walking reaches a node v where v is at level j, then check whether the key $R[j+1]$ is in the I_1 table of v (the constant time cost, I_1, is a hash table). If the key is found, all strings that are labeled at the leaves that are associated with key $R[j+1]$ are at a 1-Hamming distance at position

j. Next, continue the walk as an exact match and search for $k \leq 1$ mismatch. If the next symbol in P does not match with any child of v, the search is discontinued.

Lemma 2: The above query requires a time of $O(m + occ)$.

Proof: Traversing the pattern in *1*-ET costs m. Assuming that m nodes are visited during the walk, then m hash tables need to be looked up. The cost of each look-up is a constant time. If any, the cost for outputting the occurrences must also be added.

Theorem 1: When $k = 1$ *and for the Hamming distance and wildcard matching, problem 1 can be solved by indexing the dictionary using* $O(N\log_{\Sigma}N)$ *words with a query time of* $O(m + occ)$.

Proof: By lemmas 1 and 2.

Extension for indels

The design can be extended to handle the operations of insertions and deletions. This means that all strings with an edit distance less than or equal to k ($k = 1$) with P can be found.

Insertions and deletions result in shifting of the suffixes; such shifts must be tracked and manipulated by the design of the structure using the AVN function. If two strings s_1 and s_2 are at an edit distance of 1 that is caused by a deletion operation at a position x of s_2, then this means that $suff(s_1, x)[1 : m - x - 1] = suff(s_2, x + 1)$. Now, as the AVN function starts at the root node and *must* end up at a node with a *unique key*, the design of the structure should guarantee the same. For $suff(s_2, x + 1)$, it must end up at a node and this should not result in a conflict in computing AVN. However, $suff(s_1, x)[1 : m - x - 1]$, which is actually $1 = k$ level up from the end of suffix $suff(s_1, x)$, may result in a conflict

(as it may not be a leaf node). Therefore, this position must be guaranteed to be a leaf node with a unique key. Such a preprocessing step must be performed accordingly. As an alternative solution to creating a new node with a unique key at some point a in the middle of edge e, one can identify such a point by the value of *(t, key(sink(e))*), where *sink(e)* is the sink node of edge e, t is the distance between a and *sink(e)*, and *key(x)* is the unique key that identifies x. However, for a simpler explanation, let us create a new node to identify a point in the middle of an edge. Hence, the following step must be performed.

Preprocessing step: Let us assume that at each internal node there is a list L that stores the leaf labels under that node. Now, for each internal node v in *1*-ET, and for each leaf l in L of the subtree rooted at v, and assuming v is at level i, choose any string s labeled at l, then walk up by 1=k level of the parent node of the leaf node of $suff(s, i)$. If a node is reached, say x, then check whether x has a leaf node as a child. If not, create a new leaf node with a unique key. If there is no node, then a new node with a unique key is created. As a child of this new node, create a leaf node with a unique key. The cost will be $O(Nlog_{\Sigma}N)$ space and time. This will help to track the effects of shifting the suffixes because of the deletion and insertion processes.

Insertions and deletions can occur in the pattern or in the strings. Before explaining these four possible cases, the following corollary must be introduced, given that the preprocessing step has already been performed.

Corollary 3: Given a KST and two strings, s_1 and s_2, where $\text{len}(s_1) = \text{len}(s_2) = m$ and s_1 is in the KST, if the edit distance between s_1 and s_2 is one and the edit operation is a deletion at position x in s_2 where x is not the last position, then

AVN(suff(s_1, x))[1:m-x-1] = AVN(suff(s_2, x+1)). Similarly AVN(suff(s_1, x))[-2] = AVN(suff(s_2, x+1))[-1]. Note that suff(s_1, x))[1:m-x-1] will always end up at a leaf node (that must have a unique key) because of the preprocessing step.

Corollary 4: Given a KST and two strings, s_1 and s_2, where len(s_1) = len(s_2) = m and s_1 is in the KST, if the edit distance between s_1 and s_2) is one and the edit operation is an insertion at position x in s_2 where x is not the last position, then

AVN(suff(s_1, x+1)) = AVN(suff(s_2, x)[1:m-x-1]). Note also that suff(s_2, x)[1:m-x-1] will always end up at a leaf node (that must have a unique key) because of the preprocessing step; therefore, similarly, AVN(suff(s_1, x+1))[-1] = AVN(suff(s_2, x))[-2].

2. The four possible cases for the edit distance are as follows.

Case 1: Deletion in the strings. This can be handled using the I_1 table. Based on corollary 3, one can check whether AVN(suff(P, i)[-2] is in I_1. If so, then all strings that are labeled at the leaves that are associated with the key of AVN(suff(P, i)[-2] must have an edit distance with P as a deletion (in the strings) at position i.

Case 2: Insertion in the strings. For this case, another hash table I_{1_ins} needs to be initialized at each internal node. Then, step 4 of section 2.3.1 should be computed, but instead of adding (AVN(suff(s, $i+1$))[-1], l) into I_1, (AVN(suff(s, i))[-2], l) is added to I_{1_ins}. Note that because of the preprocessing step, AVN(suff(s, i))[-2] is always a leaf node with a unique key. This allows us to check whether AVN(suff(P, $i+1$)[-1] is in I_{1_ins}, based on corollary 4. If so, then all the strings that are labeled at the leaves that are associated with the key AVN(suff(P, i)[-2] must have an edit distance with P as an insertion (in the strings) at position i.

Before proceeding to the next two cases, note that a deletion in the strings is similar to an insertion in the pattern. Likewise, an insertion in the strings is similar to a deletion in the pattern.

Case 3: Deletion in the pattern. There is no need to modify the construction of *1*-ET. This case can be computed by searching AVN(suff(P, $i+1$)[-1] in I_{1_ins}.

Case 4: Insertion in the pattern. This case can be computed by searching AVN(suff(P, i)[-2] in I_1.

Theorem 2: When k=1 and for the edit distance, problem 1 can be solved by indexing the dictionary using O(Nlog_ΣN) words, and O(m + occ) of query time.

Proof: The cost for the preprocessing step will be $O(Nlog_\Sigma N)$ words. At each internal node, there are two hash tables corresponding to the operations of mismatch and insertion. Computing I_{1_ins} will be the same as computing I_1 in step 4 of section 2.3.1, which is $O(Nlog_\Sigma N)$ words of space. The query time, as described for each case, will be $O(m+occ)$.

2.3.2 Case of $k \geq 2$

In the case of $k = 1$, the main step in the design is the association of only the key of the last node(leaf), AVN(.)[-1], of the suffixes that are labeled at the leaves. In the case of $k \geq 2$, the keys of all the nodes that are returned by AVN(.) for some suffix s need to be associated with all tuples that contain s in the I_{k-1} tables that are stored in the nodes on the path of s in (*k-1*)-ET. Before describing the steps of the design, we need to state the following corollary.

Corollary 5: Given a KST and two strings, s_1 and s_2, where len(s_1) = len(s_2) and s_1

is in the KST, if the Hamming distance between s_1 and s_2 is k and the mismatches occur at positions $pos = \{p_1, p_2, ..., p_{k-1}, p_k\}$ assuming that there is a node at each level of position in pos in the path to s_1, then

$AVN(s_1[1 : p_1 - 1])) = AVN(s_2[1 : p_1 - 1])),$

$AVN(s_1[p_1 + 1 : p_2 - 1])) = AVN(s_2[p_1 + 1 : p_2 - 1])),$

.

.

$AVN(s_1[p_{k-1} + 1 : p_{k-1}])) = AVN(s_2[p_{k-1} + 1 : p_k - 1]))$

Equivalently,

$AVN(s_1[1 : p_1 - 1]))[-1] = AVN(s_2[1 : p_1 - 1]))[-1],$

$AVN(s_1[p_1 + 1 : p_2 - 1]))[-1] = AVN(s_2[p_1 + 1 : p_2 - 1]))[-1],$

.

.

$AVN(s_1[p_{k-1} + 1 : p_{k-1}]))[-1] = AVN(s_2[p_{k-1} + 1 : p_k - 1]))[-1]$

Construction stage:

1. The first step is to collect not only the key of the last visited node (leaf) in the path of some suffix s, but also all the keys of the nodes in the path of s. Therefore, we perform the following for each internal node v.

   (a) Given that the descendant leaves under node v are stored in L and the level of v is i, then initialize a hash table I_k, and for each leaf l in L choose any string s. Next, compute *AVN(suff(s, i + 1))*. Note that in the computation of AVN(),

if the walking is on an edge, then AVN() returns the length of that edge and a tag indicating that fact. If the walking is at a node, then the key of that node is returned with a tag indicating that the walking is at a node.

(b) After computing *AVN(suff(s, i + 1))*, first traverse k-ET to leaf l, skipping one level, and perform the following:

```
if the next node u in AVN(suff(s, i + 1)) is aligned with the middle of an edge:
    v.I_k.add( ((u.key(), edge), l))
if the next node u1 in AVN(suff(s, i + 1)) is aligned with a node u2:
    for each tuple p in u2.I_k-1 that has l:
       v.I_k.add(((u1.key(), p[1]),.., p[k]))
```

Note that there is no need to walk explicitly to leaf l; instead, we check the alignment between a visited node in the path with a node in *AVN(suff(s, i + 1))*, or the length of an edge that was visited in the path with the edge's length in *AVN(suff(s, i + 1))*, which is a simple convolution. Therefore, the following cases must be experienced while walking to leaf l in k-ET.

Case 1: The next node u in *AVN(suff(s, i + 1))* is aligned with the middle of an edge in k-ET. Then, add to I_k of v a tuple that contains: the key of u, a tag indicating the alignment is at an edge, and l.

Case 2: The next node $u1$ in *AVN(suff(s, i + 1))* is aligned with a node $u2$ while traversing k-ET. Then, for each tuple p that has l in I_{k-1} of $u2$, associate the key of $u1$ and the items in P in their order as a tuple and then add this new tuple into I_k of v.

In the case of an unbalanced KST, steps 1.1 and 1.2 may cost more than $O(log_{\Sigma}n)$

for each suffix. In step 1.1, the paths for all suffixes under some internal node are computed by traversing the KST beginning from the root. This can be optimized instead, as these suffixes share the same prefixes, hence the same nodes will be visited. Furthermore, all tuples in an I_{k-1} will eventually be associated with the same aligned node's key in the *AVN(.)* list, based on step 1.2.

Therefore, steps 1.1 and 1.2 can be computed using depth-first search and backtracking. Beginning by computing *AVN(suff(s, i + 1))* for the leftmost leaf *lf1* of, say the subtree rooted at v, where s is a string labeled at *lf1*, and i is the level of v. While traversing *1*-ET, if a node u is reached, then for each tuple in I_{k-1} of u, associate the node's key x (in *AVN(.)*) with each tuple, and then add the new tuple into I_k of v, and store x in node u. If a node's key x (in the *AVN(.)*) is aligned with the middle of an edge, then add the *(x, L)* into I_k of v, where L stores the strings that are labeled at the sink node of the edge. When *lf1* is reached, then backtrack to the parent node, p of *lf1*. Next, obtain the key x that was stored at p, go to node x in the KST and beginning from x, compute *AVN(suff(s, i))* where s is any string labeled at the leftmost leaf *lf2* of p, and i is the level of p. Then, walk to leaf $lf2$ in the same manner by using the new AVN() list. Repeat the process until the depth-first search is finished. Note that this process will reduce the construction time by a factor of $O(log_{\Sigma}n)$ in the case of a balanced KST.

At the end of the walking, note that the keys of $O(log_{\Sigma}n)$ node may be associated with $O(\frac{log_{\Sigma}^{k-1}n}{k!})$ tuples that are in I_{k-1} during the walking to leaf l. Therefore,

eventually, each level costs $O(N\frac{log_{\Sigma}^{k-1}n}{k!})$ words of space. As there are $O(log_{\Sigma}N)$ levels, the bound will sum to $O(Nlog_{\Sigma}N\frac{log_{\Sigma}^{k-1}n}{k!})$. Fig. 2.1 shows a sample of *2-ER*.

2. Steps 1.1 and 1.2 are performed for *suff(s, i+1)* on the tables I_{k-1} skipping 1 level. Similarly, the same steps are performed for *suff(s, i+2)*,...,*suff(s, i+k-1)* on the tables of I_{k-2},...,I_1 skipping 2,..., $k-1$ level, respectively.

3. Thus far, the above steps cover the case in which, at each internal node, the symbols at the first $k-1$ levels are errors. However, they do not cover the case in which all the first k symbols after the internal nodes are actually errors. In this case, perform step 4 of section 2.3.1, for suff(s, i + k) instead for suff(s, i + 1) as in the case of $k = 1$. The cost for this case is the same as that for case $k = 1$, $O(Nlog_{\Sigma}N)$ words of space.

   Lemma 3: The above tree structure uses $O(Nlog_{\Sigma}N\frac{log_{\Sigma}^{k-1}n}{k!})$ *space.*

   Proof: As there are $log_{\Sigma}N$ levels, note that all I_k tables at, say level i, contain all tuples of tables I_1, I_2, ..., I_{k-1} that are at each level $i+1$, $i+2$, ..., $log_{\Sigma}N$. This sums up to $\frac{log_{\Sigma}^{k-1}n}{k!}$. Thus, over all $O(log_{\Sigma}N)$ levels, the cost of constructing k-ET for any k will be $O(Nlog_{\Sigma}N\frac{log_{\Sigma}^{k-1}n}{k!})$ words.

Query stage:

When $k = 1$, the number of possible error positions is m, as m is the length of P. For $k \geq 2$, the number of possible combinations of error positions would be $\binom{m}{k}$, which is bounded to $O(\frac{m^k}{k!})$.

Before stating the steps of querying a pattern, the following cases that may arise when the AVN(.) function is computed for each suffix of the pattern need to be described.

Case 1: Traversing a suffix in the KST diverges at an internal node. The design of the algorithm already covers this case, as all the internal nodes in the KST are marked back in the ET.

Case 2: Traversing a suffix in the KST diverges in the middle of an edge. In this case, jumping (skipping the next symbol in the edge) should be allowed k times during the walking *at that edge or any subsequent edge*. If the walking ends up at a leaf after no more than k jumps, then deduct the number of jumps that were performed out of the k mismatches during the searching process. If the walking, after no more than k jumps, ends at an internal node, then this case is similar to case 1 but after deducting the number of jumps that were performed out of the k mismatches during the searching process. If a node (internal or leaf) is not reached after exactly k jumps, then *P will have no outputs at all of k mismatches with any string in the dictionary*, as one of its suffixes does not reach a leaf or an internal node after allowing k jumps (where jumps represent mismatches). Note that jumps are counted only at edges and not on any internal node, as the algorithm's design is already marking the internal nodes back in the ET, and the jumps (assumed to be errors) after these internal nodes are already accounted for in the design.

According to these cases, we define the following function.

Definition 4: For a string s, an integer k, and a given KST, the function *All Visited Nodes with k jumps*, denoted as *AVNJ(s, k)*, returns a list of the keys of the nodes and the lengths of the edges (with their order) resulting from walking s in the KST allowing k jumps

(in case of mismatches) on only the edge, and the positions of any jumps that occurred.

Therefore, the query process is computed as follows.

1. Collect $AVNJ(s,k)$ for each suffix $s_1, s_2, ..., s_m$ of the pattern. Then, there are m lists. Let us call this list R.

2. Walk the pattern P in k-ET, then at each internal node v and assuming v is at level i, search whether I_k contains any of the combinations of $\binom{m-i}{k}$ keys that can be extracted from the list R. If so, report the leaves' labels that were associated with the key combinations as the output. If the walk is on an edge, skip (jump) over any mismatches that are encountered, which is a simple convolution.

Lemma 4: The above query time is $O(\frac{m^k}{k!} + occ)$ *or* $O(m + \frac{log_\Sigma^k n}{k!} + occ)$ *in the average case.*

Proof: At each node v visited while walking the pattern, it is necessary to search in the hash tables at v for combinations of error positions, which can be $O(\frac{m^k}{k!})$, that can be extracted from list R. Note that, in the average case, each list in R will have $O(log_\Sigma n)$ nodes (unique keys). Hence, in this case, a search of $O(\frac{log_\Sigma^k n}{k!})$ combinations will be needed.

Theorem 3: When $k \geq 2$ *and for the Hamming distance and wildcard matching, problem 1 can be solved by indexing the dictionary using* $O(Nlog_\Sigma N \frac{log_\Sigma^{k-1} n}{k!})$ *words, and* $O(\frac{m^k}{k!} + occ)$ *(*$O(m + \frac{log_\Sigma^k n}{k!} + occ)$ *in the average case)* query time.

Proof: By lemmas 3 and 4.

Extension for indels

In order to handle the insertions and deletions for any value of k, the following modifications should be considered.

1. A leaf node at the k^{th} level above all leaves in the KST must be created or guaranteed. Therefore, visit k levels above each leaf and perform the preprocessing step given in section 2.3.1. Note that, during the construction of *1*-ET to *(k-1)*-ET, the preprocessing step for each case of 1 to $k-1$ must have been computed already.

2. For insertion operations only, at each internal node and assuming i is the level of the node, perform step 1 in section 2.3.2 for *suff(s, i)[1:m-i-k-1]* on table I_{1_ins}, then add the results into the I_{k_ins} table. In addition, perform the same for *suff(s, i)[1:m-i-k-2]*,..,*suff(s, i)[1:m-i-2]* on the tables of I_{2_ins},..,I_{k-2_ins}.

3. Likewise, step 3 of section 2.3.2 must be performed, but for *suff(s, i)[1:m-i-k]* and by adding the results into I_{k_ins}.

   Note that the edit distance can be any combination of substitutions, deletions, or insertions. For this, perform steps 1, 2, and 3 above for all the tables at a node; namely, the I_k and I_{k_ins} tables, as well as I_{k_ins}. Then, add the results into a hash table I_{k_edit}. This will add an extra space of 2^k words. Therefore, the total cost for building k-ET that can handle a k edit distance will be $O(2^k N log_\Sigma N \frac{log_\Sigma^{k-1} n}{k!}))$.

4. The number of combinations to be searched will increase by a factor of 3^k. Hence, the query time for the edit distance will be $O(\frac{m^k}{k!} + 3^k occ)(O(m + \frac{log_\Sigma^k n}{k!} + 3^k occ)$ in the average case).

Theorem 4: When $k \geq 2$ and for the edit distance, problem 1 can be solved by indexing the dictionary using $O(2^k N log_\Sigma N \frac{log_\Sigma^{k-1} n}{k!})$ words, and a query time of $O(\frac{m^k}{k!} + 3^k occ)$ ($O(m + \frac{log_\Sigma^k n}{k!} + 3^k occ)$ in the average case).

Proof: Omitted.

2.4 Algorithm Design for Text Indexing

The design and construction of the ET for problem 2 are similar to that of problem 1, but there are some differences. Here, we describe the differences and preprocessing steps that are required to resolve them in order to apply the same design and construction of problem 1 to problem 2.

1. The depth of each path in the suffix tree might not be less than or equal to m. Paths with depths greater than m are useless while searching for a pattern of length m. Moreover, such paths increase costs through backward traversing of the tree and during the creation of new nodes.

2. In problem 1, a leaf node for each suffix of the strings must have been already constructed; this is guaranteed by the design and construction of generalized suffix trees. This is not the case for problem 2, in which there is just one text string. The leaves for the suffixes of each suffix *(or specifically m-mer)* are not constructed explicitly.

Now, in order to resolve these issues, the following steps are needed:

1. All paths in the suffix tree must have a depth of no more than m. This can be achieved by traversing all the paths, then the depth of each path is counted by adding the lengths

of the edges on each path. When a depth of m is reached, then if that point is already a node, trim all edges/nodes below that node and store the labels of the descendant leaves explicitly. If the point is on an edge, create a leaf node, then explicitly store the labels of the descendant leaves of the sink node of the edge, and trim the edge below that point. The cost of this step will be $O(n)$ time and space, since there is no need to read the edges' symbols; only the length of the edges need to be read (constant time). Moreover, the number of new nodes that will be created is $O(n)$. Such a suffix tree is called a *Trimmed Suffix Tree* of depth m (TST_m).

2. Beginning from TST_m, we need to mark/tag suffixes of these suffixes similar to the design of problem 1. Note that in problem 1 not all m suffixes of the strings were considered in the design, since we only computed the AVN(.) for the suffixes of the descendant strings under the internal nodes. Thus, $O(nlog_{\Sigma}n)$ of suffixes will be under consideration as opposed to $O(nm)$. In order to resolve this, note that after performing step 1, the sixth suffix (for instance) of the suffix at position 1020 will be the prefix from root to position $m-6$ of the suffix 1026. Therefore, the cost to guarantee/create a leaf node for the sixth suffix of suffix 1020 is to start from the leaf node of suffix-1026 and walk as far as position $m-6$. Then, make sure there is a leaf node there or create a new one with a unique key for identification. Again, there is no need to walk on the edge explicitly to reach point $m-6$, as it is sufficient to read the length of the edges. Hence, the cost will be $O(log_{\Sigma}n)$.

   In the case of an unbalanced suffix tree, this step can be performed by using a depth-first search and backtracking to the KST, as described in section 2.3.2. In conclusion,

the cost of guaranteeing/creating a leaf node for any suffix under consideration for any string labeled at an internal node is $O(log_{\Sigma} n)$ additional time.

3. There is no need to build another compressed tree for the text in this problem, as the TST_m may be considered as a sufficient representation for all the k ETs. Therefore, all operations and all the k ETs can be constructed at the top of the TST_m. Alternatively, each of the k ETs can be constructed using an independent tree.

   After performing these modifications, k-ET can be built using the same steps employed in problem 1, at a cost of $O(n\frac{log_{\Sigma}^{k} n}{k!})$ words.

The above design constructs an ET structure that can answer a query for any pattern of length m. In order to design the ET to handle any pattern of length less than or equal to $n - k$, perform the following modifications beginning from the KST.

1. Index all the leaves from left to right of the KST so that the set of leaves under the subtree rooted at each internal node, v, is indexed using an interval of two integers.

2. Now, for any case of k (even in the case of k=1), collect the keys in the KST using a depth-first search. In addition, instead of associating all the tuples of I_{k-1} with the explicit leaves' label that are stored at a visited node, associate them instead with the index label (computed by the above step) of a visited node instead. Therefore, step 1.b in section 2.3.2 is modified to the following, given that this step must be performed using a depth-first search, as is discussed in section 2.3.2:

```
if the next node u in AVN(suff(s, i + 1)) is aligned with the middle of an edge:
    v.I_k.add( ((u.key(), edge), IL(edge.sink)))
```

```
    // where IL(edge.sink)) is the index label of the sink node of the edge.
if the next node u1 in AVN(suff(s, i + 1)) is aligned with a node u2:
    for each tuple p in u2.I_k-1:
        v.I_k.add(((u1.key(), p[1]),.., p[k]))
```

Theorem 5: When $k \geq 1$ *and for the Hamming distance and wildcard matching, problem 2 can be solved by indexing the text using* $O(n\frac{log_{\Sigma}^{k}n}{k!})$ *words, and a query time of* $O(\frac{m^k}{k!} + occ)$ *(*$O(m + \frac{log_{\Sigma}^{k}n}{k!} + occ)$ *in the average case).*

Proof: Omitted.

Theorem 6: When $k \geq 2$ *and for the edit distance matching, problem 2 can be solved by indexing the text using* $O(n\frac{log_{\Sigma}^{k}n}{k!})$ *words, and a query time of* $O(\frac{m^k}{k!} + 3^k occ)$ *(*$O(m + \frac{log_{\Sigma}^{k}n}{k!} + 3^k occ)$ *in the average case).*

Proof: Omitted.

Chapter 3

ET-Motif: Solving the Exact (l, d)-Planted Motif Problem Using Error Tree Structure

In this chapter, an exact solution (*ET-Motif*), for the (l, d)-Planted Motif Problem using the Error Tree (ET) structure is proposed. In addition, *Quorum Planted Motif Search (qPMS)* and *Edit Distance Planted Motif Search (EDPMS)* problems are also addressed.

3.1 Introduction and Related Work

Motif finding is an important and a challenging problem in biology and bioinformatics. A motif is a substring that occurs in a set of DNA or protein sequences, where the substrings can differ in a number of mutations. The problem is also known in the literature as the *closest substring problem*. In terms of biological applications, finding such a motif can lead to the finding of promoters, enhancers, locus control regions, transcription factors, and more.

The problem involves several formulations: *Planted Motif Search*, *Simple Motif Search*, and *Edit-distance-based Motif Search*. Planted Motif Search (PMS) is the most studied formulation as it better reflects the models of real motifs in real DNA/protein data. The problem is stated as follows: Given n sequences of length m each, l denotes the length of the motif of interest, and d the number of mutations allowed, the *(l, d)*-PMS involves finding

a motif M that has at least one substring of length l (l-mer) in each sequence, where the *Hamming distance* between M and each such substring is no more than d. These substrings are called instances or variants of the motif M. The Hamming distance between two strings of the *same* length is the number of substitutions required to transform the first string into the second one.

An algorithm that solves the PMS problem can be *exact*, where all motifs for the given sequences can be found; or *approximate*, where not all motifs in the sequences can be found. The worst time complexity of any exact algorithm must be exponential to one of the parameters, as it has been proven that the exact PMS problem is NP-complete [44]. In this dissertation, we address the exact solutions for the PMS problem.

For both, the exact and approximate algorithms, several approaches have been proposed to solve the PMS problem; which are mainly pattern-based and sample-driven. A pattern-driven approach selects candidate motifs from the input sequences, and then determines its validity. Many algorithms have adopted such an approach, such as Voting [20], PMS1 [69], PMS2 [69], PMS3 [69], stemming [42], PMS4 [70], PMS5 [24], PMS6 [9], and PairMotif [93]. On the other hand, sample-based approaches find the positions of the motif's instances in the sequences. Among the algorithms that have adopted this approach are: PMS8 [61], WINNOWER [65], DPCFG [92], RecMotif [83], and ListMotif [84].

Furthermore, PMS algorithms can be categorized based on the tree or graph structure adopted to solve the problem. Several graph-based algorithms were proposed, such as WINNOWER [65], DPCFG [92], cWINNOWER [46], and MotifCut [30]. The main theme in graph-based algorithms is to build a graph in which the nodes are l-mers (substrings of

length l) derived from the input sequences, and an edge connects two nodes if the Hamming distance between both of the nodes (l-mers) is no more than $2d$. Then, the motif finding proceeds by searching for cliques, then computes the motif out of these cliques. Such a design was implemented in WINNOWER.

Several algorithms were proposed to solve the PMS problem by using different tree structures. Most of these algorithms utilize the linear time and space of the *suffix trees* (ST) [89], and use different or novel tree structures such as *search tree* and *neighborhood tree* [27]. Among these algorithms are mainly SPLELLER [74], WEEDER [63], CENSUS [27], RISOTTO [66], FLAME [29], TreeMotif [85], and MITRA [26]. The first step in most of these algorithms is to build a ST for the input sequences. Then, preprocess the tree using several techniques or build another tree structure that employs the suffix tree. For instance, all d-neighbors of all l-mers in the input sequences are traversed in the suffix tree to be validated as a motif [74]. The d-neighbors of an l-mer r are all l-mers that are at a Hamming distance of no more than d from r.

A survey of the PMS problem has been addressed in several papers such as [87], [94], and [23].

3.2 Solving PMS problem using ET design

The design of ET contributes considerably from various aspects in solving the PMS problem. First, the design of ET involves assigning each node in an ST a unique key, so that any string (such as the prefix, substring, or suffix of a string) can be represented by a key value. Such a representation allows for faster manipulation of the alignment process by hashing these

keys. In addition, it facilitates referencing of the *same* tree to track prefixes, substrings, and suffixes between the strings in the input data and the string that needs to be aligned. For the PMS problem, this can be of great benefit since the problem involves considerably redundant prefixes, substrings, and suffixes. Before proceeding to the second contribution, let us state the following definition.

Definition 3: Assume strings x and y are of Hamming distance of no more than d and $P = p_1, p_2, .., p_d$ is a sorted list of the positions where the mismatches occur. Then, we call $s[p_1 + 1..p_2 - 1], s[p_2 + 1..p_3 - 1], .., s[p_d + 1..|x|]$ the *common exact substrings*.

The design of ET enables us to compute the alignment without computing the enumeration of $(\Sigma - 1)^d$ strings. This is due to the fact that ET's design computes the alignment by finding the common exact substring and disregards the mismatch positions as the alignment is of no more than d distance.

Lastly, finding alignments using ET is dependent on the depth of ST. If the depth is $O(l)$ (unbalanced ST), then the alignment cost can be $O(\binom{l}{d} + occ)$. If the depth is $O(log_\Sigma n)$ (balanced ST), then the cost is $O(l + \binom{log_\Sigma nm}{d} + occ)$, where d is the allowed Hamming distance and nm is the total length of the input sequences. Note that, processing of each l-mer in the input sequences is independent from that of the other l-mers; this disregards any similarities (even approximate) among them. Undoubtedly, these similarities can be exploited for a faster solution. Therefore, employing tree structures that can reveal such similarities can result in a faster solution.

3.3 Algorithm for PMS problem

The problem is to find a motif M of length l, where in each sequence there is at least one l-mer at a Hamming distance of no more than d from M.

The main ideas of the ET-Motif algorithm are as follows. First, all d-neighbors of each l-mer in an arbitrary sequence in the input sequences can be generated. Then, the alignments of no more than d Hamming distance between each neighbor and all l-mers in the other input sequences can be searched using the design of ET. Second, as the d-neighbors of the *suffixes* at positions i of *all* d-neighbors of an l-mer r are the *same* as the d-neighbors of suffix i of r. Therefore, any processing that can be computed on the d-neighbors of the *suffix* at position i of r, can be applied implicitly for each suffix at the same position i of all d-neighbors of r. Thus, there is no need to explicitly perform the same processing for each suffix at the position i of all d-neighbors of r.

Before describing the algorithm, let us state the following definition:

Definition 3: Assume ST is a suffix tree in which each node is assigned a unique key. Then, the function $keys(ST, s, p_1, p_2, .., p_d)$ returns a tuple of $(k_0, k_1, k_2, .., k_d)$ keys where $k_0 = key(s[1..p_1 - 1])$, $k_1 = key(s[p_1 + 1..p_2 - 1])$, .., $k_{d-1} = key(s[p_{d-1} + 1..p_d - 1]), k_d = key(s[p_d + 1..|s|])$.

3.3.1 ET-Motif algorithm

The algorithm is stated as follows.

1. Build a *generalized suffix tree* (GST) [91], for all n input sequences.

2. Trim (truncate) all paths in the GST so that each path is of a depth of no more than

l.

3. Assign a unique key for each node in GST.

4. For each l-mer r in s_1, initialize a hash table E. Then:

   (a) Generate d-neighbors for r, associate each neighbor with two integers, t and c, and add them into E. The integer t is used to temporarily store the sequence number of the last alignment that supported the associated neighbor. While the integer c stores the number of sequences that support the associated neighbor.

   (b) Add all l-mers in E into the ST. For any node that was visited by at least one l-mer, mark it as a *Target Node* (TN). Then, store a set N that stores all such l-mers at each TN.

   (c) Initialize $l-1$ hash tables, $S_1, S_2, .., S_{l-1}$. Then, for each S_i, compute for each combination of $\binom{l-i+1}{d}$, say P, and suffix i of r, say s, the function $keys(ST, s, P)$. Then, add the result into S_i.

   (d) At each TN node, let us have $n-1$ lists, $L_2, .., L_n$, where L_j stores the leaves' labels (input sequences' l-mers) that are in sequence j.

   (e) For each i in $2, 3, ..., n$:

   For each descendant leaf f in L_i at some TN, say tn, that is at level v:

   For each combination of $\binom{l-v+1}{d}$, P, and suffix v of f and s, compute $keys(ST, s, P)$. Then, find the entry of $keys(ST, s, P)$ in S_v. If found, for each neighbor h in N at tn, search for h in E. Then, check if the variable t that is associated with h is equal to i. If yes, do nothing.

If not, set t to i and increment the variable c that is associated with h by 1. Stop the search for the other leaves in L_i, because an l-mer that is of no more than d Hamming distance of the neighbors in h will already have been found in sequence i.

(f) Scan all neighbors in E. If the variable c that is associated with some neighbor is equal to n, output that neighbor as a valid motif.

3.4 Time and space complexity and comparison

This section states the time and space complexity of the ET-Motif algorithm, and presents a comparison of common tree-based algorithms.

Theorem 1: The (l,d)-planted motif problem can be solved in $O(nm^2 \sum_{j=0}^{d} \binom{l}{j}$ *time and* $O(\sum_{j=0}^{d} \binom{l}{j}(\Sigma-1)^j) + nml)$ space.

