HIGH PERFORMANCE ALGORITHMS FOR K-MER COUNTING AND GENOMIC READ OVERLAP FINDING

A Dissertation
Presented to
The Academic Faculty

By

Shaowei Zhu

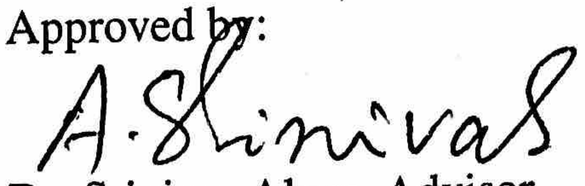
In Partial Fulfillment
of the Requirements for the Research Option Program
for Bachelor of Science in Computer Science in the
College of Computing

Georgia Institute of Technology

December 2017

HIGH PERFORMANCE ALGORITHMS FOR K-MER COUNTING AND GENOMIC READ ALIGNMENT

Approved by:



Dr. Srinivas Aluru, Advisor
School of Computational Science and Engineering
Georgia Institute of Technology

Dr. Richard Vuduc
School of Computational Science and Engineering
Georgia Institute of Technology

Date Approved: December 10, 2017

Eventually we'll be able to sequence the human genome and replicate how nature did intelligence in a carbon-based system.

Bill Gates

To my parents.

ACKNOWLEDGEMENTS

This project would never have become possible without the kind guidance of my research mentors, Prof. Srinivas Aluru, Prof. Vijay Vazirani, or Prof. Richard Vuduc. I would like to thank the professors for their bright ideas and enduring passion for inspiring discussions, as well as the huge amount of insightful advice they gave on this project.

I am also very grateful and honored to have worked at the Computational Biology Lab at the School of Computational Science and Engineering along with PhD student Rahul Nihalani, a knowledgable and reliable friend. Also, I would like to thank Sriram Chockalingam for implementing the parallel version of the many-block LSH algorithm.

TABLE OF CONTENTS

LIST OF TABLES

SUMMARY

Advancements in genomics are enabling a deeper understanding of how human body works and bringing us a new era of personalized healthcare. Genome sequencing and genome assembly are important procedures that facilitate genomic understanding whose quality affect nearly all downstream analysis. Modern high throughput sequencers could generate a few billion DNA subsequences with high accuracy in one experiment, proposing challenges to existing genome assembly pipelines that could only processes millions of reads per week. For the de Bruijn graph (DBG) based pipelines whose running time do not increase much with the number of input reads, efficient k-mer counting that is stable under the presence of repeats can serve as a useful heuristic for genome reconstruction. For Overlap-Layout-Concensus (OLC) pipelines, efficient sequence overlap finding is the first and most computationally intensive step in the process that must be improved to accommodate the huge amount of reads.

To address these problems, this work presents a new *de novo* k-mer counting method that utilizes read pairs double matching, and a new approach to constructing the sequence overlap graphs based on locality sensitive hashing (LSH). We provide theoretical performance estimation of the latter method, followed by experimental verifications on simulated read data (large dataset contains about 1.25 billion reads). Our approach is the first overlap finder that could construct billion-scale overlap graphs.

The experiment results for the *de novo* k-mer counting algorithm indicate it can provide a quite robust upper bound of k-mer occurrences. The experiment results for the overlap finding method suggest that our approach is at least 24 times faster (on the billion node size overlapping graph) and more flexible than one of the most influential overlap finders currently available, Minimap.

CHAPTER 1
INTRODUCTION

1.1 Background

Genomics, the study of genomes with a focus on the DNA sequences, has been an important subject in modern biological research since 1970s, when scientists were first able to reveal the DNA sequence of simple organisms [1]. However, it was not until 1990s, when the complete genome of *H. influenzae* was sequenced [2], that problems in genomics received much interest from computer science researchers. The increasing volume of genomic data led to an even larger amount of computations, which soon rendered many naïve algorithms (previously created and widely used by biologists) computationally infeasible [3]. Since then, the rapidly decreasing cost of genome sequencing and the increasing amount of available raw genomic data have been demanding high performance computing architectures and algorithms, with the hope to tackle previously daunting and over-ambitious tasks in genomics.