Proof: Steps 1, 2, and 3 each costs $O(mn)$ time and space. Step 4.a and 4.b each cost $O(\sum_{j=0}^{d} \binom{l}{j}(\Sigma-1)^j)$ time and space. Step 4.c costs $O(\sum_{j=0}^{d} \binom{l}{j}$ time and space. For step 4.d, the cost is $O(nml)$. Alternatively, if we use n indices for the leaves' labels of each sequence, this step can be performed using $O(n^2m)$ space. For step 4.e, the time cost is $O(nm \sum_{j=0}^{d} \binom{l}{j}$ in total (assuming all $O(nm)$ l-mers in the input sequences are labeled at all TN nodes). Lastly, step 4.f costs $O(\sum_{j=0}^{d} \binom{l}{j}(\Sigma-1)^j)$ time. Since step 4 can be performed in $m-l+1$ l-mers, then step 4 costs $O(nm^2 \sum_{j=0}^{d} \binom{l}{j}$ in total. Therefore, the total time cost is $O(nm^2 \sum_{j=0}^{d} \binom{l}{j})$, and the space cost is $O(\sum_{j=0}^{d} \binom{l}{j}(\Sigma-1)^j) + nml)$.

Theorem 2: If the GST of the input sequences is balanced, the (l,d)-planted motif problem can be solved in $O(nm^2 \sum_{j=0}^{d} \binom{log_\Sigma nm}{j}$ *time and* $O(\sum_{j=0}^{d} \binom{l}{j}(\Sigma-1)^j + nmlog_\Sigma nm)$

space, .

Proof: the proof is similar to the proof of Theorem 1 except in two steps. Step 4.d costs $(nmlog_{\Sigma}nm)$, because the depth of the ST is $(log_{\Sigma}nm)$. For step 4.f, not all l suffixes of each l-mer of the input sequences are considered for such computations, but only $(log_{\Sigma}nm)$ suffixes; hence, the total time cost is $O(nm\sum_{j=0}^{d}\binom{log_{\Sigma}nm}{j})$.

Theorem 3: The (l,d)-planted motif problem can be solved in $O(nm\sum_{j=0}^{d}\binom{l}{j})$ *time and* $O(m\sum_{j=0}^{d}\binom{l}{j}(\Sigma-1)^{j})$ space.

Proof: This theorem indicates that by using more space (factor of m), the PMS problem can be solved using a lower time bound by a factor of m. This can be achieved using the following modifications.

1. Steps 1, 2, and 3 are the same.

2. Initialize a hash table E. Then, for each l-mer r at position j in s_1:

   (a) Generate d-neighbors for r, and associate each neighbor with j and two integers t and c. Then, add them into E.

   (b) Initialize $l-1$ hash tables, $S_1, S_2, .., S_{l-1}$, then for each S_i and for each combination of $\binom{l-i+1}{d}$, P, and the ith suffix of r, s, compute the function $keys(ST, s, P)$. Create a hash table at the entry of $keys(ST, s, P)$ and add j into this hash table.

3. For each tuple of (r, h) in E, where h is a neighbor that was generated from l-mer r, add h into the ST. Let us mark any node that is visited by h as a *Target Node* (TN). At each TN, create up to m sets where set j stores all the tuples that have j in their first item (the first item of the tuple). In addition, associate a flag with each set. Let

all such sets be stored in a list N.

4. Let us store at each TN node, $n-1$ lists $L_2, ..., L_n$; so that L_j stores the leaves' labels (input sequences' l-mers) that are in sequence j.

5. For each i in $2, 3, .., n$:

   For each descendant leaf f in L_i at some TN, say tn, where t is at level v:

   For each combination of $\binom{l-v+1}{d}$, P, and for suffix v of f, s, compute $keys(ST, s, P)$. Then,

   (a) Find the entry y of $keys(ST, s, P)$ in S_v. If found, then for each tuple (that is in the form (r, h)) in each set e in N at tn, search for r in the hash table of y. If found, search for h in E. Then, check if the variable t that is associated with h equals i. If yes, do nothing. If not, set t to i and increment the variable c that is associated with h by 1.

   (b) Set the associated flag with set e to 1, as the neighbors in e will have already been supported by some l-mer with no more than d Hamming distance in s_i.

   (c) When all flags of all sets at tn are 1, stop the search for other leaves in L_i, because all neighbors in N will already have been supported by an l-mer in s_i that is of no more than d Hamming distance.

   Set all flags of all sets in N of all TN to be 0.

6. Scan all neighbors in E. If the variable c that is associated with some neighbor equals n, output that neighbor as a valid motif.

Theorem 4: In the case of a balanced generalized suffix tree (that is built for the input sequences), the (l,d)-planted motif problem can be solved in $O(nm\sum_{j=0}^{d}\binom{log_{\Sigma}nm}{j})$ *time and* $O(m\sum_{j=0}^{d}\binom{l}{j}(\Sigma-1)^j)$ *space..*

Proof: Omitted.

Many algorithms have been proposed for the problem. Table 3.1 presents a comparison between the known bounds, and the ones that are proposed in this chapter.

Table 3.1: Comparison of the time and space bounds of the tree-based algorithms that address the PMS problem, and the stated bounds in this chapter.

Authors	Time bound	Space bound (words)
SPELLER [74]	$O(lnm^2\sum_{j=0}^{d}\binom{l}{j}(\Sigma-1)^j)$	(lnm^2)
Mitra [26]	$O(lnm\sum_{j=0}^{d}\binom{l}{j}(\Sigma-1)^j)$	(lnm)
CENSUS [27]	$O(nm\sum_{j=0}^{d}\binom{l}{j}(\Sigma-1)^j)$	$O(lnm)$
RISOTTO [66]	$O(nm^2\sum_{j=0}^{d}\binom{l}{j}(\Sigma-1)^j)$	(nm^2)
This dissertation, balanced GST	$O(nm^2\sum_{j=0}^{d}\binom{log_{\Sigma}nm}{j})$	$O(nml)$
This dissertation, unbalanced GST	$O(nm^2\sum_{j=0}^{d}\binom{l}{j})$	$O(nml)$
This dissertation, balanced GST	$O(nm\sum_{j=0}^{d}\binom{log_{\Sigma}nm}{j})$	$O(m\sum_{j=0}^{d}\binom{l}{j}(\Sigma-1)^j)$
This dissertation, unbalanced GST	$O(nm\sum_{j=0}^{d}\binom{l}{j})$	$O(m\sum_{j=0}^{d}\binom{l}{j}(\Sigma-1)^j)$

3.4.1 Pruning Procedures

Several pruning procedures can be applied to speed up the solution for the PMS problem and/or to reduce the required space. To explain some of the main procedures, let us assume that the GST of the input sequences has already been built, and at each node in the GST

we have n lists, $L_1, L_2, .., L_n$ where L_j stores the leaves' labels (input sequences' l-mers) that are in sequence j.

- Find the sequence that has the maximum number of common neighborhood in its l-mers. The fastest and heuristically most feasible process to accomplish this is to find the sequence whose l-mers share more exact substrings. There are two factors for this computation, the length of the exact substring and the number of l-mers in the sequence that contains such a substring.

  Let us perform the process for some sequence x. First, we initialize an integer g. Then, traverse each node t in the GST where t is at level j, and calculate *the size of* $L_x \times$ *length of the suffix (which is* j*)* $\times \sum_{g=0}^{d} \binom{j}{g} (\Sigma - 1)^g$, where L_x is the list that contains all the leaf's labels that correspond to sequence x. Add up this value to g. Perform the same process for all the sequences in the input sequences. Then, let the sequence with the maximum value of g be the sequence for step 4. The cost of this procedure is $O(nml)$ time and space.

- Similarly, we compute the 1-neighbor of each sequence. Then, add them into a hash table where the count for each neighbor is computed using ET. For step 4, select the sequence that has neighbors with the maximum total counts. Next, perform the same steps for (*2, 3, ..., d'*)-neighbors where $d' << d$. The cost of these operations is $O(nm \sum_{j=0}^{d'} \binom{l}{j})$. Since d' is small compared to d, this cost can be neglected but may speed up the solution of the problem.

- This procedure attempts to prune neighbors that are not expected to be valid motifs

based on the following observation. If an l-mer r is a valid (l, d) motif, then any substring u of length l' of r is also a valid (l', d) motif. The contrapositive of this statement is also true; meaning if u is not a valid (l', d) motif, then r is not a valid (l, d) motif. Applying this procedure can prune neighbors that are not candidate motifs.

To apply such a procedure, select l' to be of a short length compared to l, i.e., $l' = \frac{l}{2}$. Then, scan s_1 using windows of length l' as follows: Let us assume that the window w is $s_1[i..i + l' - 1]$, which is at position i. This means that w lies within each l-mer $i - l', i - l' + 1, .., i = L$. Then, check if w is a valid (l', d) motif. If not, for each l-mer r in L and during the generation of d-neighbors of r, prune any neighbor that has w as an exact match.

3.5 Edit Distance Motifs and Edited Motif problems

The algorithm can be modified in simple ways to handle other variations of the PMS problem; namely, qPMS and EDPMS.

qPMS problem is defined similarly to PMS problem but with a slight modification. The motif is valid if it is presented in q sequences instead of all n sequences. Therefore, to solve the qPMS problem using the ET-Motif algorithm, step 4 must be computed not only for one sequence (s_1), but also for $n - q + 1$ sequences; then, at each round, step 4.f must be modified so that if the variable c that is associated with some neighbor is more than q, then this neighbor is a valid motif. Hence, Theorems 1-4 can be modified by increasing the time bounds by $n - q + 1$ factor; thereby, they are valid theorems for the qPMS problem as well.

The definition of the EDPMS problem is also similar to the PMS problem but with a

simple modification. The motif must be no more than d edit distance instead of Hamming distance with any l-mer in the input sequences. This can also be solved because the design of ET does support the edit distance, but with an increase of a factor of $O(3^d)$ in the costs of time and space. Therefore, the following modifications in Theorems 1-4 can be applied so that they can be valid for the EDPMS problem. An increase of $O(\sum_{j=0}^{d} 3^j \binom{l}{j})$ instead of $O(\sum_{j=0}^{d} \binom{l}{j})$ in the time bounds, and with a space bound of $O(\sum_{j=0}^{d} 3^d \binom{l}{j}(\Sigma-1)^j)$ instead of $O(\sum_{j=0}^{d} \binom{l}{j}(\Sigma-1)^j)$. For a balanced GST, the same increase in the space and time bounds is still applied, but by using $O(\binom{log_{\Sigma} nm}{j})$ combinations instead of $O(\binom{l}{j})$.

Chapter 4

HGA: De novo Genome Assembly Method for Bacterial Genomes Using High-coverage Short Sequencing Reads

A novel *hierarchical genome assembly* (HGA) methodology is introduced that is designed to take advantage of high-coverage reads. The method starts by independently assembling disjoint subsets of the sequencing reads, combining the assemblies of the subsets, and finally re-assembling the combined contigs along with the original reads. Empirical evaluation of this methodology for eight leading assemblers using seven GAGE-B bacterial datasets consisting of 100-bp Illumina HiSeq and 250-bp Illumina MiSeq reads shows that HGA leads to significant improvement in the assembly quality, based on N50 and corrected N50 metrics, for *all* evaluated datasets using *most* evaluated assemblers (those used to assemble the disjoint subsets).

4.1 Background

De novo genome assembly is one of the fundamental problems in bioinformatics. Interest in the problem has been renewed in the past decade because of the advent of *next-generation sequencing* (NGS) technologies, which generate a large numbers of short (100-400 bp) reads at relatively low sequencing error rates. There are three main approaches for *de novo* genome

assembly: the greedy strategy, the string overlap graph, and the de Bruijn graph. In the greedy approach, the assembly algorithm works by selecting seed reads and greedily extending them with maximum overlapping reads until no more overlap is possible. This was the approach adopted by some early assemblers such as SSAKE [90], SHARCGS [25], and VCAKE [40]. Unfortunately, the greedy approach does not take into account the ambiguities that are induced by repeats and sequencing errors, resulting in numerous misassembly errors.

The string overlap graph approach is based on building a graph with reads as nodes and edges connecting every pair of nodes if the corresponding reads overlap given a minimum overlap length. Building the overlap graph involves a computationally intensive all-against-all pairwise comparison step. After constructing the graph, the reads layout is computed and the consensus sequence determined using multiple sequence alignment. This approach, which was implemented in assemblers such as Newbler [52], SGA [80], and CABOG [54], is more efficient for long reads such as those generated by Sanger and 454 sequencing.

The third approach, based on the de Bruijn graph model [64], is by far the most common technique in assemblers that target NGS data, including ABySS [81], ALLPATHS-LG [33], Euler-USR [14], MaSuRCA [96], SoapDenovo2 [49], SPAdes [10], and Velvet [95]. Building the de Bruijn graph starts by collecting all substrings of length k (referred to as k-mers) of all reads, then building a graph with k-mers as nodes and edges that connect two k-mers a and b if the suffix of length $k-1$ of a matches the prefix of length $k-1$ of b and the $(k+1)$-mer obtained by overlapping a and b appears in the reads. The de Bruijn graph can be built in linear time, but requires massive amounts of memory to store, typically much

larger than the string overlap graph. After building the de Bruijn graph, each assembler uses heuristics to simplify graph structures such as cycles and bulges (induced by repeats in the genome) and bubbles and tips (created primarily by sequencing errors and heterozygous sites). Lastly, assemblers select a set of simple paths in the de Bruijn graph that would eventually form the contigs. For further details on algorithms for NGS genome assembly, the reader is directed to [55, 77, 78].

Despite the large number of assemblers that have been developed, genome assembly from NGS reads remains challenging. In particular, recent benchmarking efforts have shown that the performance of existing assemblers is highly variable between datasets and degrades appreciably with the complexity of the genome [41, 75]. For large genomes, the highest-quality assemblies are currently obtained by jointly assembling multiple paired-end libraries generated with a wide range of insert sizes using algorithms such as ALLPATHS-LG [33]. However, sequencing libraries with long insert sizes are expensive to generate. Recently, the GAGE-B study [51] showed that a comparable assembly quality could be obtained for bacterial genomes by running state-of-the-art assemblers such as MaSuRCA [96] and SPAdes [10] on a single short-insert library with very high coverage (100-300×). In contrast, other assemblers including ABySS [81], CABOG [54], SoapDenovo2 [49], SGA [80], and Velvet [95], appeared less able to take advantage of such high-sequencing depth, with nearly flat N50 contig length above 100× (see Fig. 1 in [51]).

Our contribution is to resolve the assembly problem using different approaches. The contribution is summarized in the following observations. First, in relation to tips, bubbles, bulges, cycles, and false branching, *the graph will be less complex* using lower coverage and

short k-mers than it would be with higher coverage and short k-mers. In addition, resolving these complexities will lead to more efficiency and produce fewer errors in the resulting contigs, although they will generally be shorter. Therefore, in order to resolve this, we first split the whole reads into several partitions to produce lower coverage in each partition.

Second, assembly using low coverage usually results in contigs that are shorter than those from high coverage, and it produces more gaps. In order to resolve this, we merge or assemble all the contigs that are produced from the assemblies of each partition along with all the reads again. This recovers any gaps that are due to the low-coverage assembly. Moreover, assembling contigs that are produced from all partitions adds support in selecting the common contigs (more likely to be true contigs) and in filtering out any redundant or false (partial or full) contigs. In addition, the more accurate the contigs that are inputted to the re-assembly step, the better the assembly that can be expected.

The nature of the process of partitioning the reads and then assembling the results, and the potential for applying several levels of partitioning especially for large genomes led us to the notion of **Hierarchical Genome Assembly**.

4.2 Methods

Our study used four Illumina MiSeq datasets and three Illumina HiSeq datasets that were used in the GAGE-B study [51].

Input data: The datasets are for four bacterial genomes. Table 4.1 describes the genomes and the datasets.

As referred by GAGE-B, the data can be downloaded from the Sequence Read Archive

Table 4.1: Descriptions of the bacterial genomes and sequence reads that were used. All data sets are paired-end reads.

Dataset	Genome size (Mb)	GC content (%)	Sequencing technology	Read length (bp)	Fragment length (bp)	Avg. coverage	# Proteins
Bacillus cereus ATCC 10987	5.4	35	MiSeq	250	600	100x	6,014
Mycobacterium abscessus 6G	5.1	64	HiSeq	100	335	115x	4,992
Mycobacterium abscessus 6G	5.1	64	MiSeq	250	335	100x	4,992
Rhodobacter sphaeroides 2.4.1	4.6	69	HisSeq	101	220	210x	4,474
Rhodobacter sphaeroides 2.4.1	4.6	69	MiSeq	251	540	100x	4,474
Vibrio cholerae CO1032(5)	4.0	48	HiSeq	100	335	110x	3,693
Vibrio cholerae CO1032(5)	4.0	48	MiSeq	250	335	100x	3,693

at NIH's National Center for Biotechnology Information using the following SRR accession numbers: *Rhodobacter sphaeroides*, MiSeq: SRR522246, HiSeq: SRR522244; *Mycobacterium abscessus*, MiSeq: SRR768269, HiSeq: SRR315382; *Vibrio cholerae*, MiSeq: SRR769320, HiSeq: SRR227312; *Bacillus cereus*, MiSeq data were downloaded from the Illumina website. Then, GAGE-B down-sampled the data to collect up to 250× coverage with HiSeq data and 100× coverage with MiSeq data. Next, the raw data were cleaned by removing adapter sequences and trimming the reads based on q10 quality. Both the raw (down-sampled) and the cleaned datasets are available at the GAGE-B website (`http://ccb.jhu.edu/gage_b`). These are the datasets that were considered in this study.

All tested genomes have multiple chromosomes, while some have plasmids as well. The genome of *V. cholerae* has two chromosomes, *B. cereus* and *M. abscessus* have one chromosome and one plasmid, and *R. sphaeroides* has two chromosomes and five plas-

mids. In order to compute the correctness of assemblies, the following strains were used as reference genomes: *B. cereus* ATCC 10987 (GenBank accession numbers NC_003909 and NC_005707); *M. abscessus* ATCC 19977 (accession numbers NC_010394 and NC_010397); *R. sphaeroides* 2.4.1 (accession numbers NC_007488, NC_007489, NC_007490, NC_007493, NC_007494, NC_009007 and NC_009008); *V. cholerae 01 biovar eltor str. N16961* (accession numbers NC_002505 and NC_002506).

Assemblers: We tested our method using the following eight open-source genome assemblers, which were also tested in GAGE-B:

Abyss v1.5.1, Cabog v7.0, Mira v4.0.2 [11], MaSuRCA v2.2.1, SGA v0.10.13, SoapDenovo2 v2.04, SPAdes v3.0.0, and Velvet v1.2.10.

To describe the methods, we employ metrics that are used in the QUAST tool [36], which is a commonly used and accurate tool for evaluating and analyzing assembly results. Namely, we use the following metrics: *Number of contigs*, *N50*, *NA50*, *NG50*, *NGA50*, *Genome fraction (%)*, *Duplication ratio*, *Global misassemblies*, *Local misassemblies*, *number mismatches per 100 kbp (MP100K)*, *number indels per 100 kbp (IP100K)*, and *number Unaligned length*. For the descriptions of these metrics, we refer the reader to QUAST [36]. The descriptions of the metrics are also included in the Appendix.

4.2.1 Hierarchical Genome Assembly

The HGA method involves the following steps. First, all reads are partitioned into p disjoint partitions, where $p > 1$. Then, each partition is assembled independently. After assembling all partitions sequentially or in parallel, assemblies of all partitions are assembled together

to form *combined contigs*, or merged together to form *merged contigs*. Lastly, the merged contigs or the combined contigs are re-assembled with the whole reads again. Fig. 4.1 depicts these steps diagrammatically.

This method will be compared mainly to the *basic assembly* process shown in Fig. 4.1, which involves assembling the whole reads together, then outputting the assembly results. We denote the basic assembly as **B(*k*-mer, c)**, where *k-mer* is the k-mer length used in the assembly and c is the coverage of the reads.

Partitioning step

The main motivation for partitioning the reads set into smaller partitions is to obtain a lower data coverage. Thereby, we expect to obtain a graph with less complexity, permitting more efficient resolution of the assembly's ambiguities. It is true that the results of higher coverage may produce longer contigs, but they are likely to contain more errors in terms of global and local misassemblies, MP100K, IP100K, and unaligned contigs. In order to show this experimentally, we added into Tables 3-9 (Appendix), a row presenting the average values over all the partitions for each metric, in order to compare those results with the basic flow results. The results show that, for most assemblers and most genomes, the average values of local misassemblies, global misassemblies, MP100K, and IP100K over all the partitions are less than the values for the basic flow (the flow that involves no partitioning of the reads dataset), especially for HiSeq datasets for which the coverage is higher and the reads are shorter (higher expectations of graph complexities).

For the steps of combining the contigs (contigs assembly) and the re-assembly, it is

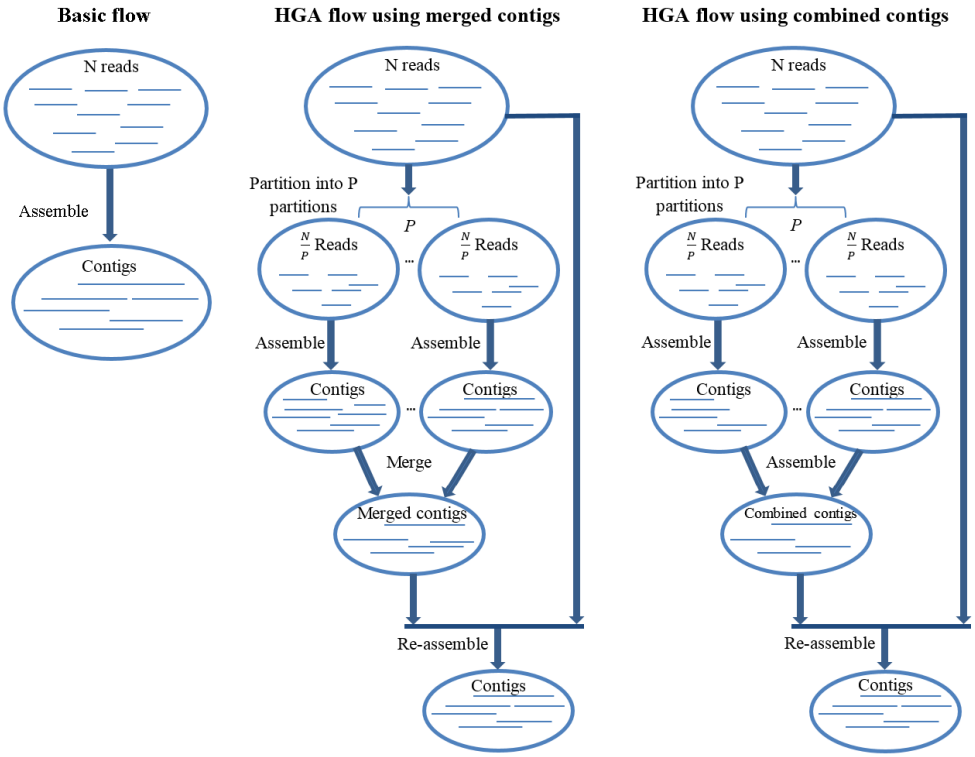


Fig. 4.1: HGA flows: Flow diagrams represent the basic assembly flow and hierarchical assembly flows. The basic flow represents the assembly of all reads in the dataset together. HGA flow using merged contigs represents the flow of partitioning the reads in the dataset into p disjoint partitions, then assembling each partition independently. Next, the contigs of each partition's assembly are merged together. Lastly, the merged contigs are re-assembled with the whole reads. The only difference between HGA flows using merged contigs and combined contigs is that the latter combines the contigs of all partitions' assemblies rather than merging them. We used Velvet to assemble the contigs.

critical that the contigs that are involved in these steps are longer and contain fewer errors in terms of local misassemblies, global misassemblies, MP100K, and IP100K. The results of these steps will be more efficient and accurate with contigs that are longer and more accurate.

The first step of the method is to partition the reads set by splitting the whole reads set (N reads) into p disjoint partitions. Each partition thus contains $\frac{N}{p}$ reads. After performing several experiments to determine the number of partitions to produce, we concluded that the constraint is to obtain a coverage for each partition greater than or equal to $10x$ rather than the number of partitions. Thus, in general, the partition's coverage is the only constraint to be considered for this step.

Contigs Assembly (combining) step

After assembling each partition, we merge the resulting assemblies together to form the merged contigs, or assemble (combine) them together to form the combined contigs. Initially, we assembled the contigs of the partitions using minimus2 [82], but after analysis of several experiments, we found that minimus2 is actually misleading in combining contigs. Despite the improvement in terms of NA50 of minimus2's results, there was significant increase in both duplication ratio and misassembly events. This occurs because the input data are not short reads, but long reads (contigs). Therefore, as minimus2 computes the pairwise alignment between all reads (contigs in this case), and if we have two or more contigs that are true (aligned) but not contiguous in the reference and they share an x-mer (which may be a repeat), then minimus2 will output these contigs as a single one. Moreover, one of these

contigs will most likely again share x-mer truly or falsely with other contigs, resulting in repeated outputting of the same contigs multiple times, eventually increasing the duplication ratio. In addition, assembling true and not-contiguous contigs will increase the number of global or local misassemblies. Hence, the improvement in the results of NA50 is mostly false positive.

It is clear that string graph assemblers such as SGA would not work effectively on assembling contigs. Some contigs may start overlapping in the middle of other contigs. This is not covered by string graphs by definition, as they compute the overlap at the ends of the reads (contigs, in this case). Therefore, we switched to assemblers that use the de Bruijn graph. Among all assemblers based on the de Bruijn graph and that take contigs as input data, we sought to determine which assembler has the best contigs-only assembly results. Velvet was found to be the best choice. Moreover, experimenting with different k-mer lengths in running Velvet to assemble the contigs, a k-mer value of 31 and setting the value of the expected coverage to be equal to the number of partitions, led to the best contigs-assembly results. The results of running Velvet as a contigs assembler and the comparisons with the results of minimus2 are provided in Table 11 (Appendix).

The results of combining contigs is denoted as **C(*k*-mer, p, c)**, where p is the number of partitions, c is the coverage of each partition, and k-mer is the k-mer length used in the assembly of each partition. Lastly, the contigs that result from merging the partitions' contigs are denoted as **M(*k*-mer, p, c)**.

Re-assembly step

In this step, the merged or combined contigs are re-assembled with all the reads again. To accomplish this, an assembler that takes long sequences (contigs) as an input is required. In addition, the assembler should preferably be based on de Bruijn graphs. For these reasons, SPAdes and Velvet were the convenient candidates. After testing both the candidates on two genomes, SPAdes produced better re-assembly results than Velvet. Details of the tests and results are provided in Tables 12 and 13 (Appendix). Hence, all re-assembly was performed using the SPAdes assembler. We denote this step as **HGA(*k*-mer, contigs)**, where *k-mer* is the k-mer length used in the reassembly process, and *contigs* refers to the merged or combined contigs.

In total, we have the following flows: B(k-mer), HGA preprocessing flows M(k-mer, p, c) and C(k-mer, p, c), and HGA re-assembly flows HGA(k-mer, M(k-mer, p, c)), and HGA(k-mer, C(k-mer, p, c)).

Re-assembling the reads with long sequences (contigs) has several advantages. First, these contigs were produced not by assembling all the reads together, but by the combined or merged contigs of the assembly of different partitions. Therefore, they are more accurate and refined in terms of errors, are considerably longer than the reads, and are not redundant (meaning that they are not produced from the assembly of the *same* reads as one partition; instead, they are assembled from several disjoint subsets, and are hence structurally different). In addition, we found that if contigs, which are assembled from a dataset, are re-assembled again with the same dataset then the final assembly's results will not improve and will even deteriorate. Secondly, re-assembling long sequences (contigs) may result in connect-

ing different connected components in the graph. Finally, during the path-finding process, the re-assembly process may increase the chances of selecting the true paths by traversing the longest path (resulting from the inclusion of long sequences in the input data).

To further explore and justify the advantages of the re-assembly step, we performed a simple test on a real dataset of *M. abscessus* bacteria (the dataset that was used in this study and which was described in the previous section). We assembled the real HiSeq dataset of *M. abscessus*, along with the genome of *M. abscessus* itself, using the SPAdes assembler and a range of k-mer lengths of 21, 31,..., 91. The NA50 result of the assembly at $k = 91$ was 99% of the length of the genome of *M. abscessus*. This indicates that the assembly of contigs with the reads would indeed be computationally effective and could lead to an optimal assembly. However, the more accurate these contigs, the better the re-assembly results.

4.3 Results

Before presenting the results, it must be noted that, some assemblers, namely, Abyss, Mira, MaSuRCA, SGA, SPAdes, and Velvet, are newer versions than those used in GAGE-B. For these assemblers, we did the following. Firstly, the assemblies' results reported in GAGE-B were compared. Secondly, as these assemblies were obtained using older versions, we ran the basic flow using the new versions of all assemblers employing k-mer lengths (or overlap lengths for SGA) of 21, 31,..., 91 for HiSeq datasets, and 21, 31,..., 101 for MiSeq datasets. Then, we reported the assembly that has the highest N50. Moreover, MIRA and CABOG assemblers do not take k-mer or overlap values as parameters, so their single assembly is reported rather than the maximum assembly results over the k-mer (or overlap value) range.

Each dataset was assembled by each assembler as paired-end reads. We followed the run commands for each assembler that were provided in GAGE-B's supplementary document, which are also in the Appendix. For the basic assembly flow, we used k-mer lengths (or overlap lengths for SGA) of 21, 31,..., 91 for HiSeq data sets, and 21, 31,..., 101 for MiSeq data sets for all assemblers, except for MIRA and CABOG. For HGA flows, we split the reads into 2, 4, and 8 partitions. We did not split the reads more than that because the coverage of each partition would be too low (less than $10x$) to assemble. Then, we assembled each partition independently using the same k-mers set that was used in the basic flow (note that this step can be performed in parallel). For instance, we split the reads data sets into 4 partitions, then assembled each partition using k-mer lengths of 21, 31,..., 91 for HiSeq data sets, and 21, 31,..., 101 for MiSeq data sets.

During the next step of combining the contigs, we first merged the contigs of all partitions together to form the merged contigs. In addition, we combined them to form the combined contigs using Velvet with a k-mer length of 31, and specifying to Velvet the expected coverage to be the number of partitions. For the re-assembly step, we used the SPAdes assembler.

For experimental purposes and for the HiSeq datasets, the following combinations exist: (1, 21), (1, 31),..., (1, 91); (2, 21), (2, 31),..., (2, 91); (4, 21), (4, 31),..., (4, 91); (8, 21), (8, 31),..., (8, 91). For each combination, we run the re-assembly step using k-mer lengths of 21, 31,..., 91 with the merged or combined contigs that were produced from the combination. For the MiSeq datasets, there are the following combinations: (1, 21), (1, 31),..., (1, 101); (2, 21), (2, 31),..., (2, 101); (4, 21), (4, 31),..., (4, 101); (8, 21), (8, 31),..., (8, 101). For

each combination, we run the re-assembly step using k-mer lengths of 21, 31,..., 101 with the merged or combined contigs that were produced from the combination.

4.3.1 Assembly results

Figure 4.2 shows the results of the assemblies using both flows of the HGA method compared to GAGE-B's best-reported results. We compared the results based on the corrected N50 (NA50) correspondent to the *highest N50* result, which is a common metric that has been used to evaluate the accuracy of the assembly outputs for *de novo* genome assembly. In addition, GAGE-B reported the evaluation results of their assemblies, but as QUAST had a newer version and as the genes GFF files may be updated, we downloaded the assemblies from the GAGE-B website and assessed them along with the assemblies of HGA using the same versions of QUAST and the latest gene GFF files, for consistency.

The HGA method has two main flows: HGA(k-mer, M(k-mer, p, c)) and HGA(k-mer, C(k-mer, p, c)), as illustrated in Fig. 4.2. We abbreviate them as HGA(merged) and HGA(combined), respectively. The full results of the main flows, preprocessing flows, and GAGE-B results for the different metrics are available in Tables 3-9 (Appendix).

Note that the results using the same assembler always improve using the HGA method. In addition, the maximum result of HGA flows across different genomes is larger than the maximum of GAGE-B's reported results. Of the following assemblers, SPAdes, MaSuRCA, and Velvet generally produce the best HGA assembly. We can see that the results using HGA were improved significantly for both HiSeq and MiSeq datasets. For *R. sphaeroides* HiSeq datasets, it improved from 177 Kbp (the maximum of GAGE-B results) to 315 Kbp

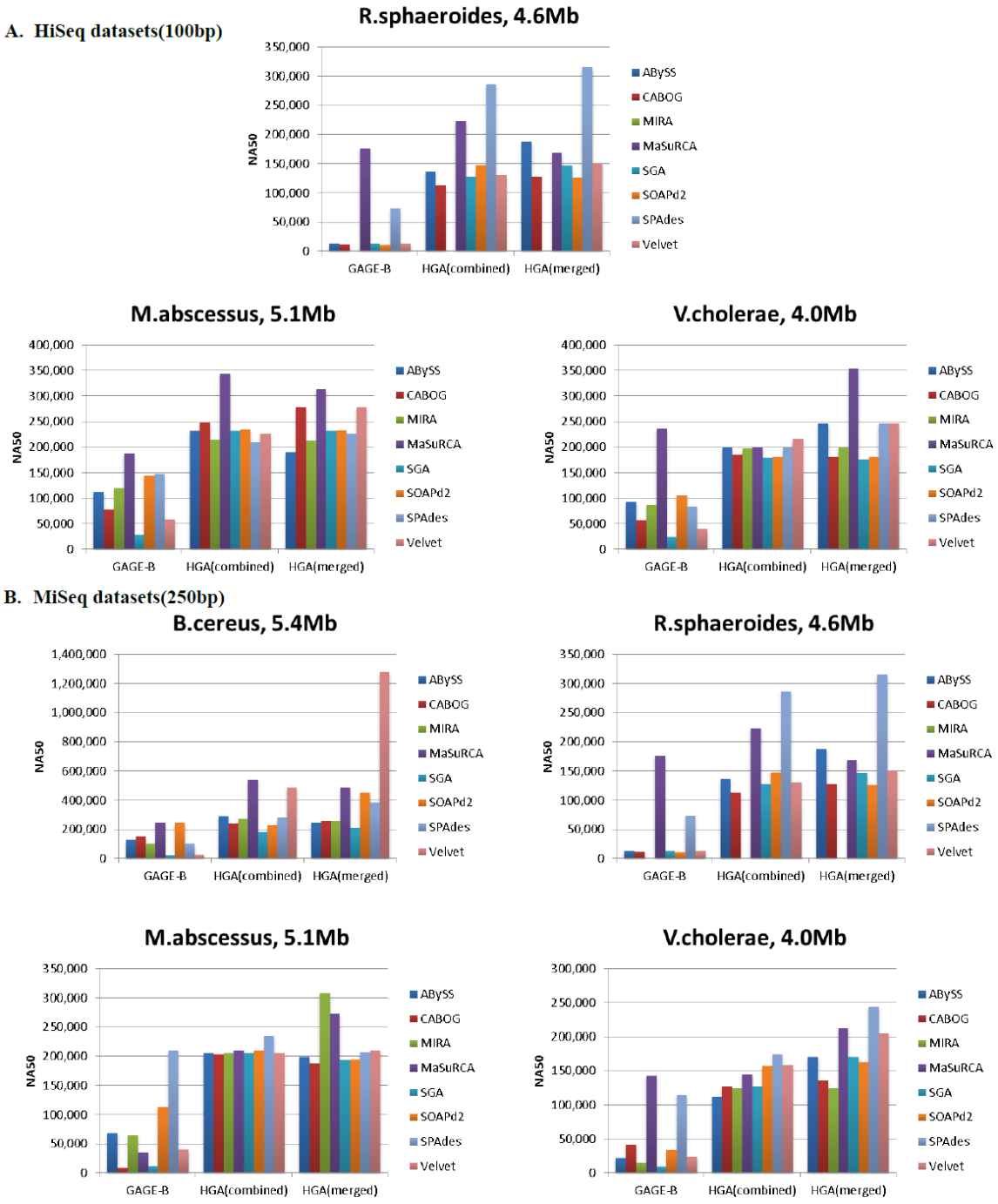


Fig. 4.2: NA50 Results: NA50 (corrected N50) corresponding to the assembly with the highest N50 results for GAGE-B and HGA assemblies.

(the maximum of HGA results). Likewise, it improved from 189 Kbp to 344 Kbp for *M. abscessus* and 236 Kbp to 354 Kbp for *V. cholerae*.

The results of the MiSeq datasets showed a considerable improvement. *B. cereus* results improved from 247 Kbp to 1,276 Kbp, and *R. sphaeroides* from 143 Kbp to 246 Kbp. The improvement for *M. abscessus* was from 210 Kbp to 309 Kbp. Moreover, *V. cholerae* results increased from 247 Kbp to 356 Kbp. Lastly, as a result of these improvements, the number of positively called proteins did increase, as shown in Fig. 1 (Appendix).

The way in which the HGA method contributes to improving the assembly process is described as follows. Assembling the whole reads together using a long k-mer length, which is effective in resolving repeats in the genome, will decrease the overlapping between the reads due to errors in the reads. Hence, there will be more connected components in the graph and eventually the resultant contigs will be shorter. Now, let us assume that such an assembly involves correct contigs (with no errors). Then, firstly, such contigs will connect different connected components. Secondly, reads will overlap with such contigs. Thirdly, at a branching point, the branch that is connected with contigs will have more weight. This will aid assemblers in selecting the true path during the path-finding process. In conclusion, this will improve the overall assembly process firstly by assembling repeat regions more accurately and secondly by producing longer contigs. The problem now is to produce contigs that are as accurate as possible. Furthermore, the process is also true in reverse, meaning that if the contigs are not highly accurate but are assembled from repeat regions, then assembling such contigs with the whole reads using short k-mer lengths will also improve the assembly results, as shown in Tables 3-9 (Appendix). This is because repeats were already resolved in

the inputted contigs and performing the assembly using short k-mer lengths is more feasible for correcting the errors in the reads or contigs.

In order to produce contigs that are more accurate, note that without even partitioning the whole reads, assembling the whole reads using a short k-mer and then re-assembling the resultant contigs with the whole reads using a long k-mer leads to an improvement in the assembly results. As an example, Table 5 (Appendix) shows that the result of assembling one partition (the whole reads) using a k-mer length of 21, and then re-assembling the resultant contigs with the whole reads using a k-mer length of 41 was better than 2, 4, and 8 partitions, and was even the highest compared to all other methods and assemblers. Moreover, besides the fact that a short k-mer length is better in correcting the errors in the reads, we observed experimentally (Tables 3-9 (Appendix)) that contigs that were produced from low coverage contained fewer errors in terms of local misassemblies, global misassemblies, MP100K, and IP100K in most experiments. Therefore, we decided to apply such an approach as an attempt, along with using short k-mers, to produce more corrected contigs to be inputted to the re-assembly step.

Partitioning the whole reads has two main advantages. First, when building a graph using a set of reads, all errors in all reads will create error-related complexities (such as false branching, tips, and bubbles) in the graph. While all errors in the reads will still induce error-related complexities, when building a graph using, for instance x partitions of the reads, those errors will be distributed into x graphs. Hence, in each of the x graphs, we will have less error-related complexity compared to the graph that is built using the whole reads set. This will aid the assembler in applying the algorithms to resolve more feasibly the error-

related complexities in each graph, and ultimately contigs that result from all partitions will contain fewer errors. However, note that the complexities that are usually induced by, for example, repeats or SNPs will still be present.

Second, with the assembly of a single partition, assemblers are compelled to produce contigs of coverage of $1\times$, in order to avoid redundancy in the assembly results. Therefore, at a branching point (a node in the graph with indegree $= 1$ and outdegree ≥ 1), assemblers must traverse a single and *best* possible branch, given that it might be heuristically false positive (hence produce a false contig). Now, when partitioning the whole reads into multiple partitions, the resultant contigs of all partitions' assemblies will initially be of coverage $P\times$ and not only $1\times$ where P is the number of partitions. Then, it is highly possible that, at the same branching point, different branches will be traversed in the assembly of different partitions. Hence, as more branches will be traversed, the true branch is more likely to be traversed, resulting in a true contig. Thus, the resultant contigs of all partitions will be more likely to contain true contigs. This latter analysis also explains the reason that the re-assembly process using merged contigs ($P\times$ of converge) showed better results compared with both the combined contigs ($1\times$ coverage) and the re-assembly results using the contigs that were produced from one partition.

4.3.2 Multi-*k*-mers assembly

Some assemblers use multi-k-mers for assembly such as SPAdes. The results of SPAdes that were provided by GAGE-B were already *optimized* over multiple k-mers. Moreover, we executed SPAdes using two k-mers: one in the preprocessing step and another in the

re-assembly step. This was performed to test for improvement using multi-k-mer assembly using the same k-mers that were used in the HGA method. Some results showed slight improvements over the single k-mer assembly, but were not even close to the improvements observed for the HGA methods. Some results were even worse; this can be explained by the fact that the complexity of the de Bruijn graph using multiple k-mers would not be reduced, and may even be increased compared to the complexity of the de Bruijn graph of a single k-mer. In addition, it is hard to find the combination of k-mer lengths that will lead to the best assembly, as this process involves costly enumeration and running trials for each tested combination. The HGA methods utilize multi-k-mers not *collectivity* but in a phased manner.

4.3.3 Impacts of contig correctness and length in the re-assembly process

The correctness and the length of the contigs that are re-assembled with the whole reads are critical in improving the HGA results. As an example, Tables 3-9 (Appendix) show that MaSuRCA HiSeq assemblies are among the highest in terms of MP100K compared to the other assemblers, but, in terms of local and global misassemblies, MaSuRCA is one of the assemblers that show the lowest results. Moreover, as an evaluation for the re-assembly step, lower MP100K and IP100K would produce better results than lower local or global misassemblies, since MP100K and IP100K are measured per 100 kbp. Therefore, a genome of length 5 Mbp and MP100K = 5 will induce 250 mismatches, whereas local or global misassembly values are usually in the tens on average. As for the MP100K for MaSuRCA's HiSeq assembly, basic flow of *R. sphaeroides* it is 47.5 ($\sim$2,090 mismatches = genome length

(Mbp) × 10 × genome fraction of the assembly × MP100K), for *M. abscessus* it is 21.2 (1,080), and for *V. cholerae* it is 23.8 (930). Such a large number of mismatches in the contigs will induce fewer overlaps between the reads and these contigs during the re-assembly process. These values were reduced by the partition step to 22.8 (1,000), 6.7 (340), and 14.8 (570), respectively, and were further reduced/increased in the combining step to 18.3 (800), 5.6 (280), and 17.2 (670), respectively. This explains why the re-assembly process was better using the combined contigs rather than the merged contigs for the results of HGA using the MaSuRCA assembler. The exception was for *V. cholerae*, which correlates with the fact that the combining step did not decrease the errors but actually increased them. It is worth mentioning that, although the HiSeq contigs (combined and merged) by MaSuRCA showed more errors in terms of MP100K, these contigs were more contiguous (longer) compared to those resulting from other assemblers. The longer the contigs that are re-assembled using the HGA method, the better the HGA results. The results of HGA using MaSuRCA were among the best for MiSeq assemblies and the best for two HiSeq assemblies, *M. abscessus* and *V. cholerae*.

In conclusion, this analysis suggests that resolving the causes of MP100K and IP100K more carefully at least during the steps before the re-assembly process (namely the assembly of the partitions and/or combining the contigs), would improve the results of the re-assembly step, and hence the overall assembly results.

4.3.4 Testing error-free reads

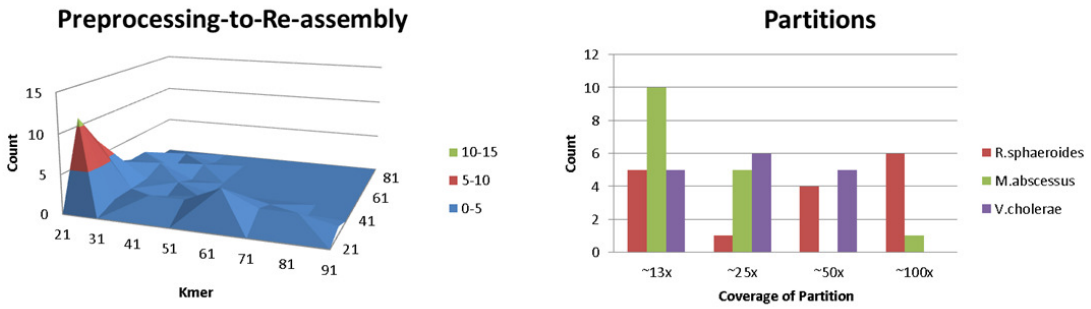
As noted in the discussion section, the contigs in the partitioning and combining steps were expected to obtain as much correction/refining as possible before being re-assembled again with all reads. This explains the improvement in the HGA results. For a further justification, reads with no errors were simulated from the *M. abscessus* genome with the same coverage as the real dataset, and then HGA methods were run on those reads. The assembly results of the HGA flow of combining contigs and HGA using combined contigs showed no improvement over the basic flow. This was not the case with the real dataset, which involved errors in the reads and for which the improvements were more significant.

This indicates that the HGA methods were able to correct more errors in the reads than in the basic flow. Moreover, even with error-free reads, HGA using merged contigs showed an improvement over all the other HGA methods and the basic flow. This again supports the point that assembling contigs that are high in coverage ($P\times$ of coverage) with the whole reads is better than combining the contigs (which produces a coverage of $1\times$) first, then assembling the combined contigs with the whole reads. The test results are provided in Table 10 (Appendix).

4.3.5 Partition-*k*-mer to re-assembly-*k*-mer relations

As the HGA method involves two assembly steps (preprocessing and re-assembly), each step uses the same or different k-mer lengths. Moreover, as we tested all combinations of the two k-mers and selected the assemblies with the highest N50 results (eight highest assemblies for eight assemblers for both combined and merged flows). In Fig. 4.3, out of these 16

A. HiSeq datasets(100bp)



B. MiSeq datasets(250bp)

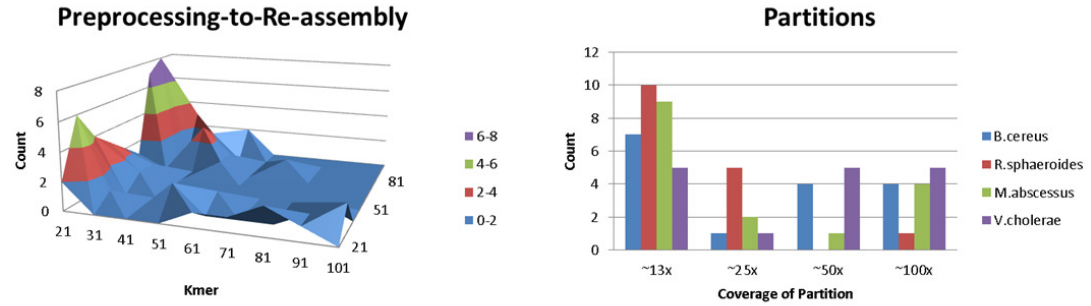


Fig. 4.3: Partition-k-mer to Re-assembly-k-mer Relations: As there are 16 highest results (two flows for eight assemblers), we built two plots. The first plot is a surface chart that shows the count of combinations of k-mers (preprocessing k-mer to re-assembly k-mer) that were applied by the 16 highest results. The second plot shows the counts of partitions that were applied by the 16 highest results.

assemblies, we plot how many times each k-mer was used during the preprocessing and re-assembly steps in order to observe any correlations.

It was observed that the best assembly results on MiSeq data were obtained when short k-mers (21 and 31) were used in the preprocessing step and long k-mers (61, 71, and 81) were used in the re-assembly step. For HiSeq data, short k-mers (21 and 31) in the preprocessing step and medium k-mers (31, 41, and 51) in the re-assembly step yielded better assembly results.

Fig. 4.3 shows the plots of the k-mer lengths that were used in the preprocessing and re-assembly steps, as well as how many times each k-mer was observed. We note that 21 and 31 were dominant during the preprocessing step. This can be explained by the fact that the assembly process using short k-mer lengths leads to more accurate contigs that are inputs to the next step (re-assembly step). This again emphasizes that the more the accurate the contigs, the better the assembly results. For the re-assembly step, medium-to-long k-mers were dominant. This can be explained because, with such lengths of k-mers, resolving the complexities that are induced from repeats is prior, especially when the corrected contigs collaborate in reducing the complexities/computations that are usually induced from errors. Therefore, the assemblies using such a combination were mostly the highest in terms of N50 results.

Adopting a short k-mer as a stereotype in the preprocessing step and a medium-to-long k-mer in the re-assembly step might help in eliminating the common confusion of which k-mer length (short or long) to apply for assembly. In addition, it might help in avoiding the need for extra steps; for example, running preprocessing tools such as K-merGenie [19]

to find the best k-mer length to use in the assembly process. This process involves trying several assembly trials, and then selecting the best assembly results. In addition, avoiding running an assembly using multiple k-mers (which costs more space and time) and finding the best k-mer set is a difficult process because many combinations need to be tested.

Lastly, we plot the most frequent number of partitions used among the highest assembly results, as shown in Fig. 4.3. We conclude that more partitioning (less coverage) was most frequent. This can be explained, again, by the fact that low-coverage assembly results in a graph that is less complex, hence more corrected contigs are inputted into the re-assembly step.

4.4 Discussion

Firstly, we designate the following notations: R as the length of the reference genome, L as the length of the input reads, C as the expected genome coverage by the reads, K as the length of the k-mer, and *Ideal case* as the case in which there are no errors in the sequencing reads, the length of any repeat in the reference genome is greater than L, and the reads were sequenced uniformly.

Analysis 1: In the ideal case when C is greater than or equal to 100×, for instance, building the de Bruijn graph using short or long k-mers will lead to the same and optimal assembly. The only difference is that the assembly using large k-mers results in faster and more memory-efficient assembly. For a non-ideal case, reads contain errors that induce tips, bubbles, or false branching. In the genome, repeats shorter than L will induce cycles, bulges, false branching, tips, and bubbles. Non-uniform coverage results in varying k-mer counts and

reduces the overlapping between the reads, thus creating gaps and increasing the number of components in the graph.

There is a tradeoff between using long or short k-mers. Long k-mers lead to fewer false branching, better resolution of repeats, and fewer cycles. However, this allows less error correction and detection and less overlapping between the reads (more components and gaps). In contrast, short k-mers output the opposite, namely, more false branching, more cycles, and lower repeats resolution, but more effective detection and correction of errors and fewer components and gaps in the graph.

Analysis 2: When the reads are of a high coverage, there is less chance of missing some regions of the genome, and the contigs resulting from these reads will be longer. However, the graph will be more complex than one built from low-coverage reads. Hence, the algorithms for error-correction and path-finding that are applied by the assemblers will be less effective at a high coverage. This is the main motivation behind partitioning the whole reads to low-coverage sets. In order to resolve the effects of having more gaps in the assemblies of the partitions compared to the assembly of the whole reads, the contigs are re-assembled with the whole reads, thus utilizing the high-coverage of the original whole-reads set.

4.5 Availability and requirements

Project name: HGA v1.0.0. '' **Project access:** github.com/aalokaily/Hierarchical-Genome-Assembly-HGA.

Operating system: Tested on Linux/Ubuntu OS, AMD Opteron(tm) 2.4GH, 256GB Memory, 64 cores. Programming language: Python 2.7.6.

Other requirements: SPAdes v3.0.0 , Velvet v1.2.10. Velvet must be installed as indicated in Appendix.

License: GNU GENERAL PUBLIC LICENSE.

Any restrictions to use by non-academics: None.

List of abbreviations

HGA: Hierarchical Genome Assembly. NGS: next-generation sequencing. MP100K: number of mismatches per 100 kbp. IP100K: number of indels per 100 kbp.

Additional Files

Appendix — supplementary tables, experiments, and descriptions

Appendix contains detailed results of the assemblies of the genomes, the assemblers that were used in this research, the results of experiments that were performed to explain the method, and the improvements showed in the manuscript, descriptions of the datasets, the metrics that were considered for evaluating the results, and the run commands that were used for each assembler.

Chapter 5

Toward a Better Compression for DNA Sequences Using Huffman Encoding

Due to the uniform distribution of the bases in DNA sequences, Huffman encoding is not the most efficient encoding method to compress such sequences [76]. The focus therefore of this chapter is on how Huffman encoding can be customized for DNA sequences for a better compression ratio. For this, we propose an implementation of Huffman encoding that incorporates the nature of the repeats that occur in such sequences. The selection allows us to control the topology of the binary tree, skewed in our implementation, in order to produce bit-codes that yield a better compression ratio.

5.1 Background

Data compression, in a nutshell, is the process of encoding information using fewer bits than the original representation. The process helps to reduce the consumption of valuable resources such as storage space and transmission bandwidth. Data compression has been an active research topic for long, and a number of algorithms have been proposed for different types of data such as texts, images, videos, audio and others. In general, there are two types of compression: *lossy* and *lossless*. The lossy scenario may exclude some unnecessary data

during the compression such as frequencies in audio data that a human cannot hear. The lossless algorithms, in contrast, ensure the exact restoration of information and do not allow any loss in data. Several lossless tools for text compression exist in the literature such as the *Lempel-Ziv* (*LZ*), *Lempel-Ziv-Welch* (*LZW*), *gzip* [31], and *bzip2* [79]. These tools work well for texts with large alphabet sizes (such as English alphabet), and on the other hand, are not tailored toward compressing biological sequences, which constitute a relatively small alphabet size, and some specific biological characteristics such as repeats.

Taking DNA sequences for example. DNA sequences are composed of just four letters (bases) A, C, G, and T. In addition, there are repeats in different forms within a DNA sequence which have important biological implications. These repeats, often known as *motifs*, appear in the sequences with a much higher frequency statistically. DNA sequences can have repeats within chromosomes, the genome itself, or genomes from different species. The number of repetitions in a set of sequences depends on the amount of similarity between the sequences. These repeats usually happen in the noncoding regions of the genomes. They are likely to occur in genomes of individuals of the same species. The genomes that are members of the same species are largely similar; it usually reaches 99.5% repetition.

A few algorithms that are specifically designed to compress DNA sequences have recently been proposed. As opposed to general-purposed compression algorithms, these DNA sequence compression algorithms are referred as *special-purpose* ones and have been categorized as *Horizontal*-mode and *Vertical*-mode compression [34]. Horizontal-mode compresses the DNA sequence using the information in that sequence. Methods falling within this category include those that are *substitution*-based, *statistics*-based, *two-bit* based, and

grammar-based. In contrast, Vertical-mode compression compresses a set of DNA sequences using the information from the entire dataset, usually by referencing substrings to a different substring from the dataset. In general, Vertical-mode compression performs better for large DNA datasets [8].

For substitution-based methods, the DNA sequence is compressed by replacing a long DNA string with a smaller one that is mapped back to the original one when decoded. *BioCompress* is a representative implementation [34]. A subsequent version, the BioCompress2, further adopts the *order-2* arithmetic coding to encode nonrepeat regions of DNA sequences [35]. The reported results indicate a compression ratio of 19.50% by BioCompress and 22.22% by BioCompress2, compared with the 25% by a general-purpose tool [34].

Similar to BioCompress, the *Cfact* uses a two-pass process to look for the longest exact and reverse complement repeats [72]. To do that, the algorithm at first builds the suffix tree of the sequence then encodes the sequence using LZ in the second pass. The nonrepeat regions are encoded by using two bits per base. The *GenCompress* makes use of approximate repetitions [16]. In this algorithm, an inexact repeat subsequence is encoded by a pair of integers and a list of edit operations for the mutations in those repeats. The algorithm has two versions: *GenCompress-1*, which uses the Hamming distance (to substitute) for the repeats; and *GenCompress-2*, which uses the edit distance which involves the operations of deletion, insertion, and substitution when encoding the repeats. The compression result of GenCompress is within an average of 21.77%; however, it does not perform well when compressing large sequences.

Furthermore, another tool *DNACompress* improves the results of GenCompress [17].

The tool runs in two passes. Firstly, all approximate repeats along with their complementary palindromes are found; this step is carried out by the *PatternHunter* software [50]. After that, both of the approximate repeats and the nonrepeat regions are encoded by the LZ compression technique. The result of DNACompress indicates a better compression ratio with an average of 21.56%. The tool is able to handle larger sequences in less time compared to GenCompress.

Algorithms based on statistical methods usually work on replacing the most frequent symbols by a shorter code. The CDNA algorithm is the first to combine statistical compression with approximate repeats [47]. Other statistical DNA compressors include the ARM algorithm [7] and the XM algorithm [13].

Grammar-based compression methods use a context-free grammar to represent the input text. After that, the grammar is transformed into a symbol stream to be finally encoded in binary. An example of a grammar-based compression algorithm is the *DNASequitur* [18]. This algorithm works by inferring a context-free grammar to represent the sequence as an input. The results do not show any better compression ratio compared to the other methods. Algorithms that adopt the two-bit based method, which assigns 2 bit for each base of the four DNA bases before the encoding process, also exist [73].

5.1.1 Approach

New technologies for producing high definition audio, videos, and images are growing at a fast pace and this has resulted in the generation of massive amounts of data that can easily exhaust storage and transmission bandwidths. To use the available disk and network

resources more efficiently, the raw data in those large files are often compressed. The Huffman algorithm is one of the most commonly used algorithms in data compression [38]. Huffman encoding is a lossless encoding algorithm that generates variable length codes for symbols that are used to represent information. By encoding high frequency symbols with shorter codes and low frequency symbols with longer codes, the original information is encoded as small as possible. The codes are constructed in such a way that no code is a prefix of any other code, a property that enables unambiguous decoding. While it was proposed half a century ago, the Huffman algorithm and its variants are still actively used in the development of new compression methods [32, 62, 48, 2].

Table 5.1: Base frequency and new bit representation.

Base	Frequency	The new representation	The old representation
A	9	0	01000001
G	5	10	01000111
C	4	110	01000011
T	3	111	01010100

Given the DNA sequence "ACTGAACGATCAGTACAGAAG", for example, which contains twenty-one bases, its bit count is therefore $21 \times 8 = 168$ bits, assuming each base is stored with an 8-bit ASCII code, where A = 01000001, C = 01000011, G = 01000111, and T = 01010100.

The compression by Huffman method works as follows. The algorithm at first counts the frequency of each base, as described in Table 5.1. Each base, along with its associated frequency, is to be regarded as a *tree* node. Mark the four initial nodes as unprocessed.

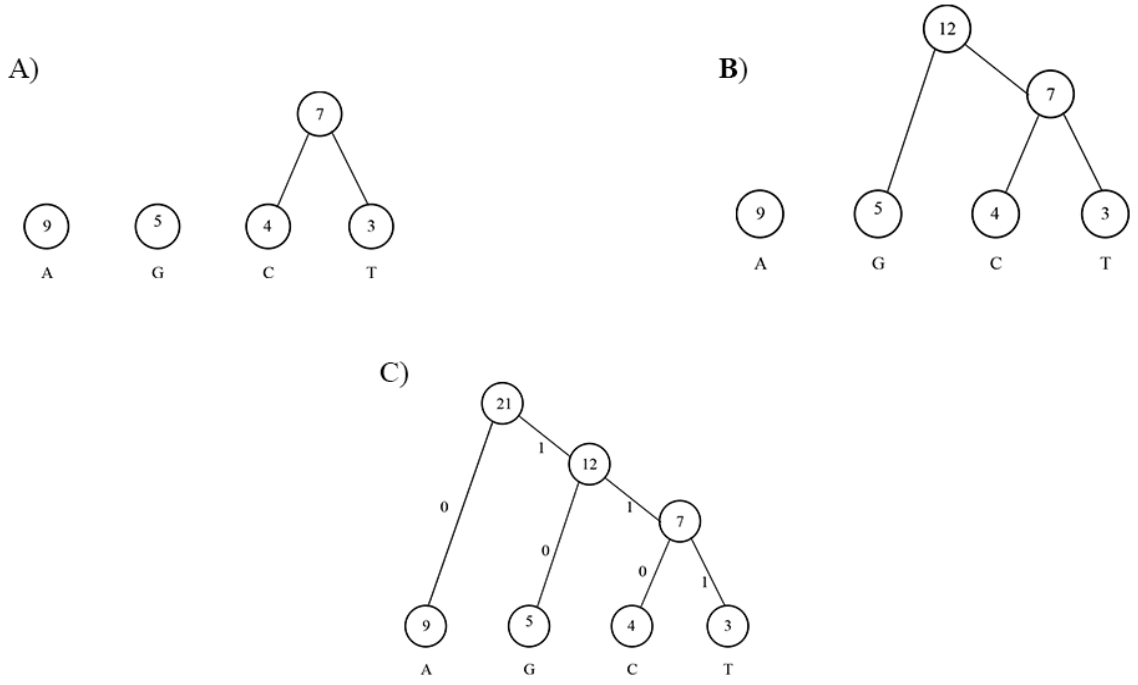


Fig. 5.1: The steps of constructing a Huffman tree.

Then, a binary tree is constructed by iterating the following steps. Firstly, search the two unprocessed nodes with the lowest frequencies. Then, construct a parent node whose frequency is the sum of those of the two child nodes. Lastly, return the parent node to the unprocessed list of nodes. The tree construction process ends when all nodes are processed. The construction of the Huffman tree for the given example is illustrated in Fig. 5.1. Note that, the left link of each internal node is labeled 0, while the right link is labeled 1.

After building the Huffman tree, each symbol has a new bit representation that can be extracted by traversing the edges and recording the associated bits from the tree's root to the desired symbol (which must be located at the leaf level). For example, the path from the root to base A results in a bit representation of 0, while the path from the root to base

T results in 111. The new bit representation for each base is described in Table 5.1.

The new bit count after encoding is (9×1)+(4×3)+(5×2)+(3×3)=40, with a compression ratio of 40/168=23.8%.

Typical algorithms extending from the concept of Huffman encoding when approaching DNA sequence compression include the *G-SQZ* [86] and the *Huffbit* [71]. During the first scan, G-SQZ encoding calculates *frequency for each base + quality pair*, and constructs a Huffman tree to generate pair-specific codes. High-frequency pairs have shorter codes. During a second scan, a header and encoded read-blocks is written to an output binary file. Each read block consists of read identifier fields followed by encoded base-quality data. The Huffbit compresses both repetitive and nonrepetitive regions in a DNA sequence. This algorithm first constructs an *extended binary tree*, and then derives the Huffman codes from the newly created extended binary tree. The authors claim a compression ratio of 12%, 20%, and 26% in the best, average, and worst-case scenarios, respectively.

Also extended from the Huffman encoding, this research aims to incorporate the characteristics of DNA sequences when constructing the binary tree, in order to derive bit-codes that allow for a better compression ratio than the conventional Huffman encoding. Hence, this research suggests an improvement for all DNA sequence compression algorithms that employ the conventional Huffman encoding.

5.2 Methods

5.2.1 The algorithm in general

First, the general approach of compression is to find the most repetitive substrings with a length up to a maximum value of k. Note k is empirically determined. Then, encode the resultant substrings using the bit-codes. Specifically, given a set of DNA sequences G and an integer k, construct the set of substrings S, where $\forall s \in S, |s| \leq k$, so that encoding all members of S results in the best compression ratio.

The algorithm, in general, works as follows.

1. Perform *Run Length Encoding* (RLE) on the genome to encode homopolymers (*i.e.,* sequences of identical bases). In brief, RLE for the sequence DDDDDDD is D7, where 7 is the count for the RLE and D is the base of the homopolymer. Some counts may appear more frequently than others. Therefore, the frequencies of the counts are collected as well. Let R be the set of tuples of the counts and their frequencies.

2. Scan G and collect the frequency of each substring of length no more than k. Let A be the set of all collected substrings.

3. Find the set of substrings S, so that by encoding this set the best compression ratio is reached. This is the major step of the algorithm. For this, a few different encoding schemes based on the Huffman encoding are designed and discussed in the following subsections.

4. Genome files may contain non-DNA bases. These bases need to be encoded into bits too. For this, scan G and collect the frequency of any non-DNA base. Let N be the

set of all collected bases. This, along with Step 2, can be merged as one step.

5. Input S, R (collected in Step 1), and N to the Huffman algorithm. The output will be the bit-codes that best represent the members of the three sets.

6. Scan G and encode each substring in G with its Huffman code. This will convert the DNA sequence to an (encoded) bit string.

7. Convert the bit string into ASCII codes to produce the compressed DNA sequence. This step involves adding headers of the words (members of the three sets) that were encoded and their bit-codes, as well as some other headers. These headers will be used during the decompression process.

5.2.2 Unbalanced Huffman tree

For Step 3, we propose a new implementation of the Huffman encoding called *unbalanced Huffman encoding/tree (UHT)*. For DNA encoding, forcing the Huffman tree to be unbalanced is a better approach than the *standard Huffman tree* (*SHT*), because when the UHT method is applied, three bases can be guaranteed to be encoded using two bits and the fourth using three bits. This, however, cannot be done by the SHT method; as encoding the best 20 k-mers (substrings of length no more than k), for instance, will encode each of the four bases with more than 2 bits.

Forcing the Huffman tree to be unbalanced can be achieved as follows.

1. Sort all collected k-mers, where $k \geq 2$, based on their frequencies.

2. Select the k-mer with the maximum frequency, say f.

3. Scan all k-mers with frequencies lower than f in a descending order until a k-mer with a frequency lower than $\frac{f}{2}$ is found. Select this k-mer and set f to be equal to the frequency of the selected k-mer. Note that, no selected k-mer is properly contained in another. This allows reducing overlap among the selected k-mers; hence, increasing the number of bases to be compressed.

4. Repeat steps 2 and 3 until no more k-mers can be selected.

5. Finally, input the k-mers (which were selected above) in addition to the four bases, to the Huffman encoding algorithm.

5.2.3 UHT that prioritizes encoding the *k*-mers that contain the least frequent base

Using the UHT method, the base with the lowest frequency, x, will be encoded using 3 bits, whereas the other bases will be encoded using 2 bits each. As a result, encoding the k-mers ($k \geq 2$), which contain at least one occurrence of x will be more feasible. For this, Step 1 is modified to allow *only* the frequencies of the k-mers, that contain x at least once, to be collected. We denote this method as *UHTL* (UHT that prioritizes encoding the k-mers that contain the *L*east frequent base).