Advancements in genomics are changing the scientific horizon and promising an era of personalized medicine for elevated human health [4]. Genome assembly is one of the central tasks in genomics that researchers have been exploring. Its purpose is to reconstruct the original genome (essentially a long sequence of nucleobases) from a large set of sequenced reads (subsequences of nucleobases) taken from it [5]. Researchers are specifically interested in *de novo* assembly, which builds the original genome from scratch without referencing a known DNA sequence [6]. This kind of assembly is of great interest and being actively researched since it is a fundamental stage in genomic analysis whose accuracy affects the results of downstream analyses that utilizes the reconstructed genome [7].

1.2 Review of Previous Work

This section will first examine different sequencing techniques that generate the raw genomic reads to be assembled. This is important since the characteristics of different sequencing methods affect the design of assembly algorithms. The rest of this section will discuss two completely different classes of genome assembly techniques, namely the de Bruijn graph-based approach and the overlap-layout-concensus approach.

1.2.1 Genome Sequencing Techniques

The two generations of sequencing techniques available for genome assembly are introduced in this subsection. Then we shift our focus to the two most popular schemes for actually performing genome assembly on sequenced data.

First-generation Sequencing Techniques

Sanger's method, which is based on the selective incorporation of chain-terminating dideoxynucleotides by DNA polymerase during *in vitro* DNA replication [1], remained the most influential work among the first-generation sequencing for tens of years. However, it suffers from high costs, complex procedures, and low throughput [5]. Also it could only be used to sequence short strands of 100 to 1000 base pairs [4, 5]. Thus it has recently been replaced by the next-generation sequencing techniques.

Next-generation Sequencing (NGS) Techniques

A popular modern approach to sequencing a DNA molecule is the shotgun approach. The shotgun approach takes reads from random starting positions along the target molecule [8]. Whole-genome shotgun (WGS) sequencing refers to the process of taking samples from chromosomes that constitute a genome. The WGS techniques could deliver whole-genome sequencing results, which were not possible with first-generation sequencing.

Popular sequencing providers nowadays include Illumina, Applied Biosystems, Helicos, Pacific Biosciences (PacBio), and Oxford Nanopore Technologies (ONT) [9]. Research have been conducted regarding the read quality, read error distribution, and the sequencing bias of these platforms [4, 5].

Repetitive subsequences of DNA molecules have been shown to affect the quality of assembly [10]. Different NGS platforms provide different strategies to mitigate the effect of repetitive structures in assembly. For example, Illumina could provide nearly accurate reads (less than 1% error rate) in pairs [11, 12]. PacBio reads, while containing more errors (approximately 15% without error correction), could be very long (up to $56k$ base pairs in length) to cover the whole repeated region [6].

1.2.2 The de Bruijn Graph (DBG, or K-mer Graph) Scheme for Assembly

The de Bruijn Graph (DBG) approach is often used to assemble a large amount of short, mostly accurate reads [5], e.g., Illumina reads.

DBG is a well-known kind of graph in computer science literature, developed independently outside the area of genome assembly. A DBG represents a set of strings made up from a finite alphabet. It is a graph whose nodes represent all possible fixed-length substrings (whether or not they appear in the set of strings that the de Bruijn graph represent) made up from the finite alphabet, and whose edges represent all perfect suffix-prefix substring matchings that occur in the strings. It is also obvious that a path on the de Bruijn graph could represent a continuous string.

A k-mer is a string containing exactly k bases (A, T, G or C). A k-mer graph is a de Bruijn graph with a more limited set of nodes, only containing k-mers that appeared in the sequenced reads. Analogous to the de Bruijn graph base, the k-mer graph of the genome sequence must contain a path that correspond to the original sequence. Obviously, it takes only linear time to construct a k-mer graph (linear in terms of the length of target genome), as long as the memory could hold all the k-mer nodes and all reads. In order to address

the memory consumption problem, ABySS has been implemented as a shared-memory platform for DBG assembly [13].