5.2.4 Multiple SHT/UHT/UHTL encoding

Note that, a single SHT, UHT, or UHTL can encode few k-mers; however, they might be insufficient. Moreover, in the UHT/UHTL approach, there is no control on the number of k-mers to be encoded as the design is restricted to select k-mers that form an unbal-

anced Huffman tree, regardless of the number of such k-mers. One way to allow more k-mers to be encoded is to apply multiple encoding. We denote this approach as *Multiple SHT/UHT/UHTL* encoding, *MSHT/MUHT/MUHTL*, respectively.

To apply multiple Huffman encoding, multiple markers can be identified/implanted in the genome sequences, so that there is a Huffman encoding associated with each marker. The marker can be a pattern or a periodic position, for example the position at some position x. Then, only the k-mers after such markers are encoded. Clearly, adopting periodic positions as markers is faster than the pattern markers. In fact, instead of computing the periodic positions or pattern markers, one can simply divide the genome sequence into multiple partitions, then encode each partition using an independent set of k-mers (hence, an independent bits encoding).

An extra cost is required because of the multiple-partitions, *i.e.,* storing the headers (k-mers and their bit-codes, which will be used during the decompression process) for each partition. However, this will allow us to encode the best *local* k-mers for each partition, which eventually saves more space than storing global k-mers when no partitioning is performed. Moreover, more k-mers are encoded than the single global scheme.

5.2.5 RLE encoding

Encoding a homopolymer using RLE depends on the base that forms the homopolymer, whether it is a DNA or a non-DNA base, and its length. If the bases of a homopolymer are a DNA base and since each base is expected to ultimately be encoded using at most 3 bits, such homopolymers can be encoded by RLE if its length is greater than 10. Since

the RLE count is expected to be encoded using at least 16 bits plus 2-3 bits for the base, this indicates that encoding a homopolymer is feasible when its length is more than 10. For homopolymers of non-DNA bases, the RLE count is required to be greater than 2, because any non-DNA base is expected to be encoded using 8 bits or more.

Moreover, there are different approaches to encode the RLE counts. First of which, is by recording the position of the RLE's encoding along with the count. However, since the sizes of the genomes are usually in millions or billions, this will require at least 22 bits (genome size of 5 million) to store the positions along with the needed space for storing RLE counts (*i.e.,* it needs more than 30 bits in total). A second approach is to add a special character before and after the RLE count, and then encode the special character and encode the digits of the count number using independent encoding scheme. However, this will require at least 8×2 bits to encode both of the special characters plus encoding the count (at least 24 bits in total). Both approaches are expected to be not feasible compared to the adopted approach as the maximum length of encoding using the adopted approach (Steps 1 and 5 in the general algorithm) is expected to not exceed 24 bits.

5.3 Experimental Results and Analysis

5.3.1 Data sets

DNA data sets of five different species have been used in the experiments, including *Cholerae*, *Abscessus*, *Saccharomyces cerevisiae*, *Neurospora crassa*, and *Chr22* [56, 57, 58, 59, 60], respectively. Additional information about these datasets is presented in Table 5.2.

The sequence data format that our tool supports is FASTA (with extensions .fasta and

Table 5.2: Data Sets Used

	Length (in Mbases)	File size (in MByte)	Data Format
Cholerae	4	4.1	FASTA
Abscessus	5	5.2	FASTA
S. cerevisiae	12	12.3	FASTA
N. crassa	38	41.6	FASTA
Chr22	50	51.5	FASTA

Table 5.3: Compression ratios of the five genomes using different Huffman encoding methods, SHT, UHT, and UHTL. In SHT, the selection of the number of the most frequent k-mers is shown, whereas, in the UHT and UHTL methods, the k-mer range is specified. Note that single bases are also encoded, by default.

Dataset	SHT, %				UHT, %		UHTL, %	
	k=8	k=16	k=32	k=64	$k\in$[2,10]	$k\in$[3,10]	$k\in$[2,10]	$k\in$[3,10]
Cholerae	29.10	30.12	32.05	31.11	27.12	26.72	26.01	26.33
Abscessus	29.07	29.07	29.58	30.29	26.72	25.33	25.72	25.44
S. cerevisiae	29.95	29.86	30.27	30.99	27.29	25.97	26.09	25.59
N. crassa	30.59	30.27	32.07	31.70	27.15	26.97	26.34	26.69
Chr22	22.47	22.70	23.97	23.13	20.85	20.57	19.75	20.11

.fna). FASTA is a text-based data format that starts with a description, usually followed by a nucleotide sequence.

As a preprocessing step, we convert the bases from lowercase to uppercase. This preprocessing is required only for genomes containing lowercase bases within the entire sequence. Among the five genome data sets used in our experiments; two genomes, *S. cerevisiae* and *N. crassa*, needed a preprocessing step.

5.3.2 Results of SHT, UHT, UHTL, and MUHTL

We implemented the algorithms that are proposed in Section 5.2. Scripts for the SHT, UHT, MUHTL methods, and for decompressing the compressed files have been made available online. Note that the UHTL method becomes a special case of the MUHTL method when the number of partitions is set to one. The github link of the programs is in Section 5.4. The SHT script considers only one parameter, which is the number of k-mers to encode, for which the k-mer range is 1-10. The UHT script does not consider any parameter and the k-mer range is 1, 3-10. The reason for excluding $k = 2$ is discussed in the following section. The MUHTL script considers one parameter, *i.e.,* the number of partitions. This parameter is left to the users so that the size of each partition can be changed any time. By observing our empirical results, we recommend the partition size to be around 1 Mbp. In addition, the k-mer range for the MUHTL method is 1-10. Lastly, the decompressing script considers no parameter. It simply takes the compressed file as an input. The compressed file can be generated using any of the three scripts, where the file extension is '.uht'. When compared with the other tools, the MUHTL method is used with a partition size of 1 Mbp.

Note that with the SHT approach, the straightforward method for selecting the k-mer set to be encoded is to select the x most frequent k-mers. Clearly, when x is large, more bits are needed to encode each selected k-mer, as described in Table 5.3. While there is no control on the number of k-mers to be selected using the UHT approach, the number of k-mers to be encoded is determined by the number of k-mers that are selected so that they form an unbalanced Huffman tree.

After analyzing the results of the UHT method using k-mers with k in [2,10] and in

Table 5.4: Compression ratios (CR) of different partitions of the MUHTL method. The number of partitions is denoted as P and the variable *Size* denotes the partition size.

Data set used	Cholerae	*Abscessus*	*S. cerevisiae*	*N. crassa*	*Chr22*
P	Size, M CR, %	Size, M CR, %	Size, M CR, %	Size, M CR, %	Size, M CR, %
1	4 26.01	5.2 25.72	12.3 26.09	41.6 26.34	51.5 19.75
2	2 25.96	2.6 25.67	6.2 26.11	20.8 26.36	25.8 19.81
4	1 25.99	1.3 25.59	3.1 26.10	10.4 26.33	12.9 19.74
8	0.5 25.91	0.65 25.65	1.5 26.08	5.2 26.33	6.4 19.80
16	0.26 25.98	0.32 25.66	0.77 26.10	2.6 26.28	3.2 19.85
32	0.13 26.00	0.16 25.63	0.38 26.11	1.3 26.16	1.6 19.88
64	0.06 26.05	0.08 25.71	0.19 26.15	0.65 26.16	0.8 19.90
128	0.03 26.19	0.04 25.81	0.09 26.19	0.33 26.16	0.4 19.93
256	0.016 26.47	0.02 26.03	0.05 26.25	0.16 26.18	0.2 19.97
512	0.008 27.05	0.01 26.45	0.02 26.44	0.8 26.23	0.1 20.06

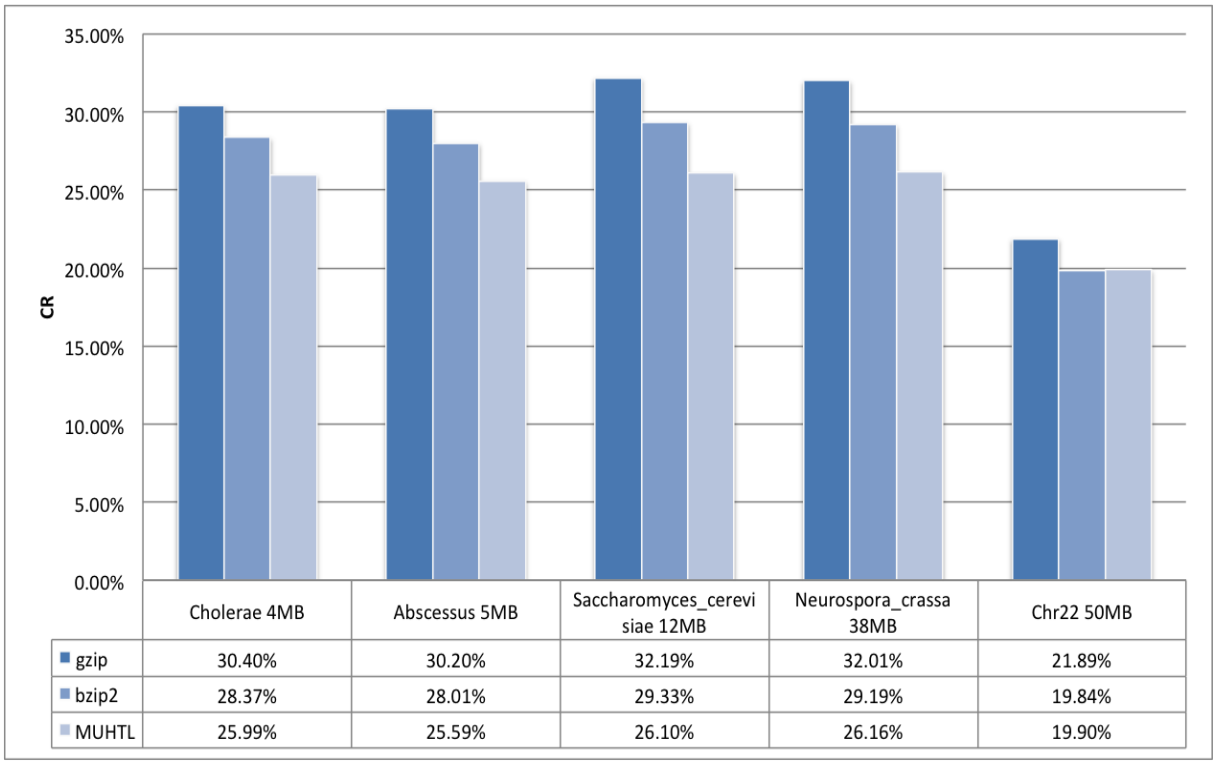
[3,10], we noticed that the results of the range [3,10] tend to be better, as shown in Table 5.3. This is likely due to the following fact. With the UHT encoding, there is a base that must be encoded using 3 bits and a 2-mer must be encoded using 4 bits. If the 2-mer does not contain the 3-bit base (in other words, contains any of the other bases that is encoded using 2-bits), encoding this 2-mer becomes unnecessary as it can be encoded using 4 bits, while encoding its bases independently also costs 4 bits. This is not the case when the range is [3,10], since any 3-mer, regardless of its bit-code, will always produce a worthwhile compression. In fact, this reason is the motivation of the UHTL method, as discussed in Section 5.2. A comparison between the UHT and UHTL methods based on the compression ratio is given in Table 5.3.

We carried out experiments on the MUHTL method using different numbers of partitions (1, 2, 4, 8, $\cdots$, 512). The results are given in Table 5.4.

After analyzing the compression results of different partitions' sizes, and using the tested datasets in the experiments, we observed that the best compression ratios by the MUHTL method took place when the partition size is around 1 Mbp. Hence, we recommend this size when using the current MUHTL method. The compression results based on this partition size, were used when the MUHTL method was compared with the other tools, as discussed in the following subsections.

5.3.3 Comparison with tools based on Huffman-encoding

Compression tools that are used in our comparison are gzip [31], and bzip2 [79]. The gzip tool is a popular compressor that works on different systems and is able to compress various data



	Cholerae 4MB	Abscessus 5MB	Saccharomyces_cerevisiae 12MB	Neurospora_crassa 38MB	Chr22 50MB
gzip	30.40%	30.20%	32.19%	32.01%	21.89%
bzip2	28.37%	28.01%	29.33%	29.19%	19.84%
MUHTL	25.99%	25.59%	26.10%	26.16%	19.90%

Fig. 5.2: Compression ratio of the MUHTL method and two well-known general purpose compressors, gzip and bzip2. Datasets are in fasta format.

types including DNA sequences. It is a variation of the LZ algorithm, which utilizes Huffman encoding to spot duplicated strings. It employs two Huffman trees, one to compresses the match lengths of a duplicated string and the other for the distance. In most cases, we found that its compression ratio is around 30%.

The bzip2 is also widely used, and it outperforms gzip and most conventional LZ-based algorithms. In the tested sequences, bzip2 yields a better compression ratio in all of the DNA sequences, as shown in Fig. 5.2. In the core of this compressor, the *Burrows-Wheeler Transform* (*BWT*) and Huffman encoding are utilized.

The results of the MUHTL method outperformed that of gzip for all the tested genomes

with a performance gain as high as 23.33%. The compression ratios of *Cholerae* and *Abscessus* are 16.9% and 18%, respectively. The sequence length did not appear to play a role in the compression ratios. Instead, we believe that the distribution of bases and the selected k-mers play a main role in the compression efficiency. For example, gzip yields similar compression ratios for *S. cerevisiae* (12 MB) and *N. crassa* (38 MB), which are 32.19% and 32.01%, respectively, as shown in Fig. 5.2. The improvements of the MUHTL method over gzip for these data sets are 23.33% and 22.35%, respectively. The data set of *Chr22* relies more on RLE encoding before the compression process, hence all tested tools show a better compression ratio for *Chr22* than the other datasets. As expected, the MUHTL method outperformed gzip by only 10% in this case.

Both bzip2 and the MUHTL method are also compared based on their compression ratios, where the MUHTL method demonstrated almost consistent dominance. For *Chr22*, bzip2 showed a slight edge over the MUHTL method by just 0.3%. However, the MUHTL method outperformed bzip2 in all the other datasets by 9.15%, 9.45%, 12.37%, and 11.58% for *Cholerae*, Abscessus, *S. cerevisiae*, and *N. crassa*, respectively.

5.4 Availability and requirements

Project name: UHT v1.0.0.

Project access:

https://github.com/aalokaily/Unbalanced-Huffman-Tree

Operating system: All experiments are carried out on the same machine running 64-bit Ubuntu on 2.8 GHz Intel Core 2 Quad processor and a memory space of 8 GB.

Other requirements: None.

License: GNU GENERAL PUBLIC LICENSE.

Any restrictions to use by non-academics: None.

5.5 List of abbreviations

SHT: standard Huffman tree; UHT: unbalanced Huffman tree; UHTL: unbalanced Huffman tree that prioritizes the encoding of the least frequent base. MSHT/MUHT/MUHTL: multiple SHT/UHT/UHTL encoding.

Chapter 6

Future Work

The authors are planning several additional studies in the future, as outlined below.

1. Future work will seek to devise an algorithm that reduces the construction space of the error tree (ET) structure. This will be a challenging undertaking indeed; nevertheless, it will be attempted since the ultimate goal is to reach a space complexity of $O(knlog_{\Sigma}n)$ words.

2. A major project planned for the future involves building a sequence read aligner based on the ET structure. This aligner will enable us to determine alignments of reads of up to two mismatches. The current construction space required to build the ET is $O(nlog^2_{\Sigma}n)$, where n is the size of the genome. For the human genome (3.3 Gbp), this space bound is impractical for low-memory computers. The future plan is to build the tree for $k = 1$ (a space of $O(nlog_{\Sigma}n)$), and then, align the 1-neighbors of the read, where 1-neighbors are defined as all sequences that are at a Hamming distance of one from the read.

   A major advantage of this aligning method and using the ET structure is that all alignments of a read are guaranteed to be found. Most of the current aligners find

the alignments by using a heuristics process (applied for the sake of speeding up the alignment process). This also reflects the error profiles of most of the sequencing machines (approximately 2% of errors of a read).

3. Although the ET structure requires a nonlinear construction space, the input sequences for the motif problem are usually not so large, especially when they are in Kbp or Mbp. This means that the construction space is feasible even for a large k. Therefore, implementing the ET-Motif algorithm will be a suitable tool for input sequences of this size.

4. Assembling sequencing reads is computed by building a *de Bruijn* graph or *String Overlap Graph* (*SOG*). The SOG strategy is the most successful assembly strategy in a practical setting, as a majority of large genomes were assembled using *Overlap Layout Consensus* (*OLC*) graph strategy [67] (SOG is a specialization of the OLC graph). This is due to the memory efficiency of SOG. However, the main challenge in building the SOG is an intensive computational step *i.e.,* computing the overlaps between *all* the reads against each other and allowing few mismatches or indels (which represent mainly the sequencing errors) [55, 68, 67]. The most practical solution for this step is the heuristic seed-and-extend method. In contrast, the ET algorithm can be used to achieve a faster and deterministic solution for this step. Thereby, contributing to a better and faster resolution for the assembly problem.

   The hierarchical genome assembly method was evaluated by using bacterial genomes (few Mbps of genome length). Applying the method to a larger genome, such as the human genome, is expected to require more time and space but is also expected to

improve the assembly results. Moreover, evaluating the method on large genomes is another future plan.

5. DNA sequences contain a considerable amount of exact and approximate repeats. Approximate repeats are sequences that are of a small Hamming or edit distance from each other. Compressing such sequences based on their Hamming or edit distance to a common neighbor sequence can improve the compression ratio at the expense of additional time that is needed to find the sequence/s that have more approximate repeats. For this, pattern-matching based on Hamming or edit distance among the sequences in the genome is required; however, by using the ET structure, this matching can be performed directly and more efficiently.

   Compressing the approximate repeats can be performed by encoding those repeats relative to a sequence that is assigned an integral key as well as the mismatched bases and their positions. For instance, let us assume the sequence "AAAAA" is assigned a key of three. The sequence "AAAA" can then be encoded as 3C1 (C is the mismatch base and 1 is the position of the mismatch).

6. A natural extension of the current compression software can allow users to specify the maximum length of repeats used in the construction of the Huffman tree. We will conduct additional research and experiments to determine whether there is a biologically relevant value to the users at run-time.

7. We will also research the possibility of variable-size partitions when constructing multiple Huffman trees for encoding, allowing the coding regions in the sequences to be

separated from the non-coding ones to derive a ranking that reflects more faithfully the status of the repeats.

8. Parallel processing will be incorporated to enable concurrent processing (such as multi-core CPUs and GPUs) of the multiple partitions of the sequences. This can ensure an efficient compression-time especially for large-scale DNA sequence datasets.

9. The software will be enabled to compress sequence datasets in the FASTQ format.

References

[1] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004.

[2] Rudolf Ahlswede, Lars Baumer, Ning Cai, Harout Aydinian, Vladimir Blinovsky, Christian Deppe, and Haik Mashurian. *General Theory of Information Transfer and Combinatorics*. Springer, 2006.

[3] Anas Al-Okaily. Error tree: A tree structure for hamming and edit distances and wildcards matching. *Journal of Computational Biology*, 22(12):1118–1128, 2015.

[4] Anas Al-okaily. Hga: de novo genome assembly method for bacterial genomes using high coverage short sequencing reads. *BMC genomics*, 17(193):1–12, 2016.

[5] Anas Al-Okaily, Almarri Badar, Al Yami Sultan, and Chun-Hsi Huang. Toward a better compression for dna sequences using huffman encoding. *Journal of Computational Biology*, accepted, 2016.

[6] Anas Al-Okaily and Chun-Hsi Huang. Et-motif: Solving the exact (l, d)-planted motif problem using error tree structure. *Journal of Computational Biology*, 23(7):615–623, 2016.

[7] Lloyd Allison, Timothy Edgoose, and Trevor I Dix. Compression of strings with approximate repeats. In *Proceedings of the 6th International Conference on Intelligent Systems for Molecular Biology (Canada)*, pages 8–16. AAAI Press, 1998.

[8] Nour S Bakr and Amr A Sharawi. Dna lossless compression algorithms: review. *American Journal of Bioinformatics Research*, 3(3):72–81, 2013.

[9] Shibdas Bandyopadhyay, Sartaj Sahni, and Sanguthevar Rajasekaran. Pms6: A fast algorithm for motif discovery. *International Journal of Bioinformatics Research and Applications 2*, 10(4-5):369–383, 2014.

[10] Anton Bankevich, Sergey Nurk, Dmitry Antipov, Alexey A Gurevich, Mikhail Dvorkin, Alexander S Kulikov, Valery M Lesin, Sergey I Nikolenko, Son Pham, Andrey D Prjibelski, et al. Spades: a new genome assembly algorithm and its applications to single-cell sequencing. *Journal of Computational Biology*, 19(5):455–477, 2012.

[11] Roger Barthelson, Adam J McFarlin, Steven D Rounsley, and Sarah Young. Plantagora: modeling whole genome sequencing and assembly of plant genomes. *PLoS One*, 6(12):e28436, 2011.

[12] Philip Bille, Inge Li Gørtz, Hjalte Wedel Vildhøj, and Søren Vind. String indexing for patterns with wildcards. *Theory of Computing Systems*, 55(1):41–60, 2014.

[13] Minh Duc Cao, Trevor I Dix, Lloyd Allison, and Chris Mears. A simple statistical algorithm for biological sequence compression. In *Data Compression Conference (Utah)*, pages 43–52. IEEE, 2007.

[14] M.J.P. Chaisson, D. Brinza, and P.A. Pevzner. De novo fragment assembly with short mate-paired reads: Does the read length matter? *Genome Res.*, 19(2):336–346, 2009.

[15] Ho-Leung Chan, Tak-Wah Lam, Wing-Kin Sung, Siu-Lung Tam, and Swee-Seong Wong. A linear size index for approximate pattern matching. In *Combinatorial Pattern Matching*, pages 49–59. Springer, 2006.

[16] Xin Chen, Sam Kwong, and Ming Li. A compression algorithm for dna sequences and its applications in genome comparison. In *Proceedings of the fourth annual international conference on Computational molecular biology (Japan)*, page 107. ACM, 2000.

[17] Xin Chen, Ming Li, Bin Ma, and John Tromp. Dnacompress: fast and effective dna sequence compression. *Bioinformatics*, 18(12):1696–1698, 2002.

[18] Neva Cherniavsky and Richard Ladner. Grammar-based compression of dna sequences. In *Proceedings of the DIMACS Working Group on The Burrows-Wheeler Transform (New Jersey)*. Citeseer, 2004.

[19] Rayan Chikhi and Paul Medvedev. Informed and automated k-mer size selection for genome assembly. *Bioinformatics*, 30(1):31–37, 2013.

[20] Francis YL Chin and Henry CM Leung. Voting algorithms for discovering long motifs the

research was supported in parts by the rgc grant hku 7135/04e. *Institute for Infocomm Research (Singapore)*, 17:21, 2005.

[21] Luís Pedro Coelho and Arlindo L Oliveira. Dotted suffix trees a structure for approximate text indexing. In *String Processing and Information Retrieval*, pages 329–336. Springer, 2006.

[22] Richard Cole, Lee-Ad Gottlieb, and Moshe Lewenstein. Dictionary matching and indexing with errors and don't cares. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 91–100. ACM, 2004.

[23] Modan K Das and Ho-Kwok Dai. A survey of dna motif finding algorithms. *BMC bioinformatics*, 8(Suppl 7):S21, 2007.

[24] Hieu Dinh, Sanguthevar Rajasekaran, and Vamsi K Kundeti. Pms5: an efficient exact algorithm for the (l, d)-motif finding problem. *BMC bioinformatics*, 12(1):410, 2011.

[25] Juliane C Dohm, Claudio Lottaz, Tatiana Borodina, and Heinz Himmelbauer. Sharcgs, a fast and highly accurate short-read assembly algorithm for de novo genomic sequencing. *Genome research*, 17(11):1697–1706, 2007.

[26] Eleazar Eskin and Pavel A Pevzner. Finding composite regulatory patterns in dna sequences. *Bioinformatics*, 18(suppl 1):S354–S363, 2002.

[27] Patricia A Evans and Andrew D Smith. Toward optimal motif enumeration. In *Algorithms and Data Structures*, pages 47–58. Springer, 2003.

[28] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, pages 390–398. IEEE, 2000.

[29] Avrilia Floratou, Sandeep Tata, and Jignesh M Patel. Efficient and accurate discovery of patterns in sequence data sets. *Knowledge and Data Engineering, IEEE Transactions on*, 23(8):1154–1168, 2011.

[30] Eugene Fratkin, Brian T Naughton, Douglas L Brutlag, and Serafim Batzoglou. Motifcut: regulatory motifs finding with maximum density subgraphs. *Bioinformatics*, 22(14):e150–e157, 2006.

[31] Jean-Loup Gailly and Mark Adler. The gzip home page. *27th July*, 2003.

[32] CR Glassey and RM Karp. On the optimality of huffman trees. *SIAM Journal on Applied Mathematics*, 31(2):368–378, 1976.

[33] Sante Gnerre, Iain MacCallum, Dariusz Przybylski, Filipe J. Ribeiro, Joshua N. Burton, Bruce J. Walker, Ted Sharpe, Giles Hall, Terrance P. Shea, Sean Sykes, Aaron M. Berlin, Daniel Aird, Maura Costello, Riza Daza, Louise Williams, Robert Nicol, Andreas Gnirke, Chad Nusbaum, Eric S. Lander, and David B. Jaffe. High-quality draft assemblies of mammalian genomes from massively parallel sequence data. *Proceedings of the National Academy of Sciences*, 108(4):1513–1518, 2011.

[34] Stéphane Grumbach and Fariza Tahi. Compression of dna sequences. 1994.

[35] Stéphane Grumbach and Fariza Tahi. A new challenge for compression algorithms: genetic sequences. *Information Processing & Management*, 30(6):875–886, 1994.

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

[37] Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *siam Journal on Computing*, 13(2):338–355, 1984.

[38] David A Huffman et al. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[39] Trinh ND Huynh, Wing-Kai Hon, Tak-Wah Lam, and Wing-Kin Sung. Approximate string matching using compressed suffix arrays. *Theoretical Computer Science*, 352(1):240–249, 2006.

[40] William R. Jeck, Josephine A. Reinhardt, David A. Baltrus, Matthew T. Hickenbotham, Vincent Magrini, Elaine R. Mardis, Jeffery L. Dangl, and Corbin D. Jones. Extending assembly of short dna sequences to handle error. *Bioinformatics*, 23(21):2942–2944, 2007.

[41] K.R. Bradnam et al. Assemblathon 2: evaluating de novo methods of genome assembly in three vertebrate species. *GigaScience*, 2(1):10+, 2013.

[42] Pavel P Kuksa and Vladimir Pavlovic. Efficient motif finding algorithms for large-alphabet inputs. *BMC bioinformatics*, 11(Suppl 8):S1, 2010.

[43] Tak-Wah Lam, Wing-Kin Sung, and Swee-Seong Wong. Improved approximate string matching using compressed suffix data structures. In *Algorithms and Computation*, pages 339–348. Springer, 2005.

[44] J Kevin Lanctot, Ming Li, Bin Ma, Shaojiu Wang, and Louxin Zhang. Distinguishing string selection problems. *Information and Computation*, 185(1):41–55, 2003.

[45] Moshe Lewenstein, J Ian Munro, Venkatesh Raman, and Sharma V Thankachan. Less space: Indexing for queries with wildcards. *Theoretical Computer Science*, 557:120–127, 2014.

[46] Shoudan Liang, Manoj Pratim Samanta, and BA Biegel. cwinnower algorithm for finding fuzzy dna motifs. *Journal of bioinformatics and computational biology*, 2(01):47–60, 2004.

[47] David Loewenstern and Peter N Yianilos. Significantly lower entropy estimates for natural dna sequences. *Journal of computational Biology*, 6(1):125–142, 1999.

[48] Giuseppe Longo and Guglielmo Galasso. An application of informational divergence to huffman codes. *Information Theory, IEEE Transactions on*, 28(1):36–43, 1982.

[49] Ruibang Luo, Binghang Liu, Yinlong Xie, Zhenyu Li, Weihua Huang, Jianying Yuan, Guangzhu He, Yanxiang Chen, Qi Pan, Yunjie Liu, et al. Soapdenovo2: an empirically improved memory-efficient short-read de novo assembler. *Gigascience*, 1(1):18, 2012.

[50] Bin Ma, John Tromp, and Ming Li. Patternhunter: faster and more sensitive homology search. *Bioinformatics*, 18(3):440–445, 2002.

[51] Tanja Magoc, Stephan Pabinger, Stefan Canzar, Xinyue Liu, Qi Su, Daniela Puiu, Luke J Tallon, and Steven L Salzberg. Gage-b: an evaluation of genome assemblers for bacterial organisms. *Bioinformatics*, 29(14):1718–1725, 2013.

[52] Marcel Margulies, Michael Egholm, William E Altman, Said Attiya, Joel S Bader, Lisa A Bemben, Jan Berka, Michael S Braverman, Yi-Ju Chen, Zhoutao Chen, et al. Genome sequencing in microfabricated high-density picolitre reactors. *Nature*, 437(7057):376–380, 2005.

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

[54] Jason R Miller, Arthur L Delcher, Sergey Koren, Eli Venter, Brian P Walenz, Anushka Brownley, Justin Johnson, Kelvin Li, Clark Mobarry, and Granger Sutton. Aggressive assembly of pyrosequencing reads with mates. *Bioinformatics*, 24(24):2818–2824, 2008.

[55] Jason R Miller, Sergey Koren, and Granger Sutton. Assembly algorithms for next-generation sequencing data. *Genomics*, 95(6):315–327, 2010.

[56] NCBI. *Vibrio cholerae O1 biovar El Tor str. N16961, taxid = 243277*, 2001.

[57] NCBI. *Mycobacterium abscessus ATCC 19977, taxid = 561007*, 2008.

[58] NCBI. *Saccharomyces cerevisiae S288c, taxid = 559292*, 2011.

[59] NCBI. *Neurospora crassa OR74A, taxid = 367110*, 2013.

[60] NCBI. *Chr22*, 2015.

[61] Marius Nicolae and Sanguthevar Rajasekaran. Efficient sequential and parallel algorithms for planted motif search. *BMC bioinformatics*, 15(1):34, 2014.

[62] D Stott Parker, Jr. Conditions for optimality of the huffman algorithm. *SIAM Journal on Computing*, 9(3):470–489, 1980.

[63] Giulio Pavesi, Giancarlo Mauri, and Graziano Pesole. An algorithm for finding signals of unknown length in dna sequences. *Bioinformatics*, 17(suppl 1):S207–S214, 2001.

[64] P. A. Pevzner, H. Tang, and M. S. Waterman. An eulerian path approach to dna fragment assembly. *Proc Natl Acad Sci U S A*, 98(17):9748–9753, 2001.

[65] Pavel A Pevzner, Sing-Hoi Sze, et al. Combinatorial approaches to finding subtle signals in dna sequences. In *ISMB*, volume 8, pages 269–278, 2000.

[66] Nadia Pisanti, Alexandra M Carvalho, Laurent Marsan, and Marie-France Sagot. Risotto: Fast extraction of motifs with mismatches. In *LATIN 2006: Theoretical Informatics*, pages 757–768. Springer, 2006.

[67] Mihai Pop. Genome assembly reborn: recent computational challenges. *Briefings in bioinformatics*, page bbp026, 2009.

[68] Mihai Pop, Steven L Salzberg, and Martin Shumway. Genome sequence assembly: Algorithms and issues. *Computer*, 35(7):47–54, 2002.

[69] Sanguthevar Rajasekaran, Sudha Balla, and C-H Huang. Exact algorithms for planted motif problems. *Journal of Computational Biology*, 12(8):1117–1128, 2005.

[70] Sanguthevar Rajasekaran and Hieu Dinh. A speedup technique for (l, d)-motif finding algorithms. *BMC research notes*, 4(1):54, 2011.

[71] P Raja Rajeswari, ALLAM APPARAO, and R KIRAN KUMAR. Huffbit compress–algorithm to compress dna sequences using extended binary trees. *Journal of Theoretical and Applied Information Technology*, 13(2):101–106, 2010.

[72] Eric Rivals, Jean-Paul Delahaye, Max Dauchet, and Olivier Delgrange. Fast discerning repeats in dna sequences with a compression algorithm. *Genome Informatics*, 8:215–226, 1997.

[73] Bacem Saada and Jing Zhang. Dna sequences compression algorithms based on the two bits codation method. In *Proceedings of the International Conference on Bioinformatics and Biomedicine (Washington DC)*, pages 1684–1686. IEEE, 2015.

[74] Marie-France Sagot. Spelling approximate repeated or common motifs using a suffix tree. In *LATIN'98: Theoretical Informatics*, pages 374–390. Springer, 1998.

[75] Steven L Salzberg, Adam M Phillippy, Aleksey Zimin, Daniela Puiu, Tanja Magoc, Sergey Koren, Todd J Treangen, Michael C Schatz, Arthur L Delcher, Michael Roberts, et al. Gage: A critical evaluation of genome assemblies and assembly algorithms. *Genome research*, 22(3):557–567, 2012.

[76] Muhammad Sardaraz, Muhammad Tahir, Ataul Aziz Ikram, and Hassan Bajwa. Seq-compress: An algorithm for biological sequence compression. *Genomics*, 104(4):225 – 228, 2014.

[77] Michael C Schatz, Arthur L Delcher, and Steven L Salzberg. Assembly of large genomes using second-generation sequencing. *Genome research*, 20(9):1165–1173, 2010.

[78] Michael C Schatz, Jan Witkowski, W Richard McCombie, et al. Current challenges in de novo plant genome sequencing and assembly. *Genome Biol*, 13(4):243, 2012.

[79] Julian Seward. bzip2, 1998.

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

[81] Jared T Simpson, Kim Wong, Shaun D Jackman, Jacqueline E Schein, Steven JM Jones, and Inanç Birol. Abyss: a parallel assembler for short read sequence data. *Genome research*, 19(6):1117–1123, 2009.

[82] Daniel D Sommer, Arthur L Delcher, Steven L Salzberg, and Mihai Pop. Minimus: a fast, lightweight genome assembler. *BMC bioinformatics*, 8(1):64, 2007.

[83] He Q Sun, Malcolm YH Low, Wen J Hsu, and Jagath C Rajapakse. Recmotif: a novel fast algorithm for weak motif discovery. *BMC bioinformatics*, 11(Suppl 11):S8, 2010.

[84] He Quan Sun, Malcolm Yoke Hean Low, Wen Jing Hsu, and Jagath C Rajapakse. Listmotif: A time and memory efficient algorithm for weak motif discovery. In *Intelligent Systems and Knowledge Engineering (ISKE), 2010 International Conference on*, pages 254–260. IEEE, 2010.

[85] He Quan Sun, Malcolm Yoke Hean Low, Wen Jing Hsu, Ching Wai Tan, and Jagath C

Rajapakse. Tree-structured algorithm for long weak motif discovery. *Bioinformatics*, 27(19):2641–2647, 2011.

[86] Waibhav Tembe, James Lowey, and Edward Suh. G-sqz: compact encoding of genomic sequence and quality data. *Bioinformatics*, 26(17):2192–2194, 2010.

[87] Ngoc Tam L Tran and Chun-Hsi Huang. A survey of motif finding web tools for detecting binding site motifs in chip-seq data. *Biol Direct*, 9(1):4, 2014.

[88] Dekel Tsur. Fast index for approximate string matching. *Journal of Discrete Algorithms*, 8(4):339–345, 2010.

[89] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[90] René L Warren, Granger G Sutton, Steven JM Jones, and Robert A Holt. Assembling millions of short dna sequences using ssake. *Bioinformatics*, 23(4):500–501, 2007.

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

[92] Xiao Yang and Jagath C Rajapakse. Graphical approach to weak motif recognition. *Genome Informatics*, 15(2):52–62, 2004.

[93] Qiang Yu, Hongwei Huo, Yipu Zhang, and Hongzhi Guo. Pairmotif: a new pattern-driven algorithm for planted (l, d) dna motif search. 2012.

[94] Federico Zambelli, Graziano Pesole, and Giulio Pavesi. Motif discovery and transcription

factor binding sites before and after the next-generation sequencing era. *Briefings in bioinformatics*, page bbs016, 2012.

[95] Daniel R Zerbino and Ewan Birney. Velvet: algorithms for de novo short read assembly using de bruijn graphs. *Genome research*, 18(5):821–829, 2008.

[96] Aleksey V Zimin, Guillaume Marçais, Daniela Puiu, Michael Roberts, Steven L Salzberg, and James A Yorke. The masurca genome assembler. *Bioinformatics*, 29(21):2669–2677, 2013.

Supplementary Tables, Figures, and Text

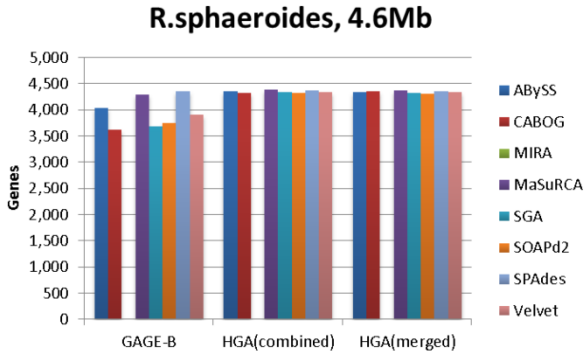
Supplementary Table 1: Additional dataset statistics.

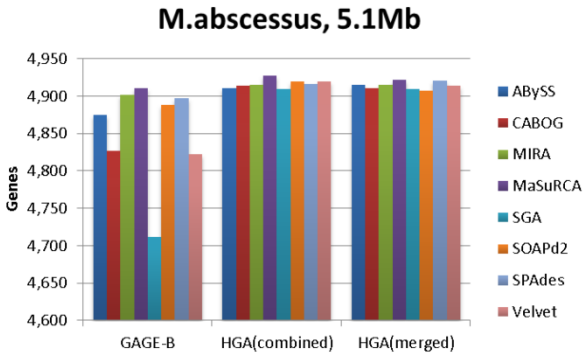
	Mean	Std. Dev. of insert	Genome size	Bases covered	%
B. cereus MiSeq	600	60	5,432,652	5,431,532	99.98
M. abscessus HiSeq	335	35	5,090,401	5,060,999	99.42
M. abscessus MiSeq	335	35	5,090,401	5,021,173	98.64
R. sphaeroides HiSeq	220	25	4,565,960	4,564,190	99.96
R. sphaeroides MiSeq	540	60	4,565,960	4,560,486	99.88
V. cholera HiSeq	335	35	4,033,464	4,009,169	99.40
V. cholerae MiSeq	335	35	4,033,464	3,879,892	96.19

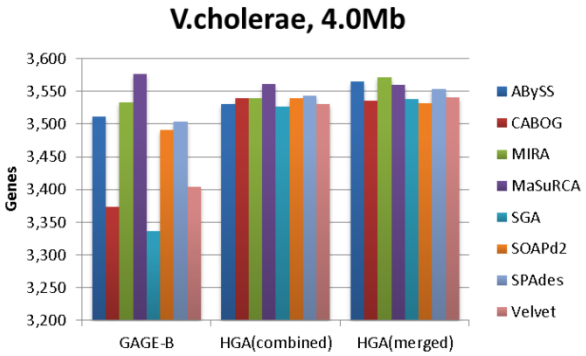
Supplementary Table 2: Datasets chosen, cleaned or raw, for each genome and each assembler. As reported in the GAGE-B study, some assemblers produced better results using the cleaned data than using the raw data, and others not; so we followed their recommendations on which datasets to use for each assembler and for each genome.

	ABy	CABOG	MIRA	MaSuRCA	SGA	SOAPdenovo	SPAdes	Velvet
B. cereus -MiSeq	raw	clean	raw	raw	raw	raw	clean	raw
M. abscessus -HiSeq	clean	clean	raw	raw	clean	clean	clean	clean
M. abscessus -MiSeq	clean	clean	clean	clean	clean	raw	clean	raw
R.sphaeroides - HiSeq	raw	raw	raw	raw	raw	raw	clean	raw
R.sphaeroides - MiSeq	raw	raw	raw	raw	clean	clean	clean	clean
V. cholerae -HiSeq	clean	clean	raw	raw	clean	clean	clean	clean
V. cholerae -MiSeq	clean	raw	clean	clean	clean	raw	clean	clean

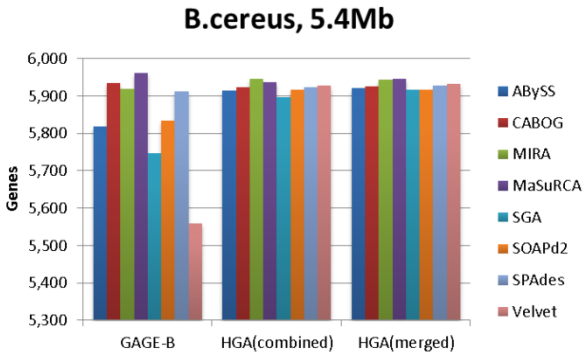
HiSeq datasets (100bp)

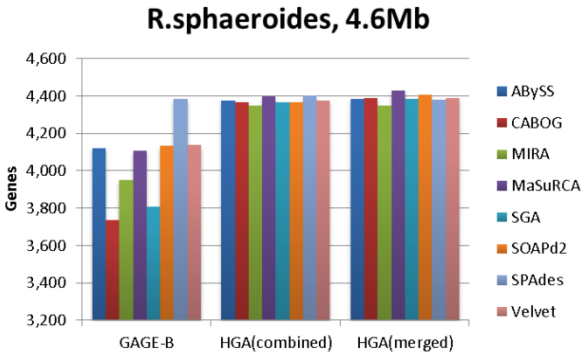


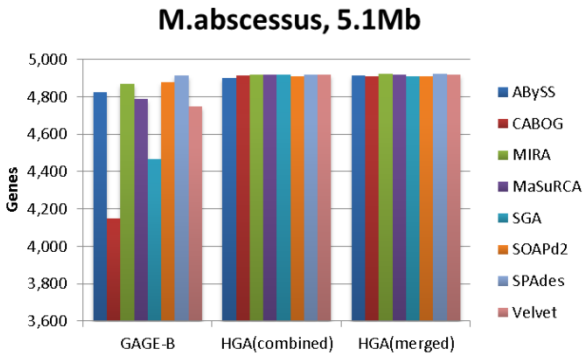


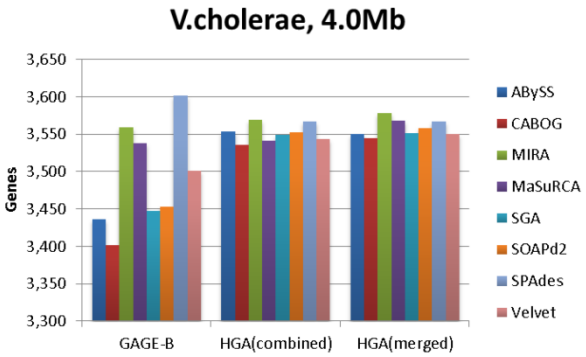


MiSeq datasets (250bp)









Supplementary Figure 1: Number of genes identified by QUAST in CAGE-B and HGA assemblies

Detailed results for all assemblies

Firstly, we state the description of QUAST's metrics that are used in the thesis and here also:
Number of contigs is the number of contigs in the assembly. **N50** is the length for which the collection of all contigs of that contig's length or longer covers at least 50% of the total length of contigs in the assembly; **NA50** is computed by firstly obtaining blocks, which resulted from breaking the contigs at misassemblies events (after aligning the contigs); after that compute the NA50 out of the blocks that were aligned to the reference, similarly of computing N50. **NG50** and **NGA50** are computed like N50 and NA50 respectively, but not out of the total length of contigs in the assembly, instead out of the reference length, here the reference genome is known in advance. **Genome fraction (%)** which is the percentage out of the reference's length, if bases that aligned to the reference. **Duplication ratio** which is the total number of aligned bases in the assembly divided by the total number of aligned bases in the reference; if the assembly contains many contigs that cover the same regions of the reference, the duplication ratio will be larger than 1.
In order to assess and evaluate the graph complexity as well as comparing different assemblies' results, we used the following metrics. **Global misassemblies** which is the number of positions in the contigs that satisfy one of the following criteria: the left flanking sequence aligns over 1 kbp away from the right flanking sequence on the reference, flanking sequences overlap on more than 1 kbp, or flanking sequences align to different strands or different chromosomes.
Local misassemblies which is the number of positions in the contigs that satisfy one of the following criteria: Two or more distinct alignments cover the breakpoint, the gap between left and right flanking sequences is less than 1 kbp, or the left and right flanking sequences both are on the same strand of the same chromosome of the reference genome. Global misassemblies indicates false assembly in the long ranges of the reference and caused by assembling regions that are far away from each other. This may mostly occur in the graph by false connection between two or more of connected components or false branching in the same connected component, due mainly to long repeats. Local misassemblies in the other hand, indicate false assembly between close regions and this occur in the same connected component, due mainly to errors. More global and local misassemblies indicate more complexity in the graph whether within connected components or among connected components.

In addition we used **# mismatches per 100 kbp,** and **# indels per 100 kbp**. These metrics assess the ability and efficiency of the assembler to correct errors (mismatches or indels) after assembling the reads. Last metric is **# Unaligned length** which is the total length of contigs (or parts of contigs) that fail to align to the reference genome. This metric evaluate the length of false assemblies results.

For more details on the metrics reported in Tables 3-13 please see http://quast.bioinf.spbau.ru/manual.html#sec3.1

Tables 3-9 highlight the best results as the following: for each dataset, the best results for each metric for the results of *contigs assembly* in the results that were reported by GAGE-B are colored in blue, similarly the best results for each metric for the results of *contigs assembly* in the results of HGA methods are colored in blue; the best results for each metric overall results of GAGE-B and HGA methods are in bold. Same highlighting tactic are applied for the *scaffolding results*, but instead of blue the highlights were in green.

Also, Tables 3-9 present QUAST results for the following assembly flows:

- *B(k):* gives the results of running the corresponding assembler using kmer size *k*.
- *AP(k,p,cov)*: gives the *average* value for each metric over *p* assemblies obtained by independently running the corresponding assembler with kmer size *k* on *p* disjoint parts, each with average read coverage *cov* (equal to 1/p of the original reads coverage).
- *C(k, p, cov):* the results of combining (using Velvet with kmer size 31) the assemblies obtained by independently running the corresponding assembler with kmer size *k* on *p* disjoint parts, each with average read coverage *cov*.

- *M(k, p, cov)*: is the result of only merging the contigs obtained by independently running the corresponding assembler with kmer size *k* on *p* disjoint parts, each with average read coverage *cov*.
- *HGA(k1, C(k2, p, cov)):* the results of re-assembling (using SPAdes assembler) the whole reads using kmer size *k1*, with the contigs obtained from the *C(k2, p, cov)* flow.
- *HGA(k1, M(k2, p, cov))*: the results of re-assembling (using SPAdes assembler) the whole reads using kmer size *k1*, with the contigs obtained from the *M(k2, p, cov)* flow.
- GAGE-B(k) gives the results obtained from the assemblies that we downloaded from http://ccb.jhu.edu/gage_b, where *k* is the kmer size used for assembly.

For each HGA flow the reported assemblies' results were selected based on the highest N50 assembly over all combinations of kmers sizes as described in the main thesis. The reported value of *k* for GAGE-B flows represents the kmer size(s) used in the construction of the de Bruijn graph by ABySS, MaSuRCA, SOAPD2, SPAdes, and Velvet, respectively the minimum overlap length used by SGA, similarly optimized to maximizing N50 as described in the GAGE-B paper.

Supplementary Table 3: 250bp MiSeq read assemblies of *B. cereus*, with reference genome size of 5,432,652bp and 6,014 genes. For metric and assembly flow descriptions see beginning of section.

Assembler	Type	Flow	N50	NA50	NG50	NGA50	Genome fraction	Duplication ratio	# contigs	# misassemblies	Local misassemblies	# mismatches per 100KB	# indels per 100KB	Unaligned length	# Genes
ABySS	Basic flow	B(81, 100x)	215,752	215,752	215,752	215,752	99.2	1.016	64	4	55	6.5	5.4	0	5,878
	HGA Preprocessing	AP(91, 2, 50x)	257,449	257,161	257,449	257,161	99.1	1.024	68	5	110	6.5	5.4	4,138	5,843
		C(91, 4, 25x)	111,188	110,839	111,188	110,694	98.2	1.017	149	0	112	96.7	11.8	2,309	5,711
	HGA re-assembly Contigs	HGA(101, C(51, 4, 25x))	287,730	287,730	287,730	287,730	98.7	1.001	79	3	27	48.6	6.9	3,325	5,914
		HGA(81, M(41, 8, 12x))	515,336	246,654	515,336	246,654	98.9	1.002	61	5	25	16.8	4.2	3,939	5,922
	GAGE Contigs	GAGE-B(49)	130,570	130,570	130,570	130,570	98.6	1.006	115	2	25	6.7	4.5	2,548	5,819
	HGA re-assembly scaffolds	HGA(101, C(51, 4, 25x))	309,910	287,730	309,910	287,730	98.7	**1.001**	76	3	30	48.6	6.9	3,325	0
		HGA(81, M(41, 8, 12x))	515,336	381,539	515,336	381,539	98.9	1.002	55	6	27	17.0	4.2	3,939	0
	GAGE scaffolds	GAGE-B(49)	135,613	135,296	135,613	135,296	98.6	1.006	102	3	29	6.7	4.6	2,548	0
CABOG	Basic flow	B(21, 100x)	53,412	53,412	53,412	53,412	99.0	1.005	174	1	9	6.0	2.6	0	5,803
	HGA Preprocessing	AP(21, 2, 50x)	38,529	38,529	37,388	37,388	98.8	1.005	228	1	10	4.9	2.6	1,071	5,752
		C(21, 4, 25x)	52,734	52,734	52,733	52,733	98.5	1.001	196	0	12	6.2	2.4	2,266	5,787
	HGA re-assembly Contigs	HGA(91, C(21, 2, 50x))	254,859	242,515	254,859	242,515	98.9	1.002	67	6	21	11.7	2.3	3,864	5,924
		HGA(91, M(21, 8, 12x))	381,118	260,843	381,118	260,843	98.8	1.001	59	5	21	14.0	2.5	3,249	5,926
	GAGE Contigs	GAGE-B	155,352	150,479	155,352	150,479	99.3	1.005	78	5	6	4.8	2.4	2,142	5,934
	HGA re-assembly scaffolds	HGA(91, C(21, 2, 50x))	319,561	269,969	319,561	269,969	98.9	1.002	63	6	23	12.1	2.4	3,864	0
		HGA(91, M(21, 8, 12x))	485,675	485,675	443,416	270,576	98.8	**1.001**	53	5	25	14.4	2.6	3,249	0
	GAGE scaffolds	GAGE-B	431,479	364,209	431,479	364,209	**99.4**	1.005	**33**	9	13	4.4	3.0	2,142	0
MIRA	Basic flow	B(21, 100x)	5,205	5,198	10,381	10,328	99.2	2.079	4,618	31	41	9.9	3.2	19,388	5,675
	HGA Preprocessing	AP(21, 2, 50x)	4,029	4,010	7,397	7,363	99.3	2.071	5,397	25	20	11.4	3.0	12,018	5,504
		C(21, 4, 25x)	82,948	82,948	78,740	78,740	98.5	1.002	163	1	5	6.2	2.3	2,712	5,829
	HGA re-assembly Contigs	HGA(91, C(21, 1, 100x))	381,025	269,870	269,870	269,870	98.7	1.002	64	1	15	10.1	2.3	3,864	5,945
		HGA(91, M(21, 8, 12x))	330,615	260,692	330,615	260,692	**99.4**	1.002	63	13	27	12.8	3.3	**1,380**	5,943
	GAGE Contigs	GAGE-B	116,480	100,038	116,480	100,038	99.2	1.007	153	9	14	4.8	2.1	5,903	5,919
	HGA re-assembly scaffolds	HGA(91, C(21, 1, 100x))	390,369	390,334	390,369	269,870	98.7	1.002	63	1	15	10.1	2.4	3,864	0
		HGA(91, M(21, 8, 12x))	330,615	260,692	330,615	260,692	**99.4**	1.002	60	13	29	13.0	3.1	**1,380**	0
	GAGE scaffolds	GAGE-B	-	-	-	-	-	-	-	-	-	-	-	-	-
MaSuRCA	Basic flow	B(61, 100x)	80,722	80,722	80,722	80,722	98.7	1.007	146	2	7	14.6	2.4	2,574	5,880
	HGA Preprocessing	AP(51, 2, 50x)	50,840	50,840	50,027	50,027	98.5	1.006	191	2	7	9.0	2.2	2,139	5,822
		C(91, 2, 50x)	74,832	74,829	74,832	74,829	98.0	1.001	162	0	3	6.2	1.9	2,313	5,805
	HGA re-assembly Contigs	HGA(91, C(21, 1, 100x))	542,153	542,005	542,153	**542,005**	98.7	1.002	61	4	16	7.9	2.5	3,864	5,937
		HGA(81, M(21, 2, 50x))	485,217	485,217	485,217	485,217	98.8	**1.001**	**50**	2	18	9.4	3.1	3,708	5,946
	GAGE Contigs	GAGE-B(101)	246,697	246,697	246,697	246,697	99.2	1.010	90	9	11	9.2	2.5	2,142	**5,961**
	HGA re-assembly scaffolds	HGA(91, C(21, 1, 100x))	542,153	**542,005**	542,153	**542,005**	98.7	1.002	60	4	17	8.0	2.4	3,864	0
		HGA(81, M(21, 2, 50x))	485,217	485,217	485,217	485,217	98.8	**1.001**	48	2	18	9.5	3.1	3,708	0
	GAGE scaffolds	GAGE-B(101)	337,861	337,861	337,861	337,861	99.2	1.010	83	12	13	9.2	2.5	2,142	0
SGA	Basic flow	B(81, 100x)	22,358	22,358	26,552	26,552	99.0	1.157	3,529	18	8	2.1	2.0	5,592	5,754
	HGA Preprocessing	AP(51, 2, 50x)	23,311	23,311	25,180	25,180	98.9	1.094	2,304	18	7	1.5	1.9	5,208	5,735
		C(61, 2, 50x)	41,550	41,550	40,815	40,815	98.2	1.001	259	0	4	1.7	1.9	2,266	5,714
	HGA re-assembly Contigs	HGA(81, C(41, 1, 100x))	330,637	182,008	330,637	182,008	98.8	1.002	88	7	34	13.9	3.3	4,745	5,897
		HGA(91, M(41, 1, 100x))	209,067	208,896	209,067	208,896	98.7	1.002	83	2	28	10.6	2.4	3,517	5,916
	GAGE Contigs	GAGE-B(65)	22,044	22,042	25,512	25,512	99.0	1.148	3,335	17	9	**2.1**	**2.0**	5,799	5,747
	HGA re-assembly scaffolds	HGA(81, C(41, 1, 100x))	331,201	208,794	331,201	208,794	98.8	1.002	78	7	41	13.7	3.4	4,745	0
		HGA(91, M(41, 1, 100x))	255,106	254,949	209,067	208,951	98.7	1.002	79	2	30	10.7	2.5	3,517	0
	GAGE scaffolds	GAGE-B(65)	25,767	25,767	25,512	25,512	98.0	1.012	502	2	**2**	**1.4**	**2.0**	2,139	0
SOAPd2	Basic flow	B(41, 100x)	25,831	25,831	24,621	24,621	98.1	1.001	435	0	1	0.7	1.8	2,288	5,639
	HGA Preprocessing	AP(41, 2, 50x)	14,299	14,299	13,798	13,798	98.1	1.002	769	0	2	0.9	1.9	2,366	5,388
		C(41, 2, 50x)	28,369	28,369	27,308	27,308	97.9	1.000	387	0	2	1.5	1.9	2,266	5,657
	HGA re-assembly Contigs	HGA(81, C(21, 2, 50x))	452,656	229,509	452,656	229,509	98.7	1.002	72	4	26	13.4	3.3	3,744	5,916
		HGA(81, M(21, 8, 12x))	483,098	452,295	483,098	452,295	98.7	1.002	72	4	28	12.6	3.2	3,744	5,917
	GAGE Contigs	GAGE-B(55)	246,346	246,346	246,346	246,346	98.4	**1.001**	105	**0**	20	9.0	2.7	2,316	5,834
	HGA re-assembly scaffolds	HGA(81, C(21, 2, 50x))	452,656	264,725	452,656	264,725	98.7	1.002	69	4	29	13.4	3.3	3,744	0
		HGA(81, M(21, 8, 12x))	483,098	452,295	483,098	452,295	98.8	1.002	68	6	30	12.7	3.2	3,744	0
	GAGE scaffolds	GAGE-B(55)	456,635	455,989	456,635	455,989	98.4	**1.001**	77	**0**	39	8.1	2.9	2,316	0
SPAdes	Basic flow	B(81, 100x)	127,979	127,979	127,979	127,979	98.7	1.003	105	3	44	15.1	3.8	3,744	5,876
	HGA Preprocessing	AP(61, 2, 50x)	90,025	89,938	89,201	89,111	98.6	1.003	164	6	31	16.3	3.4	3,799	5,831
		C(61, 4, 25x)	168,952	168,950	168,952	168,950	98.4	1.006	115	2	20	7.4	3.0	5,279	5,842
	HGA re-assembly Contigs	HGA(81, C(21, 2, 50x))	754,905	283,161	754,905	283,161	98.8	1.002	60	5	22	14.5	3.5	3,744	5,924
		HGA(81, M(21, 8, 12x))	826,529	382,489	826,529	382,489	98.8	1.002	72	12	19	13.3	3.5	8,472	5,927
	GAGE Contigs	GAGE-B(51,63,85)	103,691	103,691	103,691	103,691	99.1	1.377	49,967	8	14	19.0	3.0	10,464,230	5,911
	HGA re-assembly scaffolds	HGA(81, C(21, 2, 50x))	1,201,442	283,902	1,201,442	283,902	98.8	1.002	57	6	24	14.8	3.5	3,744	0
		HGA(81, M(21, 8, 12x))	931,487	444,453	931,487	444,453	98.8	1.002	68	12	21	13.7	3.6	227,499	0
	GAGE scaffolds	GAGE-B(51,63,85)	212,506	212,506	212,506	212,506	99.1	1.377	49,919	9	37	18.3	3.4	10,463,127	0
Velvet	Basic flow	B(51, 100x)	30,953	30,899	30,655	30,519	98.2	1.002	345	8	20	13.3	3.2	2,506	5,672
	HGA Preprocessing	AP(41, 2, 50x)	25,951	25,951	25,719	25,719	98.0	1.002	403	6	24	13.2	3.5	2,447	5,571
		C(31, 8, 12x)	79,075	66,133	79,075	66,111	98.3	1.003	162	5	31	12.4	3.8	2,408	5,778
	HGA re-assembly Contigs	HGA(91, C(21, 8, 12x))	485,240	485,240	485,240	485,240	98.7	1.003	65	3	19	14.2	2.6	3,864	5,927
		HGA(81, M(21, 8, 12x))	**1,276,342**	**1,276,279**	**1,276,342**	485,259	98.8	1.002	58	7	24	11.9	3.5	3,744	5,933
	GAGE Contigs	GAGE-B(63)	24,577	24,577	24,465	24,465	97.8	**1.001**	404	3	11	6.1	2.4	2,652	5,559
	HGA re-assembly scaffolds	HGA(91, C(21, 8, 12x))	485,240	485,240	485,240	485,240	98.7	1.003	63	3	20	14.5	2.6	3,864	0
		HGA(81, M(21, 8, 12x))	**1,276,342**	485,259	**1,276,342**	485,259	98.8	1.002	51	7	23	12.4	4.0	544,854	0
	GAGE scaffolds	GAGE-B(63)	247,748	208,398	247,748	208,398	97.8	1.009	99	11	258	6.0	3.4	2,404	0

Supplementary Table 4: 251bp MiSeq read assemblies of *R. sphaeroides*, with reference genome size of 4,565,960bp and 4,474 genes. For metric and assembly flow descriptions see beginning of section.

Assembler	Type	Flow	N50	NA50	NG50	NGA50	Genome fraction	Duplication ratio	# contigs	# mis-assemblies	Local mis-assemblies	# mismatches per 100KB	# indels per 100KB	Unaligned length	# Genes
ABySS	Basic flow	B(41, 100x)	112,782	112,782	115,786	114,140	99.6	1.021	120	12	25	17.4	4.1	1,178	4,383
	HGA Preprocessing	AP(31, 2, 50x)	110,193	110,103	27,460	27,460	49.7	1.028	57	5	13	25.2	4.1	1,292	2,189
		C(31, 8, 12x)	21,747	21,747	18,905	18,905	87.2	1.001	449	2	3	19.9	1.8	540	3,672
	HGA re-assembly Contigs	HGA(51, C(31, 1, 100x))	128,203	112,027	128,203	112,027	99.2	1.002	111	8	7	38.0	3.6	1,439	4,377
		HGA(31, M(31, 8, 12x))	173,288	170,032	173,288	170,032	99.0	1.003	109	8	12	36.7	2.6	3,329	4,386
	GAGE Contigs	GAGE-B(49)	21,647	21,647	21,441	21,441	98.5	1.001	486	1	2	3.5	0.2	96	4,121
	HGA re-assembly scaffolds	HGA(51, C(31, 1, 100x))	128,203	112,027	128,203	112,027	99.2	1.002	108	8	7	38.3	3.6	1,439	0
		HGA(31, M(31, 8, 12x))	174,187	173,288	174,187	173,288	99.0	1.003	104	8	16	37.9	2.7	3,329	0
	GAGE scaffolds	GAGE-B(49)	21,647	21,647	21,441	21,441	98.5	**1.001**	486	1	2	3.5	**0.2**	96	0
CABOG	Basic flow	B(21, 100x)	19,766	19,554	9,985	9,903	59.2	1.000	185	3	1	9.0	0.4	0	2,496
	HGA Preprocessing	AP(21, 2, 50x)	20,414	20,414	0	0	31.9	1.001	104	1	1	10.8	0.6	16	1,342
		C(21, 2, 50x)	16,115	16,115	0	0	28.7	1.001	195	4	0	6.8	0.2	0	1,208
	HGA re-assembly Contigs	HGA(51, C(21, 8, 12x))	127,081	127,081	127,081	127,081	99.3	1.002	138	9	5	22.7	2.2	1,578	4,368
		HGA(31, M(21, 4, 25x))	167,899	136,193	167,899	136,193	99.4	1.002	129	10	10	40.1	2.8	2,478	4,390
	GAGE Contigs	GAGE-B	41,794	41,176	31,540	30,364	85.7	1.004	146	6	2	7.1	0.5	78	3,738
	HGA re-assembly scaffolds	HGA(51, C(21, 8, 12x))	127,081	127,081	127,081	127,081	99.3	1.002	131	10	8	22.7	2.3	1,578	0
		HGA(31, M(21, 4, 25x))	173,888	147,716	173,888	147,716	99.4	1.003	124	10	13	40.8	2.8	2,478	0
	GAGE scaffolds	GAGE-B	45,887	42,489	40,287	36,097	85.7	1.004	131	6	6	7.2	1.0	78	0
MIRA	Basic flow	B(101, 100x)	0	0	0	0	0.0	0.000	0	0	0	0.0	0.0	0	0
	HGA Preprocessing	AP(101, 8, 12x)	0	0	0	0	0.0	0.000	0	0	0	0.0	0.0	0	0
		C(101, 8, 12x)	0	0	0	0	0.0	0.000	0	0	0	0.0	0.0	0	0
	HGA re-assembly Contigs	HGA(41, C(21, 8, 12x))	124,257	124,225	124,257	124,225	99.1	**1.001**	150	3	5	18.5	1.9	1,766	4,348
		HGA(41, M(21, 8, 12x))	124,257	124,225	124,257	124,225	99.1	**1.001**	150	3	5	18.5	1.9	1,766	4,348
	GAGE Contigs	GAGE-B	15,445	15,271	15,792	15,445	99.3	1.022	867	16	4	39.2	2.8	43	3,951
	HGA re-assembly scaffolds	HGA(41, C(21, 8, 12x))	130,354	130,354	127,712	127,712	99.2	**1.001**	142	3	9	19.5	2.0	1,766	0
		HGA(41, M(21, 8, 12x))	130,354	130,354	127,712	127,712	99.2	**1.001**	142	3	9	19.5	2.0	1,766	0
	GAGE scaffolds	GAGE-B	-	-	-	-	-	-	-	-	-	-	-	-	-
MaSuRCA	Basic flow	B(41, 100x)	116,186	116,186	112,886	112,886	93.8	1.004	88	3	2	3.4	0.7	2,450	4,175
	HGA Preprocessing	AP(41, 2, 50x)	114,614	114,614	24,633	24,633	47.0	1.004	41	2	1	4.3	0.8	1,225	2,091
		C(21, 4, 25x)	26,970	26,970	21,651	21,400	81.3	1.001	365	1	6	5.1	0.4	26	3,497
	HGA re-assembly Contigs	HGA(71, C(51, 8, 12x))	178,403	144,067	170,376	144,067	99.3	1.003	91	11	3	22.7	1.0	356	4,397
		HGA(101, M(21, 4, 25x))	**251,279**	212,893	**251,279**	212,893	**99.8**	**1.001**	**52**	7	7	9.9	0.7	**0**	**4,430**
	GAGE Contigs	GAGE-B(63)	142,742	142,742	130,714	130,714	92.1	1.004	63	5	3	9.8	0.8	341	4,109
	HGA re-assembly scaffolds	HGA(71, C(51, 8, 12x))	178,403	144,067	170,376	144,067	99.3	1.003	89	12	4	23.0	1.1	356	0
		HGA(101, M(21, 4, 25x))	**251,279**	212,893	**251,279**	212,893	**99.8**	**1.001**	**52**	7	7	9.9	0.7	**0**	0
	GAGE scaffolds	GAGE-B(63)	165,131	165,131	144,812	144,812	92.0	1.004	55	5	6	9.7	0.9	341	0
SGA	Basic flow	B(51, 100x)	19,331	19,229	19,703	19,229	99.1	1.016	785	4	2	1.0	0.2	1,204	4,107
	HGA Preprocessing	AP(51, 8, 12x)	21,546	21,545	0	0	12.4	1.015	74	1	0	2.3	0.7	0	518
		C(51, 1, 100x)	14,751	14,460	1,967	1,903	52.7	1.002	426	7	0	9.8	0.5	129	2,168
	HGA re-assembly Contigs	HGA(41, C(61, 8, 12x))	144,749	126,907	144,749	126,907	99.2	1.003	138	9	5	27.6	2.6	1,766	4,365
		HGA(31, M(101, 8, 12x))	170,071	170,071	170,071	170,071	99.1	1.002	107	4	6	38.6	2.0	2,672	4,384
	GAGE Contigs	GAGE-B(23)	9,086	9,086	9,108	9,108	99.0	1.013	986	4	**2**	**0.8**	0.4	763	3,807
	HGA re-assembly scaffolds	HGA(41, C(61, 8, 12x))	147,992	144,746	147,992	144,746	99.2	1.003	130	9	9	27.8	2.7	1,766	0
		HGA(31, M(101, 8, 12x))	170,071	170,071	170,071	170,071	99.1	1.002	104	4	8	38.6	2.0	2,672	0
	GAGE scaffolds	GAGE-B(23)	9,055	9,055	7,971	7,971	88.6	1.009	778	**0**	**1**	**0.3**	0.4	266	0
SOAPd2	Basic flow	B(41, 100x)	24,166	23,927	23,927	23,780	98.9	1.001	460	1	1	0.5	0.2	0	4,161
	HGA Preprocessing	AP(41, 8, 12x)	24,787	24,787	0	0	12.5	1.002	49	0	0	0.5	0.2	5	527
		C(31, 1, 100x)	16,391	16,391	11,001	11,001	75.7	1.002	510	1	1	5.3	0.2	0	3,132
	HGA re-assembly Contigs	HGA(41, C(21, 8, 12x))	174,187	157,067	174,187	157,067	99.2	**1.001**	127	7	5	26.8	2.9	4,011	4,367
		HGA(61, M(31, 4, 25x))	176,506	162,346	176,506	162,346	99.5	1.002	95	6	3	14.8	1.1	376	4,405
	GAGE Contigs	GAGE-B(79)	33,829	33,829	33,491	33,491	98.4	1.005	437	**1**	10	3.6	0.4	5,085	4,134
	HGA re-assembly scaffolds	HGA(41, C(21, 8, 12x))	193,157	193,157	180,964	177,461	99.2	**1.001**	119	7	8	27.9	3.1	4,011	0
		HGA(61, M(31, 4, 25x))	177,971	175,390	177,971	175,390	99.5	1.002	91	6	3	14.7	1.1	376	0
	GAGE scaffolds	GAGE-B(79)	45,133	45,022	45,133	45,022	98.4	1.007	226	4	213	3.5	0.4	4,702	0
SPAdes	Basic flow	B(41, 100x)	124,257	124,225	124,257	124,225	99.1	1.001	151	3	5	18.4	1.9	1,766	4,348
	HGA Preprocessing	AP(21, 8, 12x)	211,854	211,852	0	0	12.4	1.001	21	3	2	27.7	1.7	856	547
		C(51, 8, 12x)	26,349	26,349	22,593	22,593	88.6	1.001	414	3	2	4.0	0.6	0	3,771
	HGA re-assembly Contigs	HGA(51, C(21, 4, 25x))	196,881	174,187	196,881	174,187	99.5	1.004	113	5	4	27.4	1.8	1,578	4,402
		HGA(31, M(21, 4, 25x))	244,543	**244,543**	244,543	**244,543**	98.9	1.004	130	6	13	38.8	2.1	5,771	4,380
	GAGE Contigs	GAGE-B(31,43,65)	118,093	115,059	118,093	118,093	99.6	1.004	185	5	4	8.2	0.8	13,389	4,384
	HGA re-assembly scaffolds	HGA(51, C(21, 4, 25x))	204,704	188,727	204,704	188,727	99.5	1.004	110	6	4	27.3	1.8	1,578	0
		HGA(31, M(21, 4, 25x))	244,543	**244,543**	244,543	**244,543**	98.9	1.004	127	6	14	39.6	2.1	5,771	0
	GAGE scaffolds	GAGE-B(31,43,65)	151,794	151,794	151,794	151,794	99.7	1.016	145	8	8	7.6	1.9	13,389	0
Velvet	Basic flow	B(41, 100x)	29,448	28,750	29,092	28,677	98.8	1.003	348	5	11	16.2	1.2	0	4,213
	HGA Preprocessing	AP(31, 8, 12x)	35,181	33,678	0	0	12.4	1.001	32	1	2	17.6	1.2	481	534
		C(21, 8, 12x)	17,228	16,762	2,996	2,864	52.8	0.999	325	8	6	12.9	1.2	1,129	2,239
	HGA re-assembly Contigs	HGA(41, C(21, 8, 12x))	170,405	157,915	170,405	147,761	99.4	**1.001**	127	9	7	25.6	2.5	1,766	4,375
		HGA(31, M(21, 8, 12x))	204,678	204,678	204,678	204,678	99.0	1.003	100	8	14	36.8	2.5	2,821	4,388
	GAGE Contigs	GAGE-B(31)	24,300	24,248	24,248	24,205	98.0	**1.001**	416	2	8	11.6	0.6	688	4,136
	HGA re-assembly scaffolds	HGA(41, C(21, 8, 12x))	176,125	170,405	174,871	170,405	99.4	**1.001**	120	9	10	26.6	2.7	1,766	0
		HGA(31, M(21, 8, 12x))	204,678	204,678	204,678	204,678	99.0	1.003	97	8	15	37.1	2.6	2,821	0
	GAGE scaffolds	GAGE-B(31)	88,399	84,960	85,272	84,960	98.0	1.009	209	19	184	11.5	1.1	**0**	0

Supplementary Table 5: 101bp HiSeq read assemblies of *R. sphaeroides*, with reference genome size of 4,565,960bp and 4,474 genes. For metric and assembly flow descriptions see beginning of section.

Assembler	Type	Flow	N50	NA50	NG50	NGA50	Genome fraction	Duplication ratio	# contigs	# mis-assemblies	Local mis-assemblies	# mismatches per 100KB	# indels per 100KB	Unaligned length	# Genes
ABySS	Basic flow	B(31, 210x)	120,554	120,554	120,554	120,554	99.0	1.019	145	9	10	15.6	5.6	0	4,352
	HGA Preprocessing	AP(31, 2, 105x)	98,263	98,263	98,795	98,263	98.9	1.019	174	9	7	16.4	9.5	58	4,323
		C(31, 4, 52x)	51,404	51,404	49,231	49,231	98.1	1.007	277	2	29	206.7	17.0	213	4,204
	HGA re-assembly Contigs	HGA(41, C(21, 2, 105x))	161,703	136,021	161,703	136,021	99.0	1.000	202	7	12	29.9	2.7	8,396	4,357
		HGA(31, M(21, 4, 52x))	187,637	187,637	187,637	187,637	98.6	1.002	236	10	15	34.2	2.5	15,503	4,338
	GAGE Contigs	GAGE-B(31)	13,460	13,460	13,319	13,319	98.4	1.009	604	6	2	8.8	0.7	661	4,030
	HGA re-assembly scaffolds	HGA(41, C(21, 2, 105x))	173,221	161,703	173,221	161,703	99.1	1.001	196	7	16	34.1	2.9	8,396	0
		HGA(31, M(21, 4, 52x))	229,984	229,984	229,984	229,984	98.6	1.002	230	10	16	34.1	2.7	15,503	0
	GAGE scaffolds	GAGE-B(31)	13,475	13,475	13,460	13,460	98.5	1.010	582	34	11	9.1	0.7	27	0
CABOG	Basic flow	B(21, 210x)	4,462	4,462	3,232	3,232	75.7	1.001	1,001	2	2	15.9	1.3	0	2,606
	HGA Preprocessing	AP(21, 2, 105x)	2,949	2,947	1,507	1,507	57.2	1.004	1,018	3	2	14.5	1.2	69	1,821
		C(21, 1, 210x)	4,861	4,256	0	0	11.8	1.000	225	12	0	70.6	7.3	288	410
	HGA re-assembly Contigs	HGA(31, C(21, 8, 26x))	127,178	112,298	127,178	112,298	98.7	1.001	259	11	19	24.3	2.1	15,503	4,321
		HGA(31, M(21, 1, 210x))	143,313	127,178	143,313	127,178	99.0	1.002	204	8	18	33.6	2.2	15,897	4,362
	GAGE Contigs	GAGE-B	12,421	12,360	11,358	11,217	90.9	1.008	537	4	3	20.1	1.6	360	3,619
	HGA re-assembly scaffolds	HGA(31, C(21, 8, 26x))	186,283	161,627	186,283	161,627	98.7	1.001	245	11	29	24.2	2.1	15,503	0
		HGA(31, M(21, 1, 210x))	201,144	201,144	201,144	201,144	99.0	1.002	191	8	28	33.6	2.2	15,897	0
	GAGE scaffolds	GAGE-B	23,585	23,406	21,196	20,580	90.9	1.009	320	6	33	21.7	6.7	146	0
MIRA	Basic flow	B(91, 210x)	-	-	-	-	-	-	-	-	-	-	-	-	-
	HGA Preprocessing	AP(91, 8, 26x)	-	-	-	-	-	-	-	-	-	-	-	-	-
		C(91, 8, 26x)	-	-	-	-	-	-	-	-	-	-	-	-	-
	HGA re-assembly Contigs	HGA(31, C(21, 8, 26x))	-	-	-	-	-	-	-	-	-	-	-	-	-
		HGA(31, M(21, 8, 26x))	-	-	-	-	-	-	-	-	-	-	-	-	-
	GAGE Contigs	GAGE-B	-	-	-	-	-	-	-	-	-	-	-	-	-
	HGA re-assembly scaffolds	HGA(31, C(21, 8, 26x))	-	-	-	-	-	-	-	-	-	-	-	-	-
		HGA(31, M(21, 8, 26x))	-	-	-	-	-	-	-	-	-	-	-	-	-
	GAGE scaffolds	GAGE-B	-	-	-	-	-	-	-	-	-	-	-	-	-
MaSuRCA	Basic flow	B(41, 210x)	24,800	24,800	24,406	24,406	95.6	1.020	382	2	2	47.5	0.5	219	4,093
	HGA Preprocessing	AP(41, 2, 105x)	41,079	40,951	39,503	39,307	95.8	1.009	226	3	4	22.8	0.4	25	4,177
		C(51, 2, 105x)	103,409	103,409	102,493	102,493	95.2	1.002	154	0	2	18.3	0.4	0	4,208
	HGA re-assembly Contigs	HGA(51, C(21, 2, 105x))	284,717	222,832	284,717	222,832	99.1	1.002	**127**	5	10	16.6	1.5	4,627	**4,381**
		HGA(41, M(21, 1, 210x))	209,442	168,043	209,442	168,043	99.2	1.001	159	9	17	21.9	1.5	7,732	4,375
	GAGE Contigs	GAGE-B(55)	176,783	176,783	176,783	176,783	97.1	1.005	130	5	4	72.1	1.0	**49**	4,298
	HGA re-assembly scaffolds	HGA(51, C(21, 2, 105x))	284,717	222,832	284,717	222,832	99.1	1.002	126	5	10	16.6	1.5	4,627	0
		HGA(41, M(21, 1, 210x))	209,442	185,588	209,442	185,588	99.2	1.002	155	10	19	26.0	2.0	7,732	0
	GAGE scaffolds	GAGE-B(55)	196,511	196,511	196,511	196,511	97.1	1.005	**125**	6	6	72.1	1.1	49	0
SGA	Basic flow	B(41, 210x)	12,788	12,788	12,113	12,113	93.0	1.003	845	1	1	2.2	0.2	49	3,687
	HGA Preprocessing	AP(41, 2, 105x)	10,121	10,111	9,165	9,134	89.8	1.004	902	2	1	2.1	0.2	159	3,479
		C(21, 2, 105x)	23,702	22,341	21,543	20,662	94.7	1.000	444	3	5	3.7	0.3	0	3,940
	HGA re-assembly Contigs	HGA(31, C(51, 8, 26x))	148,826	127,185	148,826	127,185	98.8	1.001	236	4	21	27.0	2.2	15,291	4,337
		HGA(31, M(21, 1, 210x))	159,189	146,650	159,189	146,650	98.5	1.002	234	13	18	28.9	1.8	15,773	4,322
	GAGE Contigs	GAGE-B(41)	12,793	12,793	12,057	12,057	92.9	1.004	862	**1**	**1**	**2.1**	**0.2**	205	3,679
	HGA re-assembly scaffolds	HGA(31, C(51, 8, 26x))	185,025	185,025	185,025	185,025	98.8	1.001	222	4	31	27.3	2.3	15,291	0
		HGA(31, M(21, 1, 210x))	159,189	146,650	159,189	146,650	98.5	1.002	231	13	21	28.6	1.8	15,773	0
	GAGE scaffolds	GAGE-B(41)	13,487	13,487	11,833	11,793	87.2	1.003	662	**1**	**1**	**1.7**	**0.2**	54	0
SOAPd2	Basic flow	B(61, 210x)	3,066	3,062	3,014	3,008	97.1	1.009	2,476	1	1	34.2	1.0	2,299	2,796
	HGA Preprocessing	AP(51, 2, 105x)	4,239	4,231	4,185	4,165	98.0	1.008	1,986	1	2	25.9	0.8	1,737	3,101
		C(41, 4, 52x)	20,999	20,712	20,712	20,516	98.4	1.002	498	2	1	9.4	0.5	215	4,092
	HGA re-assembly Contigs	HGA(31, C(41, 1, 210x))	157,339	147,243	157,339	147,243	98.6	1.002	237	5	21	23.3	1.8	15,540	4,330
		HGA(31, M(41, 2, 105x))	126,438	126,438	126,438	126,438	98.6	1.002	257	5	38	23.9	2.7	15,664	4,306
	GAGE Contigs	GAGE-B(55)	10,993	10,906	10,600	10,512	97.3	1.002	859	2	9	25.6	1.9	1,125	3,752
	HGA re-assembly scaffolds	HGA(31, C(41, 1, 210x))	185,556	185,548	185,556	185,548	98.6	1.002	228	5	26	23.0	2.0	15,540	0
		HGA(31, M(41, 2, 105x))	161,484	161,478	161,484	161,478	98.6	1.002	243	5	44	23.1	2.8	15,641	0
	GAGE scaffolds	GAGE-B(55)	15,850	15,839	15,520	15,517	97.2	1.003	661	2	126	24.6	4.3	6,512	0
SPAdes	Basic flow	B(31, 210x)	112,631	112,631	112,301	112,298	98.5	1.002	281	1	21	18.1	1.6	15,503	4,310
	HGA Preprocessing	AP(31, 2, 105x)	64,334	64,334	63,825	63,825	98.5	1.003	294	4	27	18.2	2.5	4,845	4,263
		C(31, 4, 52x)	111,793	111,793	111,793	105,389	98.4	1.007	219	9	29	12.3	3.1	198	4,294
	HGA re-assembly Contigs	HGA(41, C(21, 2, 105x))	286,024	286,024	286,024	286,024	99.0	**1.000**	169	8	19	27.8	1.8	8,099	4,376
		HGA(41, M(21, 1, 210x))	**314,739**	**314,739**	**314,739**	**314,739**	99.0	1.001	269	7	16	31.3	2.2	30,304	4,363
	GAGE Contigs	GAGE-B(21,33,55)	74,486	74,016	83,463	83,460	**99.5**	1.014	298	6	5	6.0	0.8	25,816	4,350
	HGA re-assembly scaffolds	HGA(41, C(21, 2, 105x))	331,799	331,799	331,799	331,799	99.0	**1.000**	163	9	21	28.1	2.0	8,099	0
		HGA(41, M(21, 1, 210x))	**465,114**	**465,114**	**465,114**	**465,114**	99.0	1.001	265	8	18	34.4	2.5	30,304	0
	GAGE scaffolds	GAGE-B(21,33,55)	127,911	127,911	127,911	127,911	**99.5**	1.013	259	6	9	6.3	1.4	25,919	0
Velvet	Basic flow	B(51, 210x)	12,779	12,779	12,225	12,225	98.0	1.003	729	2	5	14.9	1.2	552	3,864
	HGA Preprocessing	AP(41, 2, 105x)	14,403	14,178	14,045	13,979	98.2	1.002	638	3	5	17.3	0.8	384	3,949
		C(31, 4, 52x)	44,626	44,626	44,315	44,315	98.4	1.001	338	1	6	12.0	0.7	21	4,222
	HGA re-assembly Contigs	HGA(31, C(31, 8, 26x))	148,202	130,168	148,202	130,168	98.7	1.008	216	14	22	28.2	2.1	15,291	4,340
		HGA(31, M(41, 1, 210x))	159,736	150,263	150,266	150,263	98.6	1.002	220	9	21	24.0	2.3	15,291	4,334
	GAGE Contigs	GAGE-B(49)	13,800	13,775	13,087	13,087	97.9	1.001	696	2	4	10.2	0.3	404	3,904
	HGA re-assembly scaffolds	HGA(31, C(31, 4, 52x))	190,800	190,772	190,800	190,772	98.6	1.003	213	7	22	26.0	1.8	15,305	0
		HGA(31, M(51, 1, 210x))	185,472	185,440	185,472	185,440	98.7	1.001	208	4	36	26.5	3.3	15,291	0
	GAGE scaffolds	GAGE-B(49)	36,086	33,823	34,182	33,086	98.0	1.010	348	20	333	10.4	0.7	252	0

Supplementary Table 6: 250bp MiSeq read assemblies of *M. abscessus*, with reference genome size of 5,090,401bp and 4,992 genes. For metric and assembly flow descriptions see beginning of section.

Assembler	Type	Flow	N50	NA50	NG50	NGA50	Genome fraction	Duplication ratio	# contigs	# mis-assemblies	Local mis-assemblies	# mismatches per 100KB	# indels per 100KB	Unaligned length	# Genes
ABySS	Basic flow	B(51, 100x)	87,472	86,675	87,624	86,675	100.6	1.001	168	2	5	4.1	1.1	80,087	4,853
	HGA Preprocessing	AP(41, 2, 50x)	40,153	37,883	40,153	38,366	100.4	1.001	322	4	2	3.1	1.0	79,271	4,739
		C(61, 4, 25x)	79,635	75,731	79,635	75,731	100.4	1.002	189	5	3	10.0	1.1	76,474	4,831
	HGA re-assembly Contigs	HGA(31, C(51, 4, 25x))	272,898	205,053	272,898	205,053	100.6	1.003	496	6	8	12.8	0.8	222,852	4,902
		HGA(41, M(91, 8, 12x))	294,301	199,090	294,301	225,649	100.7	1.016	408	10	6	10.5	1.3	136,348	4,915
	GAGE Contigs	GAGE-B(58)	70,424	67,488	70,424	68,549	99.2	1.001	210	2	2	1.9	0.6	80,486	4,825
	HGA re-assembly scaffolds	HGA(31, C(51, 4, 25x))	272,898	210,070	274,707	225,649	100.6	1.051	492	10	9	15.9	1.1	168,639	0
		HGA(41, M(91, 8, 12x))	340,251	225,649	388,189	231,650	100.7	1.016	406	11	6	12.5	1.5	662,877	0
	GAGE scaffolds	GAGE-B(58)	73,179	73,179	70,126	70,126	99.2	**1.001**	208	2	3	**0.7**	2.2	157,344	0
CABOG	Basic flow	B(21, 100x)	22,071	21,315	22,335	21,484	100.0	1.008	352	3	4	3.5	0.5	77,638	4,667
	HGA Preprocessing	AP(21, 2, 50x)	23,089	22,579	22,871	22,584	99.7	1.001	374	3	2	3.4	0.6	77,409	4,615
		C(21, 2, 50x)	63,342	61,017	63,342	61,017	99.9	1.000	149	4	2	1.9	0.3	77,986	4,814
	HGA re-assembly Contigs	HGA(31, C(21, 8, 12x))	287,637	203,401	287,637	203,401	100.7	1.013	472	12	5	25.4	2.4	168,639	4,914
		HGA(21, M(21, 8, 12x))	278,455	187,335	278,455	187,335	100.6	1.016	546	14	8	30.2	2.0	180,878	4,910
	GAGE Contigs	GAGE-B	8,655	8,294	8,716	8,344	97.5	1.030	857	122	5	4.2	0.7	75,046	4,148
	HGA re-assembly scaffolds	HGA(31, C(21, 8, 12x))	302,807	203,401	302,807	210,004	100.7	1.013	468	12	6	25.5	2.5	168,639	0
		HGA(21, M(21, 8, 12x))	278,455	187,335	278,455	187,335	100.6	1.016	545	14	8	28.9	2.0	180,878	0
	GAGE scaffolds	GAGE-B	9,070	8,413	9,127	8,433	97.5	1.030	847	131	5	4.2	0.8	75,046	0
MIRA	Basic flow	B(21, 100x)	6,089	4,779	15,167	12,051	100.8	2.195	4,717	1,476	22	4.9	0.9	167,341	4,594
	HGA Preprocessing	AP(21, 2, 50x)	3,246	2,822	8,642	7,357	100.7	2.210	7,000	1,296	27	8.1	1.2	192,680	4,339
		C(21, 2, 50x)	36,144	35,721	36,347	35,910	100.6	1.011	338	14	5	3.7	0.5	85,563	4,753
	HGA re-assembly Contigs	HGA(31, C(21, 1, 100x))	273,781	205,053	273,781	205,053	100.7	1.005	474	6	3	12.3	1.0	210,953	4,918
		HGA(21, M(21, 2, 50x))	388,304	**308,604**	388,304	**308,604**	100.7	1.016	528	13	7	31.1	2.2	180,151	**4,924**
	GAGE Contigs	GAGE-B	81,728	64,678	114,083	74,987	**100.8**	1.150	1,760	2,358	35	4.2	0.6	45,545	4,869
	HGA re-assembly scaffolds	HGA(31, C(21, 1, 100x))	274,707	210,037	274,707	225,649	100.7	1.015	471	9	3	13.1	1.2	156,740	0
		HGA(21, M(21, 2, 50x))	388,304	**308,604**	388,304	**308,604**	100.7	1.016	527	12	7	31.0	2.4	180,151	0
	GAGE scaffolds	GAGE-B	-	-	-	-	-	-	-	-	-	-	-	-	-
MaSuRCA	Basic flow	B(101, 100x)	11,889	11,524	12,466	12,075	100.2	1.056	644	5	2	2.8	0.4	80,838	4,569
	HGA Preprocessing	AP(91, 2, 50x)	23,342	22,308	23,893	23,547	99.9	1.024	332	6	3	2.4	0.4	76,301	4,701
		C(71, 2, 50x)	140,334	121,303	140,334	121,303	100.1	1.014	89	5	3	2.7	0.4	17,310	4,867
	HGA re-assembly Contigs	HGA(91, C(21, 1, 100x))	322,474	209,888	322,474	233,055	100.7	1.012	148	7	3	4.8	0.8	61,909	4,920
		HGA(61, M(21, 1, 100x))	340,250	272,329	340,250	272,329	100.7	1.014	287	6	3	3.9	**0.4**	102,307	4,920
	GAGE Contigs	GAGE-B(99)	36,211	35,496	38,240	37,156	99.7	1.067	326	70	**2**	2.9	0.6	72,188	4,790
	HGA re-assembly scaffolds	HGA(91, C(21, 1, 100x))	340,280	209,888	340,280	233,055	100.7	1.012	147	8	3	5.4	0.9	61,909	0
		HGA(61, M(21, 1, 100x))	340,250	272,329	340,250	272,329	100.7	1.014	286	7	3	4.5	0.5	102,307	0
	GAGE scaffolds	GAGE-B(99)	36,211	35,496	38,240	37,156	99.7	1.067	325	70	**2**	2.9	0.6	72,188	0
SGA	Basic flow	B(91, 100x)	28,150	26,835	28,371	27,256	100.7	1.016	488	71	2	2.0	0.4	84,449	4,713
	HGA Preprocessing	AP(61, 2, 50x)	18,850	18,371	19,115	18,675	100.5	1.011	606	42	2	1.9	0.4	83,864	4,580
		C(21, 2, 50x)	79,650	76,525	79,650	76,525	100.5	1.003	146	4	2	2.4	0.4	71,528	4,842
	HGA re-assembly Contigs	HGA(31, C(51, 8, 12x))	363,745	205,054	363,745	205,054	100.7	1.005	463	12	4	20.1	1.6	210,953	4,919
		HGA(21, M(71, 8, 12x))	278,376	193,542	278,376	209,488	100.6	1.014	553	11	9	31.2	2.4	192,021	4,910
	GAGE Contigs	GAGE-B(65)	12,696	12,320	13,299	12,834	100.7	1.045	1,117	180	4	2.0	0.4	74,740	4,467
	HGA re-assembly scaffolds	HGA(31, C(51, 8, 12x))	363,745	210,039	363,745	225,649	100.7	1.016	460	14	4	22.9	1.8	156,740	0
		HGA(21, M(71, 8, 12x))	278,376	193,542	278,376	209,488	100.6	1.014	552	11	9	31.4	2.5	192,021	0
	GAGE scaffolds	GAGE-B(65)	12,834	12,472	13,299	12,834	100.7	1.031	815	10	**2**	1.8	**0.4**	67,076	0
SOAPd2	Basic flow	B(51, 100x)	38,506	36,994	39,634	38,168	100.6	1.002	275	2	2	1.4	0.4	83,782	4,772
	HGA Preprocessing	AP(41, 2, 50x)	15,745	15,319	15,782	15,405	100.4	1.002	616	2	1	2.2	0.4	83,569	4,508
		C(51, 2, 50x)	67,061	62,457	67,061	65,560	100.4	1.000	188	3	3	2.4	0.4	83,293	4,830
	HGA re-assembly Contigs	HGA(81, C(41, 1, 100x))	274,727	210,215	274,727	210,215	100.7	1.016	198	7	3	3.8	0.5	54,982	4,911
		HGA(21, M(61, 4, 25x))	278,409	194,290	278,409	209,503	100.6	1.017	549	12	14	32.1	2.9	192,021	4,909
	GAGE Contigs	GAGE-B(47)	131,561	113,272	131,561	113,272	100.6	1.014	**113**	5	19	2.2	0.7	**17,490**	4,877
	HGA re-assembly scaffolds	HGA(81, C(41, 1, 100x))	274,727	210,215	274,727	210,215	100.7	1.016	197	8	3	4.4	0.6	54,982	0
		HGA(21, M(61, 4, 25x))	278,409	194,290	278,409	209,503	100.5	1.017	548	12	14	31.3	2.9	194,332	0
	GAGE scaffolds	GAGE-B(47)	147,990	147,162	152,604	147,162	100.6	1.014	**101**	5	31	2.2	0.7	**17,490**	0
SPAdes	Basic flow	B(31, 100x)	231,671	193,002	253,859	193,002	100.5	1.003	500	8	4	7.4	0.6	222,954	4,900
	HGA Preprocessing	AP(21, 2, 50x)	128,044	116,659	131,477	116,659	100.4	1.014	371	9	5	15.1	1.1	108,673	4,872
		C(61, 4, 25x)	140,162	131,364	144,325	131,364	100.5	1.003	97	7	4	3.5	0.5	72,070	4,889
	HGA re-assembly Contigs	HGA(71, C(21, 8, 12x))	372,707	234,759	372,707	234,759	100.7	1.016	224	17	7	8.5	1.3	72,271	4,916
		HGA(21, M(81, 8, 12x))	**474,754**	206,114	**474,754**	206,114	100.6	1.015	530	14	10	28.0	2.3	192,329	4,923
	GAGE Contigs	GAGE-B(33,55,65,75,85,99)	215,400	209,894	220,161	209,894	**100.8**	1.018	908	20	5	4.7	0.9	259,449	4,913
	HGA re-assembly scaffolds	HGA(71, C(21, 8, 12x))	372,707	234,759	372,707	234,759	100.7	1.017	222	18	7	8.8	1.3	72,271	0
		HGA(21, M(81, 8, 12x))	**474,754**	206,114	474,754	206,114	100.6	1.015	529	14	10	26.7	2.3	192,329	0
	GAGE scaffolds	GAGE-B(33,55,65,75,85,99)	215,400	209,894	220,161	209,894	**100.8**	1.018	908	20	5	4.7	0.9	264,835	0
Velvet	Basic flow	B(91, 100x)	23,440	22,003	23,440	22,003	100.0	1.006	370	30	3	5.5	0.9	67,345	4,621
	HGA Preprocessing	AP(71, 2, 50x)	16,952	16,481	17,060	16,455	99.5	1.004	537	37	3	5.1	0.9	82,973	4,471
		C(71, 2, 50x)	54,548	53,765	54,548	53,765	100.1	1.001	220	2	3	2.7	0.6	83,376	4,778
	HGA re-assembly Contigs	HGA(31, C(101, 8, 12x))	273,781	205,054	273,781	205,054	100.6	1.003	473	8	5	10.7	0.6	222,852	4,917
		HGA(31, M(91, 8, 12x))	452,190	210,096	452,190	225,649	100.6	1.014	454	11	5	13.0	1.4	168,034	4,920
	GAGE Contigs	GAGE-B(97)	47,327	40,129	48,155	41,485	100.4	1.011	279	76	3	2.5	0.6	62,440	4,746
	HGA re-assembly scaffolds	HGA(31, C(101, 8, 12x))	274,707	210,039	274,707	225,649	100.6	1.013	471	9	5	11.1	0.7	168,639	0
		HGA(31, M(91, 8, 12x))	466,584	225,649	**571,267**	232,801	100.6	1.028	453	12	5	13.0	1.4	168,034	0
	GAGE scaffolds	GAGE-B(97)	70,983	44,950	70,983	45,957	100.4	1.011	229	124	19	2.5	0.6	62,427	0

Supplementary Table 7: 100bp HiSeq read assemblies of *M. abscessus*, with reference genome size of 5,090,401bp and 4,992 genes. For metric and assembly flow descriptions see beginning of section.

Assembler	Type	Flow	N50	NA50	NG50	NGA50	Genome fraction	Duplication ratio	# contigs	# mis-assemblies	Local mis-assemblies	# mismatches per 100KB	# indels per 100KB	Unaligned length	# Genes
ABySS	Basic flow	B(41, 115x)	148,261	147,341	148,261	148,137	99.9	1.005	85	3	9	5.5	1.1	66,112	4,901
	HGA Preprocessing	AP(31, 2, 57x)	127,253	123,841	127,253	123,841	99.8	1.004	102	3	13	5.5	1.8	66,708	4,886
		C(31, 2, 57x)	125,738	105,633	125,738	125,738	99.6	1.000	126	2	16	4.9	2.0	77,908	4,865
	HGA re-assembly Contigs	HGA(31, C(61, 8, 14x))	321,915	231,737	321,915	247,233	99.8	1.017	98	13	7	11.1	1.4	9,431	4,911
		HGA(31, M(91, 2, 57x))	336,297	190,184	336,297	233,884	99.8	1.006	94	6	12	8.5	0.9	63,512	4,915
	GAGE Contigs	GAGE-B(53)	119,446	111,763	119,446	115,738	99.8	1.003	124	4	8	3.9	1.5	66,194	4,875
	HGA re-assembly scaffolds	HGA(31, C(61, 8, 14x))	321,915	260,299	321,915	260,299	99.9	1.017	94	13	9	11.8	1.4	9,431	0
		HGA(31, M(91, 2, 57x))	336,297	190,184	336,297	233,884	99.8	1.017	92	8	13	9.0	1.1	9,299	0
	GAGE scaffolds	GAGE-B(53)	147,937	127,410	147,937	127,410	99.8	1.015	77	11	26	9.8	2.1	11,899	0
CABOG	Basic flow	B(21, 115x)	13,923	13,366	13,670	13,087	97.4	1.006	578	3	4	5.6	3.8	78,651	4,368
	HGA Preprocessing	AP(21, 2, 57x)	10,824	10,602	10,705	10,503	97.9	1.002	703	3	5	6.7	0.8	76,078	4,278
		C(21, 2, 57x)	20,502	19,499	19,952	19,415	98.3	1.000	394	4	7	8.8	1.0	77,038	4,548
	HGA re-assembly Contigs	HGA(31, C(21, 2, 57x))	364,306	247,266	364,306	247,266	99.9	1.017	89	10	10	16.2	1.4	9,360	4,914
		HGA(31, M(21, 4, 28x))	364,231	278,346	364,231	278,346	99.8	1.016	82	9	9	11.0	0.9	9,299	4,910
	GAGE Contigs	GAGE-B	81,416	78,165	81,416	78,165	99.5	1.004	127	7	6	8.6	5.8	65,602	4,827
	HGA re-assembly scaffolds	HGA(31, C(21, 2, 57x))	364,306	247,266	364,306	247,266	99.9	1.017	88	10	10	16.2	1.6	9,360	0
		HGA(31, M(21, 4, 28x))	364,231	278,346	364,231	278,346	99.8	1.016	80	9	9	11.1	1.1	9,299	0
	GAGE scaffolds	GAGE-B	94,359	89,623	94,359	89,623	99.5	**1.004**	109	10	13	8.8	6.2	65,564	0
MIRA	Basic flow	B(21, 115x)	8,616	8,242	15,845	15,260	99.9	2.043	2,805	75	7	8.0	1.0	166,010	4,534
	HGA Preprocessing	AP(21, 2, 57x)	5,811	5,583	10,086	9,823	99.9	2.047	3,654	77	8	10.9	1.0	168,031	4,229
		C(21, 2, 57x)	120,187	119,730	120,187	119,730	99.8	1.004	105	6	7	9.3	0.7	66,133	4,879
	HGA re-assembly Contigs	HGA(31, C(21, 8, 14x))	290,400	214,205	290,400	214,205	99.9	1.017	89	16	12	17.6	1.3	9,299	4,915
		HGA(31, M(21, 8, 14x))	290,400	212,994	290,400	212,994	99.9	1.019	75	21	13	23.7	1.7	9,299	4,915
	GAGE Contigs	GAGE-B	129,228	119,746	147,272	129,075	**100.0**	1.170	3,293	452	6	6.1	0.7	59,775	4,902
	HGA re-assembly scaffolds	HGA(31, C(21, 8, 14x))	304,450	214,205	304,450	214,205	99.9	1.017	88	16	13	17.6	1.3	9,299	0
		HGA(31, M(21, 8, 14x))	290,400	212,994	290,400	212,994	99.9	1.019	73	21	15	24.4	1.8	9,299	0
	GAGE scaffolds	GAGE-B	-	-	-	-	-	-	-	-	-	-	-	-	-
MaSuRCA	Basic flow	B(81, 115x)	14,899	13,999	15,406	14,677	99.8	1.044	544	2	8	21.2	1.3	79,393	4,630
	HGA Preprocessing	AP(31, 2, 28x)	28,219	27,638	29,139	28,583	99.2	1.029	291	6	4	6.7	0.7	77,561	4,770
		C(71, 2, 57x)	167,952	146,852	167,952	146,852	99.6	1.005	71	4	2	5.6	1.0	54,211	4,893
	HGA re-assembly Contigs	HGA(31, C(61, 2, 57x))	508,685	**343,659**	508,685	**343,659**	99.9	1.016	77	7	5	10.4	0.8	9,299	**4,927**
		HGA(31, M(81, 4, 28x))	**551,349**	313,850	**551,350**	313,850	99.9	1.016	80	10	4	9.8	1.2	9,299	4,922
	GAGE Contigs	GAGE-B(89)	246,830	187,809	246,830	187,809	99.9	1.018	**66**	6	2	49.7	4.2	**102**	4,910
	HGA re-assembly scaffolds	HGA(31, C(61, 2, 57x))	508,685	**343,659**	508,685	**343,659**	99.9	1.016	77	7	5	10.4	0.8	9,299	0
		HGA(31, M(81, 4, 28x))	**551,350**	313,850	**551,350**	313,850	99.9	1.016	78	11	5	9.8	1.2	9,299	0
	GAGE scaffolds	GAGE-B(89)	246,830	187,809	246,830	187,809	**100.0**	1.018	**59**	11	3	50.4	4.3	**102**	0
SGA	Basic flow	B(41, 115x)	29,477	28,291	29,507	28,728	99.8	1.004	371	2	2	1.1	0.4	78,127	4,725
	HGA Preprocessing	AP(31, 2, 57x)	21,090	20,597	21,224	20,854	99.6	1.003	512	3	1	1.2	0.3	78,207	4,611
		C(21, 2, 57x)	63,311	55,471	63,311	58,424	99.6	1.000	187	2	3	1.5	0.4	77,935	4,819
	HGA re-assembly Contigs	HGA(31, C(71, 8, 14x))	344,886	232,801	344,886	232,801	99.8	1.005	100	6	10	8.3	1.1	63,512	4,909
		HGA(31, M(61, 4, 28x))	286,710	231,671	286,710	231,671	99.8	1.006	101	10	7	22.8	1.7	63,512	4,909
	GAGE Contigs	GAGE-B(65)	28,734	27,712	28,781	27,760	99.8	1.006	378	**3**	**1**	**1.2**	**0.4**	66,157	4,712
	HGA re-assembly scaffolds	HGA(31, C(71, 8, 14x))	344,886	232,801	344,886	232,801	99.8	1.016	97	8	11	9.8	1.5	9,299	0
		HGA(31, M(61, 4, 28x))	313,536	278,346	313,536	278,346	99.8	1.017	97	12	9	14.5	1.1	9,299	0
	GAGE scaffolds	GAGE-B(65)	28,734	27,712	28,781	27,760	99.7	1.005	363	**3**	**1**	**1.2**	**0.4**	66,157	0
SOAPd2	Basic flow	B(61, 115x)	20,458	20,325	20,620	20,395	99.8	1.006	481	4	1	1.9	0.3	66,698	4,593
	HGA Preprocessing	AP(51, 2, 57x)	17,073	16,361	17,155	16,535	99.8	1.004	572	3	1	1.4	0.3	72,611	4,516
		C(51, 2, 57x)	58,445	53,919	58,445	54,497	99.7	1.003	199	3	2	1.5	0.4	66,556	4,811
	HGA re-assembly Contigs	HGA(71, C(21, 8, 14x))	365,034	234,293	365,034	234,293	99.9	1.016	76	8	8	5.2	0.9	6,405	4,919
		HGA(31, M(81, 4, 28x))	360,431	233,090	360,431	233,090	99.9	1.005	103	10	10	8.9	1.1	63,512	4,907
	GAGE Contigs	GAGE-B(49)	148,639	144,505	148,639	147,199	99.9	1.014	91	9	15	1.8	0.7	17,699	4,888
	HGA re-assembly scaffolds	HGA(71, C(21, 8, 14x))	365,034	243,843	365,034	243,843	99.9	1.016	73	8	9	6.1	0.9	6,405	0
		HGA(31, M(81, 4, 28x))	360,431	233,090	360,431	233,090	99.9	1.016	101	12	10	9.3	1.1	9,299	0
	GAGE scaffolds	GAGE-B(49)	150,256	147,925	150,256	147,925	99.9	1.014	85	10	20	1.8	0.7	17,699	0
SPAdes	Basic flow	B(31, 115x)	271,028	209,890	271,028	209,890	99.8	1.005	117	4	11	5.4	0.8	63,512	4,903
	HGA Preprocessing	AP(31, 2, 57x)	188,056	164,996	188,056	164,996	99.7	1.003	116	4	13	5.0	0.8	71,627	4,895
		C(31, 4, 28x)	231,735	185,600	231,735	185,600	99.8	1.011	97	3	16	4.4	1.0	68,268	4,902
	HGA re-assembly Contigs	HGA(31, C(61, 8, 14x))	397,162	209,890	397,162	209,890	99.8	1.006	93	15	6	15.8	1.8	63,512	4,916
		HGA(61, M(21, 8, 14x))	428,516	225,656	428,516	225,656	99.9	1.017	81	13	8	13.2	1.4	8,404	4,921
	GAGE Contigs	GAGE-B(33,55,65,75,85,99)	150,258	147,871	150,258	147,871	99.9	1.006	96	4	5	2.2	0.6	56,101	4,897
	HGA re-assembly scaffolds	HGA(31, C(61, 8, 14x))	428,650	209,890	428,650	209,890	99.9	1.016	89	16	9	16.4	1.9	9,299	0
		HGA(61, M(21, 8, 14x))	548,170	225,656	548,170	225,656	99.9	1.017	79	14	9	14.4	1.5	8,404	0
	GAGE scaffolds	GAGE-B(33,55,65,75,85,99)	215,724	209,754	223,056	213,402	**100.0**	1.069	70	8	7	5.0	1.5	56,103	0
Velvet	Basic flow	B(51, 115x)	53,593	47,961	53,706	47,961	99.7	1.006	169	6	9	4.5	0.9	54,213	4,809
	HGA Preprocessing	AP(41, 2, 57x)	35,165	34,625	35,165	34,889	99.5	1.002	277	6	8	3.6	0.8	71,656	4,719
		C(31, 4, 28x)	162,336	154,189	162,336	156,735	99.6	1.015	81	9	18	4.6	0.9	11,899	4,885
	HGA re-assembly Contigs	HGA(61, C(21, 8, 14x))	418,776	225,668	418,776	233,193	99.9	1.017	**66**	12	12	12.5	0.9	4,034	4,919
		HGA(31, M(51, 1, 115x))	421,917	278,346	421,917	278,346	99.9	1.017	92	8	11	8.2	1.1	9,299	4,914
	GAGE Contigs	GAGE-B(49)	60,955	58,337	61,615	58,337	99.8	**1.003**	155	4	9	3.2	0.7	66,313	4,822
	HGA re-assembly scaffolds	HGA(61, C(21, 8, 14x))	418,776	225,668	418,776	233,193	99.9	1.017	64	13	12	12.5	0.9	4,034	0
		HGA(31, M(51, 1, 115x))	421,917	278,346	421,917	278,346	99.9	1.017	92	8	11	8.2	1.1	9,299	0
	GAGE scaffolds	GAGE-B(49)	248,309	147,755	262,034	147,755	99.8	1.019	74	9	82	3.2	0.8	201	0

Supplementary Table 8: 250bp MiSeq read assemblies of *V. cholera*, with reference genome size of 4,033,464bp and 3,693 genes. For metric and assembly flow descriptions see beginning of section.

Assembler	Type	Flow	N50	NA50	NG50	NGA50	Genome fraction	Duplication ratio	# contigs	# mis-assemblies	Local mis-assemblies	# mismatches per 100KB	# indels per 100KB	Unaligned length	# Genes
ABySS	Basic flow	B(91, 100x)	92,024	92,024	85,271	85,271	101.5	1.002	213	6	2	3.3	2.9	767	3,495
	HGA Preprocessing	AP(61, 2, 50x)	34,710	34,710	34,525	34,525	100.6	1.002	393	3	1	4.1	2.8	1,166	3,367
		C(81, 2, 50x)	71,270	71,270	68,996	68,996	100.8	1.000	214	5	1	5.1	2.9	915	3,433
	HGA re-assembly Contigs	HGA(101, C(51, 2, 50x))	243,230	199,213	243,230	199,213	101.8	1.002	232	10	6	8.0	2.9	57,060	3,554
		HGA(101, M(31, 2, 50x))	243,230	153,070	243,230	153,070	101.8	1.002	236	10	5	8.4	3.3	57,576	3,550
	GAGE Contigs	GAGE-B(65)	60,973	60,473	60,473	60,272	101.1	1.001	267	2	0	3.3	2.7	1,062	3,436
	HGA re-assembly scaffolds	HGA(101, C(51, 2, 50x))	243,230	199,213	243,230	199,213	101.8	1.002	231	10	7	8.2	2.9	57,060	0
		HGA(101, M(31, 2, 50x))	243,230	153,070	243,230	153,070	101.8	1.002	235	10	6	8.5	3.3	57,576	0
	GAGE scaffolds	GAGE-B(65)	60,973	60,473	60,473	60,272	101.1	1.001	267	**2**	**0**	3.3	**2.7**	1,062	0
CABOG	Basic flow	B(21, 100x)	24,874	24,874	22,450	22,450	97.6	1.003	285	10	6	8.4	3.1	49	3,288
	HGA Preprocessing	AP(21, 2, 50x)	20,379	20,352	18,978	18,646	97.2	1.003	348	9	4	7.5	3.0	106	3,242
		C(21, 4, 25x)	55,962	54,574	50,415	49,407	98.2	1.000	168	6	5	6.3	3.0	0	3,370
	HGA re-assembly Contigs	HGA(61, C(21, 8, 12x))	180,921	174,450	180,921	174,450	101.6	1.001	578	10	6	10.2	3.4	185,292	3,536
		HGA(51, M(21, 2, 50x))	211,814	195,600	211,814	195,600	101.6	1.002	662	9	6	12.8	3.6	219,922	3,545
	GAGE Contigs	GAGE-B	33,710	33,710	32,790	32,784	100.8	1.011	241	17	7	8.2	3.4	5,249	3,401
	HGA re-assembly scaffolds	HGA(61, C(21, 8, 12x))	199,144	196,419	199,144	196,419	101.6	1.001	575	10	9	10.6	3.4	185,292	0
		HGA(51, M(21, 2, 50x))	211,814	195,600	211,814	195,600	101.6	1.002	662	9	6	12.8	3.6	219,922	0
	GAGE scaffolds	GAGE-B	33,710	33,710	32,790	32,784	100.8	1.011	241	17	7	8.2	3.4	5,249	0
MIRA	Basic flow	B(21, 100x)	15,464	14,740	34,378	32,422	102.5	2.102	2,056	171	16	16.2	3.6	24,175	3,496
	HGA Preprocessing	AP(21, 2, 50x)	6,703	6,578	16,784	16,663	102.3	2.132	3,243	114	15	15.9	4.0	16,801	3,294
		C(21, 2, 50x)	113,674	107,406	112,602	107,406	101.4	1.002	127	11	5	6.5	3.1	5,461	3,510
	HGA re-assembly Contigs	HGA(101, C(21, 2, 50x))	268,342	249,690	268,342	249,690	101.9	1.002	209	16	4	14.2	3.3	57,001	3,569
		HGA(31, M(21, 8, 12x))	**443,239**	153,279	**443,239**	259,073	102.3	1.005	824	31	6	26.0	4.9	288,613	3,578
	GAGE Contigs	GAGE-B	112,926	106,563	112,926	108,689	102.3	1.029	431	106	12	9.5	3.8	23,688	3,559
	HGA re-assembly scaffolds	HGA(101, C(21, 2, 50x))	321,330	249,690	302,120	249,690	102.0	1.003	207	17	4	14.2	3.2	57,001	0
		HGA(31, M(21, 8, 12x))	**443,239**	153,279	**443,239**	259,073	102.3	1.005	824	31	6	26.0	4.9	288,613	0
	GAGE scaffolds	GAGE-B	-	-	-	-	-	-	-	-	-	-	-	-	-
MaSuRCA	Basic flow	B(91, 100x)	33,065	33,065	33,065	33,065	101.1	1.020	246	5	8	5.4	3.0	453	3,458
	HGA Preprocessing	AP(101, 2, 50x)	49,033	49,033	47,227	47,227	100.1	1.007	179	6	6	4.1	2.8	719	3,439
		C(101, 2, 50x)	152,046	152,046	152,046	144,720	100.0	1.000	108	5	5	4.2	2.9	31	3,466
	HGA re-assembly Contigs	HGA(81, C(51, 1, 100x))	307,457	246,453	307,457	246,453	101.8	1.002	407	14	3	10.4	3.5	120,450	3,541
		HGA(101, M(21, 1, 100x))	351,283	246,455	351,283	246,455	101.9	1.001	210	8	6	6.4	3.3	56,943	3,568
	GAGE Contigs	GAGE-B(99)	76,131	76,131	76,131	76,131	101.6	1.024	**173**	19	3	6.2	3.0	**811**	3,538
	HGA re-assembly scaffolds	HGA(81, C(51, 1, 100x))	307,457	246,453	307,457	246,453	101.8	1.002	406	14	4	10.2	3.3	120,450	0
		HGA(101, M(21, 1, 100x))	355,721	**355,721**	351,283	246,455	101.9	1.001	209	8	6	6.5	3.3	56,943	0
	GAGE scaffolds	GAGE-B(99)	76,131	76,131	76,131	76,131	101.6	1.024	**173**	19	3	6.2	3.0	811	0
SGA	Basic flow	B(101, 100x)	46,219	46,219	46,611	46,611	102.1	1.032	671	34	4	3.4	2.8	7,709	3,490
	HGA Preprocessing	AP(81, 2, 50x)	23,789	23,715	23,789	23,715	101.4	1.022	740	21	4	3.5	2.9	7,744	3,388
		C(101, 2, 50x)	85,247	85,247	82,971	82,971	100.6	1.001	216	4	3	4.1	2.8	5,385	3,453
	HGA re-assembly Contigs	HGA(91, C(41, 2, 50x))	240,631	120,569	240,631	120,569	101.9	1.002	324	16	6	10.4	3.2	88,422	3,549
		HGA(21, M(91, 8, 12x))	247,412	175,508	247,412	175,508	101.6	1.001	900	15	7	42.5	5.5	309,834	3,551
	GAGE Contigs	GAGE-B(65)	23,501	23,501	27,303	27,303	102.1	1.090	1,726	77	3	4.1	2.8	9,631	3,447
	HGA re-assembly scaffolds	HGA(91, C(41, 2, 50x))	145,294	120,569	240,631	120,569	101.9	1.003	323	17	6	10.4	3.4	88,422	0
		HGA(21, M(91, 8, 12x))	247,412	175,508	247,412	175,508	101.6	1.001	900	15	7	42.5	5.5	309,834	0
	GAGE scaffolds	GAGE-B(65)	27,303	27,303	27,303	27,303	100.8	1.024	647	5	2	**3.3**	2.7	**746**	0
SOAPd2	Basic flow	B(51, 100x)	28,882	28,882	27,948	27,948	101.0	1.002	444	2	0	3.1	2.7	7,151	3,368
	HGA Preprocessing	AP(41, 2, 50x)	15,426	15,426	14,567	14,567	100.2	1.002	696	2	0	3.5	2.6	6,821	3,222
		C(41, 2, 50x)	33,703	33,703	31,050	31,050	100.3	1.001	390	4	0	4.0	2.6	6,624	3,373
	HGA re-assembly Contigs	HGA(81, C(21, 8, 12x))	246,243	246,243	246,243	246,243	101.9	**1.000**	404	10	6	7.3	2.9	122,779	3,552
		HGA(91, M(21, 1, 100x))	336,603	246,263	336,603	246,263	101.8	1.001	328	9	6	6.3	3.1	96,185	3,558
	GAGE Contigs	GAGE-B(49)	71,357	68,152	71,357	65,464	101.2	1.003	244	16	35	8.2	3.3	7,695	3,453
	HGA re-assembly scaffolds	HGA(81, C(21, 8, 12x))	246,243	246,243	246,243	246,243	101.9	**1.000**	403	10	6	7.3	2.9	122,779	0
		HGA(91, M(21, 1, 100x))	336,603	246,263	336,603	246,263	101.8	1.001	327	9	7	6.4	3.1	96,185	0
	GAGE scaffolds	GAGE-B(49)	91,942	89,759	91,942	89,759	101.3	1.004	212	17	70	8.5	3.4	7,684	0
SPAdes	Basic flow	B(61, 100x)	195,243	153,030	199,486	195,243	101.6	1.002	611	10	3	6.8	2.9	186,269	3,523
	HGA Preprocessing	AP(31, 2, 50x)	106,265	92,042	106,265	93,426	101.3	1.006	561	10	6	17.9	4.0	144,717	3,499
		C(51, 4, 25x)	162,478	152,274	162,478	146,048	101.1	1.001	172	7	5	8.0	2.8	16,182	3,480
	HGA re-assembly Contigs	HGA(101, C(31, 1, 100x))	355,721	**355,721**	355,721	246,421	101.9	1.002	219	9	6	7.7	2.9	58,263	3,567
		HGA(101, M(31, 1, 100x))	355,646	246,515	355,646	**355,646**	101.9	1.001	481	10	6	8.0	2.9	183,577	3,567
	GAGE Contigs	GAGE-B(33,55,65,75,85,99)	246,623	246,623	262,160	262,160	**102.6**	1.007	1,475	5	4	5.0	3.0	442,574	**3,602**
	HGA re-assembly scaffolds	HGA(101, C(31, 1, 100x))	355,721	**355,721**	355,721	246,421	101.9	1.002	218	9	7	7.9	2.9	58,263	0
		HGA(101, M(31, 1, 100x))	355,646	246,515	355,646	**355,646**	101.9	1.001	480	10	7	8.1	2.9	183,577	0
	GAGE scaffolds	GAGE-B(33,55,65,75,85,99)	246,623	246,623	262,160	262,160	**102.6**	1.007	1,474	5	4	5.0	3.0	448,197	0
Velvet	Basic flow	B(91, 100x)	93,019	93,019	92,024	92,024	100.2	1.002	178	5	4	5.3	2.9	6,285	3,477
	HGA Preprocessing	AP(51, 2, 50x)	44,637	44,064	43,586	42,681	99.0	1.001	258	8	9	8.2	3.1	6,649	3,359
		C(41, 4, 25x)	110,407	110,407	97,303	97,303	99.9	1.004	190	4	9	9.1	3.2	6,426	3,427
	HGA re-assembly Contigs	HGA(81, C(71, 4, 25x))	243,230	127,745	243,230	127,745	101.9	1.002	399	13	7	9.5	3.0	120,450	3,544
		HGA(21, M(81, 8, 12x))	253,117	246,406	253,117	246,406	101.7	1.005	899	16	8	40.9	5.7	310,575	3,550
	GAGE Contigs	GAGE-B(97)	92,036	67,096	92,036	63,574	101.7	1.003	201	14	2	4.1	2.8	6,703	3,501
	HGA re-assembly scaffolds	HGA(81, C(71, 4, 25x))	243,230	127,745	243,230	127,745	101.9	1.002	399	13	7	9.5	3.0	120,450	0
		HGA(21, M(81, 8, 12x))	253,117	246,406	253,117	246,406	101.7	1.005	899	16	8	40.9	5.7	310,575	0
	GAGE scaffolds	GAGE-B(97)	109,996	67,096	109,996	63,574	101.7	1.003	181	27	8	4.1	2.9	6,324	0

Supplementary Table 9: 100bp HiSeq read assemblies of *V. cholera*, with reference genome size of 4,033,464bp and 3,693 genes. For metric and assembly flow descriptions see beginning of section.

Assembler	Type	Flow	N50	NA50	NG50	NGA50	Genome fraction	Duplication ratio	# contigs	# mis-assemblies	Local mis-assemblies	# mismatches per 100KB	# indels per 100KB	Unaligned length	# Genes
ABySS	Basic flow	B(51, 110x)	198,183	198,183	198,183	198,183	98.5	1.007	98	2	7	6.9	5.3	0	3,532
	HGA Preprocessing	AP(41, 2, 55x)	102,808	102,808	102,287	102,287	98.2	1.006	129	5	9	9.4	5.7	0	3,499
		C(31, 4, 27x)	102,959	102,959	98,781	98,502	97.5	1.003	178	6	26	49.8	8.4	1,015	3,433
	HGA re-assembly Contigs	HGA(51, C(41, 4, 27x))	344,337	198,492	344,337	198,492	98.2	1.002	119	9	7	30.1	6.0	3,517	3,530
		HGA(41, M(51, 8, 13x))	351,785	246,372	351,785	246,372	98.6	1.004	105	17	11	15.6	4.3	3,715	3,565
	GAGE Contigs	GAGE-B(51)	94,508	92,996	94,508	92,996	98.7	1.037	206	5	17	5.5	4.9	974	3,512
	HGA re-assembly scaffolds	HGA(51, C(41, 4, 27x))	344,337	246,263	344,337	246,263	98.3	1.002	117	10	8	30.4	6.2	3,517	0
		HGA(41, M(51, 8, 13x))	351,785	246,372	351,785	246,372	98.6	1.004	104	17	11	15.5	4.3	3,715	0
	GAGE scaffolds	GAGE-B(51)	217,596	157,127	217,596	157,127	98.7	1.040	122	15	68	7.1	5.1	974	0
CABOG	Basic flow	B(21, 110x)	10,150	10,150	9,380	9,292	92.4	1.001	527	8	7	9.6	4.2	52	2,966
	HGA Preprocessing	AP(21, 2, 55x)	8,039	8,032	7,321	7,297	91.3	1.001	663	8	5	7.6	3.1	263	2,811
		C(21, 2, 55x)	13,610	13,395	11,851	11,597	91.5	1.000	449	6	5	5.4	2.9	424	2,986
	HGA re-assembly Contigs	HGA(41, C(21, 2, 55x))	246,451	184,389	246,451	184,389	98.4	1.001	136	14	8	14.6	3.6	3,715	3,540
		HGA(41, M(21, 4, 27x))	246,376	181,216	246,376	181,216	98.2	1.004	129	15	10	15.5	3.6	3,715	3,536
	GAGE Contigs	GAGE-B	61,249	57,813	57,883	52,782	96.2	1.001	127	20	11	17.0	6.8	0	3,374
	HGA re-assembly scaffolds	HGA(41, C(21, 2, 55x))	251,410	198,497	251,410	198,497	98.4	1.006	132	15	9	15.4	3.6	3,715	0
		HGA(41, M(21, 4, 27x))	256,697	199,192	256,697	198,558	98.2	1.004	126	15	13	15.4	3.6	3,715	0
	GAGE scaffolds	GAGE-B	67,078	63,201	67,009	62,912	96.2	**1.001**	108	21	23	17.3	7.8	**0**	0
MIRA	Basic flow	B(21, 110x)	8,219	8,213	15,206	15,206	98.8	2.025	2,576	118	10	20.6	4.9	8,339	3,351
	HGA Preprocessing	AP(21, 2, 55x)	5,925	5,899	10,177	10,090	98.7	2.026	3,186	100	8	19.9	4.9	5,810	3,161
		C(21, 2, 55x)	66,538	66,538	65,248	65,248	97.6	1.001	224	5	5	7.1	3.0	35	3,451
	HGA re-assembly Contigs	HGA(41, C(21, 2, 55x))	343,940	198,410	246,455	175,807	98.3	1.001	118	12	6	13.5	3.5	3,715	3,539
		HGA(51, M(21, 2, 55x))	258,705	199,146	258,705	199,146	**98.8**	1.004	105	30	9	18.4	4.0	3,777	3,571
	GAGE Contigs	GAGE-B	89,511	87,069	92,000	89,505	**98.8**	1.038	733	54	12	11.5	3.5	15,709	3,533
	HGA re-assembly scaffolds	HGA(41, C(21, 2, 55x))	343,940	198,410	246,455	175,807	98.3	1.002	116	12	8	13.7	3.5	3,715	0
		HGA(51, M(21, 2, 55x))	258,705	199,146	258,705	199,146	**98.8**	1.004	105	30	9	18.4	4.0	3,777	0
	GAGE scaffolds	GAGE-B	-	-	-	-	-	-	-	-	-	-	-	-	-
MaSuRCA	Basic flow	B(81, 110x)	24,550	24,545	24,550	24,545	97.9	1.025	345	5	5	23.8	3.9	79	3,400
	HGA Preprocessing	AP(51, 2, 55x)	28,508	28,508	27,352	27,352	96.5	1.026	283	5	6	14.8	3.4	462	3,384
		C(41, 2, 55x)	116,510	116,488	116,488	116,488	97.1	1.000	143	8	5	17.2	3.5	0	3,439
	HGA re-assembly Contigs	HGA(81, C(31, 2, 55x))	394,532	198,888	**394,532**	198,888	98.6	**1.000**	86	13	8	11.4	3.2	280	3,561
		HGA(71, M(21, 2, 55x))	353,628	**353,590**	351,565	246,360	98.5	1.001	**84**	11	9	7.9	3.2	615	3,560
	GAGE Contigs	GAGE-B(89)	241,604	236,373	241,604	236,373	**98.8**	1.009	105	8	5	68.6	5.7	453	**3,576**
	HGA re-assembly scaffolds	HGA(81, C(31, 2, 55x))	394,532	198,888	**394,532**	198,888	98.7	**1.001**	85	14	8	11.4	3.2	280	0
		HGA(71, M(21, 2, 55x))	353,628	**353,590**	351,565	246,360	98.5	**1.001**	**83**	11	10	8.0	3.2	615	0
	GAGE scaffolds	GAGE-B(89)	246,505	236,373	246,505	236,373	**98.8**	1.009	102	10	6	68.6	5.7	453	0
SGA	Basic flow	B(31, 110x)	25,145	25,145	24,586	24,586	97.3	1.002	452	3	0	3.1	2.6	817	3,339
	HGA Preprocessing	AP(31, 2, 55x)	13,809	13,628	13,161	13,017	97.0	1.002	679	5	0	3.0	2.6	813	3,185
		C(21, 1, 110x)	27,336	27,336	21,768	21,768	83.1	1.001	326	2	0	4.3	2.8	892	2,900
	HGA re-assembly Contigs	HGA(41, C(71, 8, 13x))	228,218	180,513	228,218	180,513	98.1	1.003	136	10	5	16.2	3.5	3,715	3,527
		HGA(31, M(71, 4, 27x))	191,933	175,787	175,787	175,787	98.1	1.002	129	9	9	20.6	4.2	4,684	3,538
	GAGE Contigs	GAGE-B(65)	23,703	23,703	23,429	23,429	97.8	1.003	485	**3**	**0**	**3.2**	**2.6**	808	3,337
	HGA re-assembly scaffolds	HGA(41, C(71, 8, 13x))	228,218	192,230	228,218	192,230	98.1	1.003	133	10	7	16.6	3.7	3,715	0
		HGA(31, M(71, 4, 27x))	191,933	175,787	175,787	175,787	98.2	1.002	125	10	11	22.0	4.3	4,019	0
	GAGE scaffolds	GAGE-B(65)	23,808	23,808	23,429	23,429	96.8	1.002	430	**3**	**0**	**3.0**	**2.6**	551	0
SOAPd2	Basic flow	B(51, 110x)	21,504	21,504	20,717	20,717	97.8	1.002	462	2	0	3.3	2.6	2,024	3,300
	HGA Preprocessing	AP(41, 2, 55x)	17,770	17,770	17,562	17,545	97.5	1.002	541	2	0	3.4	2.6	1,447	3,235
		C(31, 4, 27x)	33,159	33,097	32,119	32,119	97.3	1.006	359	2	0	4.0	2.7	1,796	3,380
	HGA re-assembly Contigs	HGA(61, C(31, 8, 13x))	350,677	180,774	350,677	180,774	98.3	1.001	119	7	2	8.8	3.0	3,342	3,539
		HGA(61, M(31, 4, 27x))	343,953	180,774	180,774	152,171	98.3	1.001	122	7	1	7.8	3.1	3,114	3,532
	GAGE Contigs	GAGE-B(51)	135,118	106,454	125,939	106,454	98.1	1.003	139	21	38	12.5	3.7	2,076	3,491
	HGA re-assembly scaffolds	HGA(61, C(31, 8, 13x))	350,677	260,601	350,677	198,506	98.3	**1.001**	116	7	4	8.8	3.0	3,342	0
		HGA(61, M(31, 4, 27x))	343,953	198,506	302,854	198,506	98.3	**1.001**	120	7	3	7.9	3.0	3,114	0
	GAGE scaffolds	GAGE-B(51)	200,529	181,109	181,109	181,109	98.1	1.004	114	21	64	12.5	3.7	2,494	0
SPAdes	Basic flow	B(41, 110x)	176,065	176,065	176,065	151,041	98.1	1.002	172	5	5	10.4	3.3	3,715	3,511
	HGA Preprocessing	AP(41, 2, 55x)	103,942	97,862	98,645	96,551	98.1	1.003	187	6	4	11.4	3.2	1,843	3,499
		C(51, 4, 27x)	245,959	245,959	245,959	245,959	97.9	1.001	120	4	6	7.2	3.3	808	3,505
	HGA re-assembly Contigs	HGA(51, C(31, 8, 13x))	**572,670**	199,693	265,745	199,693	98.3	1.003	119	11	7	12.1	3.9	3,349	3,544
		HGA(71, M(41, 4, 27x))	352,971	246,125	246,127	246,125	98.4	1.002	87	14	7	10.4	3.5	2,576	3,554
	GAGE Contigs	GAGE-B(33,55,65,75,85,99)	83,518	83,518	77,074	77,065	98.3	1.003	205	4	2	5.5	2.9	4,085	3,504
	HGA re-assembly scaffolds	HGA(51, C(31, 8, 13x))	**572,670**	246,397	265,745	246,397	98.3	1.003	117	11	8	12.7	4.0	3,349	0
		HGA(71, M(41, 4, 27x))	352,971	246,125	246,127	246,125	98.4	1.002	86	14	8	10.5	3.5	2,576	0
	GAGE scaffolds	GAGE-B(33,55,65,75,85,99)	98,274	94,762	98,274	94,762	98.7	1.016	120	20	16	8.9	5.2	3,819	0
Velvet	Basic flow	B(41, 110x)	46,929	46,923	46,456	45,371	96.6	1.001	238	8	13	13.1	3.4	954	3,385
	HGA Preprocessing	AP(41, 2, 55x)	34,987	34,987	33,604	33,604	96.4	1.001	313	6	7	10.9	3.0	956	3,331
		C(21, 4, 27x)	80,024	78,122	78,122	74,326	96.4	1.000	192	6	5	20.6	7.4	1,266	3,395
	HGA re-assembly Contigs	HGA(41, C(31, 8, 13x))	311,244	215,454	311,244	200,509	98.2	1.002	125	11	9	16.9	3.1	3,715	3,531
		HGA(51, M(31, 4, 27x))	356,011	246,499	356,011	**246,499**	98.3	1.002	108	14	8	12.0	3.2	3,267	3,541
	GAGE Contigs	GAGE-B(49)	40,877	40,877	40,085	39,462	98.0	1.001	261	5	6	4.5	3.4	1,069	3,404
	HGA re-assembly scaffolds	HGA(41, C(31, 8, 13x))	311,244	246,407	311,244	246,407	98.2	1.006	121	12	11	18.5	3.1	3,715	0
		HGA(51, M(31, 4, 27x))	356,011	246,499	356,011	**246,499**	98.3	1.002	107	14	9	12.1	3.2	3,267	0
	GAGE scaffolds	GAGE-B(49)	172,545	171,505	172,545	171,505	98.0	1.006	124	11	129	5.3	4.4	981	0

Supplementary Table 10: Results of assembling simulated error-free reads

K	Flow	NGA50	# contigs	# misassemblies	Local misassemblies	# mismatches per 100Kb	# indels per 100Kb	Unaligned length
21	B(21, 110x)	287,266	44	1	27	4.03	0.63	0
	C(21, 4, 27x)	299,809	48	2	28	3.05	0.77	0
	HGA(21, C(21, 8, 13x))	339,413	40	1	32	2.93	0.69	0
	HGA(21, M(81, 4, 27x))	972,133	22	2	4	3.36	0.24	0
31	B(31, 110x)	392,422	38	0	17	2.08	0.41	0
	C(31, 4, 27x)	383,207	44	0	19	1.76	0.4	0
	HGA(31, C(81, 8, 13x))	392,454	38	0	18	1.95	0.47	0
	HGA(31, M(81, 4, 27x))	972,187	22	1	4	3.03	0.24	0
41	B(41, 110x)	654,250	30	0	11	1.14	0.29	0
	C(41, 8, 13x)	417,031	36	0	15	0.79	0.32	0
	HGA(41, C(91, 8, 13x))	654,250	30	0	11	1.14	0.29	0
	HGA(41, M(81, 4, 27x))	972,243	18	0	4	1.95	0.26	0
51	B(51, 110x)	654,246	28	0	8	0.39	0.18	0
	C(51, 2, 55x)	654,260	32	0	9	0.63	0.24	0
	HGA(51, C(21, 4, 27x))	654,269	33	0	6	2.12	0.26	0
	HGA(51, M(81, 4, 27x))	972,263	18	0	4	1.75	0.26	0
61	B(61, 110x)	684,726	27	0	7	0.12	0.18	0
	C(61, 2, 55x)	684,630	30	0	7	0.22	0.18	0
	HGA(61, C(91, 8, 13x))	684,726	27	0	7	0.12	0.18	0
	HGA(61, M(81, 4, 27x))	972,283	17	0	4	1.3	0.28	0
71	B(71, 110x)	684,901	25	0	2	0.06	0.06	0
	C(71, 2, 55x)	684,630	30	0	4	0.1	0.1	0
	HGA(71, C(91, 8, 13x))	684,901	25	0	2	0.06	0.06	0
	HGA(71, M(81, 4, 27x))	972,458	17	0	4	1.26	0.28	0
81	B(81, 110x)	684,921	26	0	2	0.06	0.06	0
	C(81, 4, 27x)	520,881	32	1	3	2.79	0.24	0
	HGA(81, C(91, 8, 13x))	684,921	26	0	2	0.06	0.06	0
	HGA(81, M(81, 4, 27x))	972,478	19	0	4	1.26	0.28	0
91	B(91, 110x)	41,858	204	0	2	0.02	0.06	0
	C(91, 1, 110x)	74,259	209	0	0	0.36	0	0
	HGA(91, C(61, 8, 13x))	685,023	25	0	2	0.12	0.06	0
	HGA(91, M(81, 4, 27x))	972,580	20	0	4	1.26	0.26	0
Max	**B(81, 110x)**	**684,921**	**26**	**0**	**2**	**0.06**	**0.06**	**0**
	C(71, 2, 55x)	**684,630**	**30**	**0**	**4**	**0.1**	**0.1**	**0**
	HGA(91, C(61, 8, 13x))	**685,023**	**25**	**0**	**2**	**0.12**	**0.06**	**0**
	HGA(91, M(81, 4, 27x))	**972,580**	**20**	**0**	**4**	**1.26**	**0.26**	**0**

Note that, even with error free reads, the repeats still be an assembly problem and will induce ambiguities pattern such as cycles and false branching; Now the flows of basic, combining, and HGA using combined contigs; all of them and even with long kmer sizes couldn't perform similarly to the HGA method using merged contigs and long kmer size. For HGA using merged contigs; the contigs where of coverage P where P is the number of partitions, this is unlike HGA using combined contigs where the contigs is of coverage 1, furthermore when combining contigs of coverage P to create contigs of coverage 1, we may misassemble or disregard a true contigs. This justify why truly HGA using merged is outperform HGA using combined contigs. Laslty, HGA using merged contigs is unlike to the combining flow where the flow is involving assembling the whole reads, and unlike the basic flow where the flow involve just the whole reads with no contigs..

Contigs combining experiments

As attempt to resolve this; we run minimus2 using large minimum-overlap value; but we noticed that the output of large overlap value has less genome fraction results; and this is explained because true contigs which overlap truly using low overlap value, will be ignored and not outputted. Moreover even with large overlap the duplication ratio and the misassemblies events didn't decrease much, because we still have non-contiguous contigs that share large x-mer (if not in form of repeats; still some contigs may have assembled the same large region, that form large repeat). In conclusion, we found minimus2 on combining assembly contigs is not effective and misleading; and decided to switch to whether a string graph assembler or a de Bruijn graph assembler.

We run velvet using x-mer sizes 31, 51, 71, 91; hierarchically (binary tree model) and over all partitions' contigs (merging all contigs together). As well, most assembler output assembly with expected coverage equal to 1; so when combining 2, 4 and 8 partitions we input to velvet the expected coverage to be 2, 4 and 8; respectively; instead of the default value which velvet use, {\it auto}. This actually makes the assembly process informative in advance; and therefore helps in detecting repeats, errors and paths finding. When assembling contigs hierarchically in binary tree model, whether using velvet or minimus2; the results show less genome fraction than when assembling all partitions' contigs together; and this justified because some contigs may combine better with other partition's contigs. So we tend to do the contigs assembly by merging all partitions' contigs and then assemble them all. Moreover, all results of running velvet as combiner using 31, 51, 71, and 91 as x-mer length were varying; but the results with 31 value, mostly were the best; so, in this thesis, all contigs assembly processes were computed using velvet with 31 as kmer length.

Supplementary Table 11: Sample results of contigs assembly, R. sphaeroides HiSeq, and using velvet. Second columns is the results of basic assembly flow; the following columns is the contigs assembly flow, and each column has a column for the tool used to assemble the partitions contigs, velv31 for velvet using kmer=31; Min20 & 80 for minimus2, with minimum overlap value=20 & 80.

Metrics	B(41, 210x)	C(41, 2, 105x)			C(41, 4, 52x)			C(41, 8, 26x)		
		Velv31	Min20	Min80	Velv31	Min20	Min80	Velv31	Min20	Min80
Number of contigs	959	625	336	372	748	426	479	1127	630	740
NGA50	8,704	14,751	33,203	28,718	11,763	28,114	22,575	7,767	18,074	14,252
Genome fraction %	97.74	97.43	98.24	98.15	97.59	98.63	98.53	97.45	98.43	98.4
Duplication ratio	1.003	1.002	1.023	1.029	1.002	1.091	1.058	1.003	1.134	1.103
Global misassemblies	15	1	17	14	1	20	16	1	37	38
Local misassemblies	6	4	15	9	3	16	6	2	25	13
Mismatches per 100kb	17	10	11	11	7	15	13	5	22	21
Indels per 100kb	1	1	1	1	0	1	1	0	1	1
# of unaligned contigs	1	1	2	4	0	1	7	1	1	3

Table 11, shows sample result of contigs assembly using velvet, with 31 kmer; and using minimus2 with 20, 80 overlap values. Firstly, it's clear that the NGA50 values increase considerably with both tools; genome fraction results kept the same with velvet and increase slightly with minimus2. Although both tools increased the NGA50, but each tools show notable differences in the other metrics; namely, duplication ratio, using velvet, kept in similar range compared to the basic flow; while the duplication ratio for both minimus2's assembly (20 and 40 as overlap values) increased *dramatically*; in addition, errors in the contigs in terms of global misassemblies, local misassemblies, mismatches and indels; increase *considerably* with minimus2 while kept the same or even decreased using velvet.

As results, there are two important notes: firstly, the dramatic increase in the error metrics (global misassemblies, local misassemblies, mismatches, indels and unaligned contigs) as well as the duplication ratio, indicate that minimus2 might may have combined contigs falsely, such as two (or more) contigs that are true (aligned) but not contiguous, and share kmer of length 20 or 80; may connected using minimus2 and form a new single contigs; this actually may increase the NGA50 value but false positively. Secondly, it's important for the reassembly process, where we re-assemble a contigs with the reads; to have these contigs as correct as possible, in order to get better reassembly results. Thus, we ignored minimus2 and used velvet, for the contigs assembly process.

Testing assemblers for the re-assembly step

For this step we mainly considered SPAdes and velvet, as both are de Bruijn graph based and both take long sequences (contigs) as input data. Because this step is intermediate step not final, we didn't compare the results based on the NA50 correspondent to the highest N50, as the comparison tables above. For this comparison we compared the re-assembly results produced by using SPAdes or Velvet as re-assembler, directly based on the highest NA50.

The commands used to compute the re-assembly process using SPAdes and Velvet respectively are:

```
$spade_dir/spades.py -k $kmer --12 Whole_reads.fastq --trusted-contigs merged(or combined).fasta -o ./
```

```
$velvet_path/velveth ./ $kmer -fasta -long merged(or combined).fasta -fastq -shortPaired Whole_reads.fastq
```

```
$velvet_path/velvetg ./ -exp_cov auto -ins_length $mean -ins_length_sd $std -scaffolding no
```

Note that during the installation of Velvet, Velvet should be installed with option `'LONGSEQUENCES=1'` in the `make` command, to allow Velvet to accept contigs (long sequences) as input; as well and `'MAXKMERLENGTH=111'` as the default max kmer length in Velvet installation is 31.

Supplementary Table 12: 100bp HiSeq read of *M. abscessus*, with reference genome size of 5,090,401bp. The test results are reported based on the NA50 of the re-assembly process (using the combined contigs *CC* and the merged contigs *MC*) using SPAdes and Velvet as *re-assembler*. The test was conducted over all combinations of kmers used in the partitioning step *PK* and the kmers used in the re-assembly step *RK;* as well using 2, 4, and 8 partitions *P*.

		Rk	21		31		41		51		61		71		81		91	
CT	P	Pk	SPAdes	Velvet	SPAdes	Velvet	SPAdes	Velvet	SPAdes	Velvet	SPAdes	Velvet	SPAdes	Velvet	SPAdes	Velvet	SPAdes	Velvet
CC	2	21	123,099	17,254	231,671	21,494	225,668	36,943	225,656	118,393	175,766	104,682	143,820	68,831	98,264	40,037	46,765	39,240
		31	152,087	16,577	261,099	21,494	231,757	36,102	225,660	206,566	232,830	186,960	232,849	202,126	210,217	172,373	189,572	136,708
		41	147,777	16,869	247,618	21,494	248,116	36,059	247,282	179,862	247,233	172,262	232,849	225,656	232,964	182,822	225,655	136,709
		51	143,186	16,531	245,697	21,356	226,397	36,035	230,579	179,861	232,697	185,665	225,656	209,612	225,655	166,001	176,037	120,931
		61	195,121	16,869	343,659	21,343	260,075	34,956	273,546	147,220	273,546	165,991	242,913	166,001	203,144	147,203	154,029	127,866
		71	173,017	16,577	343,659	22,567	260,196	42,189	273,579	125,786	247,266	148,435	225,655	166,001	203,143	147,759	190,177	133,740
		81	153,561	16,577	278,346	21,494	232,877	36,782	247,408	125,786	233,930	127,657	214,862	147,159	191,111	122,627	122,611	119,763
		91	143,186	16,577	232,801	22,785	156,564	38,850	156,496	110,393	141,421	110,681	162,440	110,813	125,801	79,781	92,700	79,678
	4	21	147,154	16,869	260,186	21,494	226,076	36,059	232,411	210,139	232,830	185,683	226,577	180,140	225,656	165,235	189,566	120,772
		31	148,464	16,531	313,075	21,343	278,377	36,059	278,387	206,566	278,397	185,683	243,843	181,799	225,656	166,001	221,466	127,866
		41	148,464	16,577	313,188	21,494	260,196	36,102	260,398	179,861	247,107	156,652	243,843	156,662	225,656	149,673	190,184	119,750
		51	152,286	17,254	313,188	20,838	236,908	36,059	245,440	149,669	278,397	165,991	243,845	149,665	192,651	149,665	172,817	119,735
		61	195,121	16,577	343,659	21,048	236,908	34,669	247,249	147,150	246,993	127,657	242,913	125,878	147,171	113,294	122,550	110,659
		71	198,820	16,577	343,659	22,785	237,206	41,270	273,582	125,786	236,798	125,796	213,246	121,436	174,038	122,889	125,899	106,790
		81	195,121	16,531	313,228	21,343	225,668	35,998	245,599	99,051	156,737	95,090	187,805	99,024	115,838	83,318	99,016	79,784
		91	147,154	16,577	230,762	21,601	189,669	34,420	156,364	83,636	121,832	65,093	104,393	60,741	65,896	50,539	53,343	45,943
	8	21	146,809	17,077	247,618	23,494	225,668	42,189	233,186	149,689	233,193	149,673	225,656	165,178	175,902	142,432	142,441	118,813
		31	147,154	16,577	234,296	21,265	225,668	34,420	234,296	125,786	187,685	127,657	187,069	104,830	187,179	127,466	155,508	104,759
		41	148,410	17,254	278,346	22,071	225,668	40,798	222,355	118,393	147,176	104,839	174,949	94,968	160,295	90,524	103,310	88,121
		51	147,118	16,577	278,313	22,567	189,669	37,590	156,727	79,404	121,542	57,507	83,037	39,029	53,172	28,770	29,550	29,330
		61	112,100	17,419	209,890	20,480	156,717	32,873	125,787	47,961	77,036	23,326	34,205	7,516	9,645	1,578	895	344
		71	187,108	16,577	278,346	21,265	225,668	34,420	156,364	75,439	147,708	70,132	98,786	59,101	82,886	52,946	55,722	44,038
		81	153,280	16,531	232,801	22,529	226,430	35,479	157,176	70,051	141,428	56,768	88,620	35,914	46,037	24,300	24,549	20,923
		91	185,542	17,254	232,801	22,071	189,607	33,160	139,584	62,020	104,277	31,823	45,365	12,303	17,036	4,500	3,859	2,755
Maxes			***198,820***	*17,419*	***343,659***	*23,494*	***278,377***	*42,189*	***278,387***	*210,139*	***278,397***	*186,960*	***243,845***	*225,656*	***232,964***	*182,822*	***225,655***	*136,709*
MC	2	21	172,184	17,231	260,186	21,494	225,668	36,102	225,668	134,634	225,668	149,673	243,843	142,539	210,217	105,368	105,381	63,445

		31	246,906	19,094	312,976	22,558	313,248	42,581	313,268	206,566	313,288	185,683	313,122	202,127	313,076	161,637	278,423	119,755
		41	278,303	19,805	313,035	22,529	278,569	39,656	278,387	163,653	278,397	225,668	278,386	226,392	278,609	226,392	278,423	132,568
		51	198,820	19,805	343,659	22,422	260,196	36,102	278,387	179,861	278,589	209,821	278,386	207,809	246,999	153,244	246,623	138,346
		61	198,820	19,997	343,659	22,690	260,196	36,782	273,549	163,645	278,589	155,454	226,646	178,193	273,632	178,192	225,656	154,844
		71	198,820	19,805	343,659	23,721	237,214	43,327	273,582	149,689	278,397	147,170	226,646	144,900	225,656	147,190	225,656	147,200
		81	195,121	18,839	343,659	22,529	226,358	37,876	225,656	109,384	214,894	99,050	186,389	75,837	125,894	75,300	108,085	68,566
		91	165,917	18,624	247,300	22,422	180,381	37,966	156,496	88,122	142,531	75,610	147,186	69,570	121,447	59,158	68,708	50,516
	4	**21**	174,815	19,843	260,186	22,402	232,877	45,610	226,004	192,488	226,004	205,057	226,557	157,055	243,805	157,055	243,805	98,350
		31	195,121	19,774	343,659	22,529	278,569	43,912	278,329	147,927	278,329	205,057	278,598	202,126	278,417	168,606	226,092	127,717
		41	198,820	19,843	343,659	22,529	260,196	38,611	260,189	146,446	278,329	149,744	278,308	157,596	226,392	149,731	225,656	147,190
		51	195,121	19,843	344,789	22,422	260,196	38,717	273,541	129,391	278,397	127,657	224,930	125,878	214,875	127,697	208,879	109,628
		61	195,124	19,498	343,659	22,229	236,908	36,102	273,549	149,689	246,993	154,891	242,913	127,675	214,875	110,682	163,932	104,857
		71	195,124	18,879	313,089	23,494	232,792	41,791	247,214	149,689	214,894	94,948	186,389	94,968	173,271	83,153	147,191	79,794
		81	225,667	18,879	313,850	22,597	226,216	41,270	174,008	85,051	156,737	79,444	147,186	64,778	108,077	61,087	87,246	53,956
		91	209,975	18,306	230,762	22,567	157,176	38,702	155,451	79,398	119,513	52,322	91,102	41,548	51,377	26,991	33,560	21,308
	8	**21**	172,184	19,498	260,186	22,597	225,668	42,698	232,411	102,420	232,830	118,504	226,175	104,741	210,091	78,546	200,568	59,115
		31	210,016	18,833	278,346	21,601	226,325	35,998	225,656	100,178	185,479	110,681	185,489	94,968	175,902	80,377	163,902	70,161
		41	227,190	18,879	314,356	22,529	225,668	39,522	247,282	94,928	156,737	99,459	170,861	80,006	155,502	68,088	110,488	65,610
		51	175,712	18,780	231,671	21,494	189,574	35,905	177,340	81,495	139,584	69,506	95,165	51,880	62,348	29,479	33,011	24,373
		61	112,100	17,584	209,890	20,246	156,717	32,873	125,787	47,961	77,036	23,586	34,205	7,516	9,481	1,587	904	346
		71	225,667	18,306	247,266	22,597	185,626	38,132	156,397	75,503	147,176	70,072	85,087	49,002	64,063	38,121	40,562	30,362
		81	225,667	18,390	278,346	23,494	189,669	36,500	156,455	69,940	121,075	51,697	66,427	31,574	41,513	17,766	19,738	13,572
		91	225,667	17,577	232,801	21,828	189,607	35,423	139,584	61,085	88,395	31,517	47,918	12,529	17,712	4,382	3,705	-
Maxes			***278,303***	*19,997*	***344,789***	*23,721*	***313,248***	*45,610*	***313,268***	*206,566*	***313,288***	*225,668*	***313,122***	*226,392*	***313,076***	*226,392*	***278,423***	*154,844*

Supplementary Table 13: 100bp HiSeq read of *V. cholera*, with reference genome size of 4,033,464bp. The test results are reported based on the NA50 of the re-assembly process (using the combined contigs *CC* and the merged contigs *MC*) using SPAdes and Velvet as *re-assembler*. The test was conducted over all combinations of kmers used in the partitioning step *PK* and the kmers used in the re-assembly step *RK;* as well using 2, 4, and 8 partitions *P*.

		Rk	**21**		**31**		**41**		**51**		**61**		**71**		**81**		**91**	
CT	**P**	**Pk**	**SPAdes**	**Velvet**	**SPAdes**	**Velvet**	**SPAdes**	**Velvet**	**SPAdes**	**Velvet**	**SPAdes**	**Velvet**	**SPAdes**	**Velvet**	**SPAdes**	**Velvet**	**SPAdes**	**Velvet**
CC	**2**	**21**	60,417	16,994	137,240	35,171	198,468	72,349	246,268	95,621	246,417	106,179	246,360	140,210	246,381	151,581	98,744	66,247
		31	61,392	16,551	106,150	33,716	198,848	72,349	199,733	97,164	199,311	110,792	225,933	126,179	198,888	129,194	135,310	81,800
		41	60,567	16,719	107,884	33,716	198,492	61,861	187,797	91,944	187,895	102,721	187,781	135,138	246,591	152,441	140,875	116,602
		51	68,319	19,742	129,123	42,104	188,115	75,457	180,764	112,883	198,512	113,741	213,452	125,294	213,395	125,302	135,085	111,932
		61	68,319	16,498	129,147	34,264	198,492	72,349	199,637	90,974	201,613	94,635	163,446	81,845	127,866	88,571	105,380	75,819
		71	68,319	19,343	127,817	40,280	246,377	75,468	246,397	86,148	201,646	81,069	200,332	67,944	114,971	89,132	92,002	75,748
		81	71,631	19,725	124,943	40,218	199,187	74,732	199,637	87,800	206,201	70,048	151,486	66,733	102,129	68,163	85,213	66,563
		91	68,323	16,461	109,139	34,029	175,807	72,349	198,505	80,945	199,311	66,712	109,010	63,881	95,057	51,714	59,995	43,296
	4	**21**	65,962	21,176	130,154	42,104	198,821	76,413	199,600	97,164	191,851	112,903	216,591	112,934	124,685	112,943	99,673	71,778
		31	68,319	17,358	114,798	34,029	162,815	77,422	180,696	90,974	167,903	91,964	151,486	98,700	126,782	96,302	98,584	94,782
		41	60,512	16,815	129,147	33,716	152,314	62,973	199,757	90,974	199,323	91,964	132,786	91,984	124,685	98,700	113,860	90,399
		51	71,631	19,672	143,792	40,218	199,722	72,805	180,696	86,148	180,716	90,445	127,940	82,806	115,871	75,317	85,213	71,439
		61	60,567	19,725	136,129	39,781	224,346	63,686	199,712	80,945	199,311	70,242	136,986	71,416	96,380	59,062	69,903	58,902
		71	77,183	20,405	168,338	35,768	197,503	55,498	198,858	75,614	210,179	64,833	117,445	66,733	102,635	53,630	58,193	42,794
		81	72,656	16,498	168,338	32,712	192,987	59,585	192,953	63,214	180,716	63,224	117,445	50,913	90,439	39,077	46,220	36,659
		91	72,656	19,343	143,808	35,327	167,847	63,202	198,505	65,786	126,399	51,287	100,643	37,769	64,783	25,329	29,641	20,597
	8	**21**	71,631	20,405	136,492	38,041	199,590	70,443	343,935	81,818	180,716	71,396	153,810	74,893	81,084	67,036	62,655	46,009
		31	65,501	18,120	102,639	38,030	199,554	55,953	208,395	65,971	126,764	61,180	117,439	62,568	78,018	54,554	55,905	47,572
		41	70,631	16,719	129,147	32,712	180,988	59,301	180,696	63,212	152,103	51,646	109,549	45,170	68,199	32,513	37,132	31,763
		51	90,617	16,994	143,746	32,711	198,477	58,416	152,969	48,727	108,757	32,527	94,604	20,328	31,485	9,667	9,424	9,845
		61	52,111	16,994	92,125	32,985	176,065	51,596	135,971	36,198	79,284	18,398	59,574	8,771	16,717	2,683	1,692	447
		71	83,702	16,551	137,067	32,712	197,840	52,911	202,016	56,578	190,593	45,959	108,767	33,735	62,386	23,154	22,640	17,103
		81	88,178	19,698	141,523	32,968	180,676	62,745	152,161	55,599	159,284	41,120	92,721	29,496	47,084	16,503	17,482	11,999
		91	77,151	16,994	92,042	35,247	176,065	51,596	135,971	39,468	92,241	26,389	90,503	13,499	24,223	5,605	4,474	2,342
Maxes			***90,617***	*21,176*	***168,338***	*42,104*	***246,377***	*77,422*	***343,935***	*112,883*	***246,417***	*113,741*	***246,360***	*140,210*	***246,591***	*152,441*	***140,875***	*116,602*
MC	**2**	**21**	95,555	22,719	140,075	37,179	199,163	64,314	246,465	87,298	199,201	106,179	353,590	126,179	246,243	97,421	171,655	63,641
		31	86,581	20,582	129,983	35,142	174,109	62,897	199,757	87,277	207,823	101,302	207,833	104,871	198,888	109,976	174,535	90,285
		41	95,555	22,023	135,629	36,056	156,235	61,861	199,200	91,944	199,210	101,357	201,656	129,184	152,355	129,194	174,535	129,204
		51	93,352	22,023	153,000	45,895	181,141	75,468	199,757	113,692	207,823	91,964	246,125	91,984	152,127	95,008	135,211	102,299
		61	82,046	21,721	165,986	34,029	165,996	64,545	152,161	90,974	199,210	68,980	198,525	72,838	142,280	74,386	107,841	72,672
		71	93,352	22,827	151,742	42,027	181,035	72,719	198,505	90,624	152,335	66,753	108,489	64,828	95,940	59,817	68,941	51,953
		81	106,146	23,920	151,742	42,121	180,794	72,349	174,185	85,054	153,030	64,828	108,767	57,659	79,566	47,610	68,181	42,410
		91	102,474	22,402	198,754	35,871	198,495	62,913	174,300	71,856	152,090	59,299	126,774	45,703	71,438	38,748	42,140	27,040
	4	**21**	95,555	23,439	168,042	44,321	198,468	75,457	344,908	107,308	199,323	96,281	216,591	107,431	216,538	89,890	113,860	63,641
		31	88,259	24,917	137,122	33,716	152,668	62,957	207,813	89,964	172,778	79,489	152,698	81,086	132,867	79,004	124,757	79,998
		41	89,822	22,481	153,275	34,029	137,109	71,603	156,160	90,854	199,323	75,926	152,100	83,609	124,685	83,628	124,695	69,130
		51	103,944	23,439	156,140	42,121	156,150	63,507	156,160	87,277	156,204	71,504	137,253	70,013	115,871	66,732	85,213	53,666
		61	98,525	23,439	156,122	40,155	201,593	59,374	156,142	84,131	152,171	66,940	108,834	66,939	92,147	51,646	64,899	48,191
		71	126,698	22,023	156,122	37,232	199,283	62,897	156,142	75,630	144,061	66,945	104,108	63,984	85,213	33,670	53,813	34,854
		81	126,698	21,815	151,761	30,960	180,754	62,092	152,274	63,582	144,061	57,398	92,093	40,325	68,199	31,010	45,246	26,137
		91	128,460	22,976	129,147	35,247	198,492	60,193	166,580	58,106	98,701	43,090	92,721	30,521	58,988	18,929	25,502	12,899
	8	**21**	106,146	21,434	130,164	33,971	167,863	52,263	199,303	62,790	166,947	49,665	126,808	47,103	90,439	33,167	68,181	31,858
		31	106,605	21,815	146,550	31,624	156,147	58,698	152,274	62,758	126,764	55,690	95,213	50,068	81,245	33,670	62,686	32,691
		41	126,698	22,481	152,274	31,633	175,807	62,712	198,505	66,133	152,103	43,424	104,108	39,678	69,598	30,552	42,210	29,420
		51	95,885	21,816	177,014	32,740	165,996	52,263	152,103	51,629	135,555	32,790	92,721	25,148	41,066	13,199	12,247	9,754
		61	53,982	16,994	92,143	35,247	176,065	52,263	135,971	36,847	79,284	18,733	59,574	8,829	17,098	2,727	1,708	453
		71	198,477	22,408	197,767	31,624	207,776	51,515	151,466	59,764	116,287	51,211	91,988	30,969	58,779	20,087	21,829	12,631
		81	136,067	21,812	136,129	33,101	198,497	62,094	151,051	54,856	130,126	40,158	90,533	26,857	52,350	13,939	15,262	8,313
		91	91,986	19,672	105,866	30,948	167,847	51,935	130,716	36,198	92,102	26,389	70,334	13,608	24,807	5,592	4,667	-
Maxes			***198,477***	*24,917*	***198,754***	*45,895*	***207,776***	*75,468*	***344,908***	*113,692*	***207,823***	*106,179*	***353,590***	*129,184*	***246,243***	*129,194*	***174,535***	*129,204*

Assembly commands:

The values of `$mean` and `$std` are reported in Table 1.

ABySS

```
$abyss_dir/abyss-pe k=$kmer l=1 n=5 s=200 name=asm in=file.fastq
```

CABOG

```
echo "unitigger = bog" > config

$cabog_dir/fastqToCA  -insertsize $mean $std -libraryname reads -mates
file.fastq > ./reads.frg
$cabog_dir/runCA -d ./ -p asm -s config ./reads.frg
```

MIRA

```
echo project = MyFirstAssembly > config
echo job = genome,denovo,accurate >> config
echo readgroup = DataIlluminaPairedLib >> config
echo data =file1.fastq file2.fastq >> config
echo technology = solexa >> config
echo template_size =  $mean $std >> config
echo "segment_placement = ---> <---" >> config
echo "parameters= -NW:cmrnl=warn" >> config
$mira_path/mira config
```

MaSuRCA

```
echo PATHS > config
echo JELLYFISH_PATH=$MaSuRCA_path  >> config
echo SR_PATH=$MaSuRCA_path >> config
echo CA_PATH=$cabog_path >> config
echo END >> config
echo DATA >> config
echo PE= p1 $mean $std file1.fastq file2.fastq >> config
echo END >> config
echo PARAMETERS >> config
echo GRAPH_KMER_SIZE=$kmer >> config
echo NUM_THREADS=8 >> config
echo JF_SIZE=2000000000 >> config
echo END  >> config
perl $MaSuRCA_path/masurca config
./assemble.sh
```

SGA

```
sga preprocess  --pe-mode 1 -o reads.pp.fastq file1.fastq file2.fastq

sga index --algorithm=ropebwt -t 8 reads.pp.fastq
sga correct -k $k -t 8 -o reads.ec.fastq reads.pp.fastq
sga index --algorithm=ropebwt -t 8 reads.ec.fastq
sga filter -t 8 reads.ec.fastq
sga overlap -m $kmer -t 8 reads.ec.filter.pass.fa
sga assemble -o primary reads.ec.filter.pass.asqg.gz
```

SOAPd2

```
echo [LIB] > config
echo avg_ins=$mean  >> config
echo reverse_seq=0 >> config
echo asm_flags=1 >> config
```

```
echo rank=1 >> config
echo q1=file1.fastq >> config
echo q2=file2.fastq >> config
$soapdenov2_path/SOAPdenovo-127mer all -K $kmer -F -R -E -s config -o asm
-p 4 >> SOAPdenovo.lo
```

SPAdes

```
$spade_dir/spades.py -t 8 -k ${kmer} --12 file.fastq -o ./
```

Velvet

During the installation of Velvet, Velvet should be installed with option `'LONGSEQUENCES=1'` in the `make` command, to allow Velvet to accept contigs (long sequences) as input; as well and `'MAXKMERLENGTH=111'` as the default max kmer length in Velvet installation is 31.

Now to run the assembly commands:

```
$velvet_path/velveth ./ $kmer -fastq -shortPaired file.fastq
$velvet_path/velvetg ./ -exp_cov auto -ins_length $mean -ins_length_sd
$std -scaffolding no
```

Interleaving the paired reads for the datasets tested in this thesis

If a reader will test HGA method using the datasets in this thesis, then in order to have the same results as the reported results, please note the following.

- Firstly in our tests we used the interleaved (interlaced) fastq file, rather than the 2 paired fastq files.
- Since HGA method involve a partitioning step, note that the order of the reads in the interleaved fastq file should be in the *same order* as was tested in this thesis. So, after downloading a datasets from http://ccb.jhu.edu/gage_b/ ; the downloaded pair fastq files *must* be interleaved using this script (not any other scripts to ensure the order of the reads is the same as was tested in this thesis), https://gist.github.com/ngcrawford/2232505 . The script runs as follow:

```
python interleave_fastq.py file_1.fastq file_2.fastq  file.fastq
```

Partitioning the reads sets:

We assumed the reads in the datasets are already randomized, so we sequentially selected the reads for each partition.

Note: If a reader will test HGA methods using the datasets in this thesis, then in order to get the same reported results in this thesis, reads datasets (fastq file) should be whether *raw* or *clean* as recommended in table 2 for the assembler that will be used to assemble the partitions.

Combining contigs command

Firstly we merge all parts' contigs into one file, e.g. merged_contigs.fa, and then we run the following command:

```
$velvet_path/velveth ./ 31 -fasta -long merged_contigs.fa
$velvet_path/velvetg ./ -exp_cov $parts -scaffolding no
```

Note: we input the *number of parts* as the `-exp_cov` value. Also, Velvet should be installed with option `'LONGSEQUENCES=1'` in the `make` command, to allow Velvet to accept contigs (long sequences) as input; as well `'MAXKMERLENGTH=111'` as the default max kmer length in Velvet installation is 31.

Re-assembly command

We add the parameter --trusted-contigs and give it the fasta file of the merged or the combined contigs.

```
$spade_dir/spades.py -t 4 -k ${kmer} --12 file.fastq --trusted-contigs
combined(or merged)_contigs.fa -o ./
```

Note: If a reader will test HGA methods using the datasets in this thesis, then in order to get the same reported results in this thesis, for the re-assembly step reads datasets (fastq file) should be whether *raw* or *clean* as recommended in table 2 for SPAdes assembler, as the re-assembly process was performed using SPAdes assembler.

QUAST command:

```
quast.py -R genome.fasta  --min-contig 200 contigs.fa   -G genes.gff
```

Platform:

We tested the method on Linux platform, AMD Opteron(tm) 2.4GH, 256GB Memory, 64 core. Python v2.7.6.