The DBG approach to genome assembly can be reduced to an Eulerian path finding on k-mer graphs constructed from all reads [14]. An Eulerian path is a graph traversal that passes each edge in the graph exactly once. Finding Eulerian path is an ancient linear time algorithm in graph theory.

Other implementations of the DBG include [15, 16] with emphasis on different assembly quality aspects, such as repeat regions resolving, high-continuity, etc.

1.2.3 The Overlap-Layout-Concensus (OLC) Scheme for Assembly

The OLC scheme for assembly generally consists of four stages, all-vs-all raw read overlapping, raw read error correction, assembly of error-corrected reads, and read concensus polish [17]. We will review tools developed for each stage below.

Raw Read Overlap Finding

This stage is the most challenging among the four and involves the most computational efforts [6, 17]. The purpose of this is stage is to create an overlap graph of all available raw, not error-corrected reads. The overlap graph is a graph whose nodes are read identifiers, and there is an edge between two nodes if and only if the reads represented by these two nodes have a significant overlap.

Recently developed overlap finding tools include DALINGER [18], MHAP [6], and Minimap [17]. A common place for these tools is that they all compute sketches, or concise representations of the original reads and use hash tables to match the sketches, instead of matching the original reads. A comparison of these tools could be found in [17].

Raw Read Error Correction

After obtaining the all-vs-all overlapping graph, error correction could be completed by taking the concensus of overlapping reads for conflicting bases [6, 19].

Assembly of Error-corrected Reads and Concensus Polish

After we obtain the error corrected reads and corresponding overlapping graph, we could perform the assembly. The concensus stage is usually integrated into the assembly process, while some studies skip the concensus stage but relying on the error correction of raw reads [17]. Celera assembler [20] is still a popular choice for this step that addresses repeated regions and error correcting in concensus stage well. A much more recently developed Miniasm assembler [17] is much faster than old assemblers.

1.3 Challenges in Previous Work

The most significant and fundamental challenge remained in genome assembly is properly dealing with repeats in DNA sequences.

The OLC scheme for assembly managed to overcome much of this difficulty through longer reads or paired reads [6, 17, 21]. However it still remained a problem for the DBG scheme, where reads are broken into much smaller k-mers. This highly localized feature of DBG scheme makes it hard to resolve repeats (though easy to identify them [15]), since paths correspond to repeated regions will be merged together in DBG, resulting in difficulties with the Eulerian path based methods.

Problems with the OLC scheme centered around the huge computational cost, in terms of CPU hours and memory consumption, in all-vs-all read overlapping [6, 17].

Another big gap in the field is the theoretical basis upon which people evaluate the performance of different OLC methods, especially in terms of computational time estimation and guarantee on the missing rate of overlap finding algorithms.

1.4 Primary Contributions of This Work

This work filled in the research gaps in the following ways: First, this work presents a fast approximation algorithm for k-mer counting and partial assembly of repeated regions. The k-mer counting results could be used to upper-bound the number of times a path could be traversed in DBG assembly scheme. Secondly, this work brings in a probabilistic analysis framework to the field, upon which performance of overlap finding algorithms based on sketching could be analyzed. Finally, this work adapts a recently developed dimensionality reduction method (one-permutation hashing) from data sciences to all-vs-all overlap finding. The suggested method could be about $20 - 30$x faster than previous LSH-based methods and uses much less memory.

CHAPTER 2
SEQUENCE OVERLAP GRAPH CONSTRUCTION WITH PROBABILISTIC QUALITY ESTIMATION

In this chapter, we present a novel overlap graph construction algorithm with accompanied quality estimation. First, we introduce some basic concepts in this area. Then we build on the idea of [22] to introduce our algorithm while performing error analysis on the assumptions we make. Finally, we present our algorithm formally, and validate the effectiveness of our algorithm on 3 different datasets.

2.1 Preliminaries

2.1.1 Jaccard Similarity, Minhash, and LSH

Jaccard similarity is a measure to describe the degree of similarity of two sets. The Jaccard similarity of two sets A and B is defined as

$$\mathrm{Jac}(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{2.1}$$

The value of Jaccard similarity lies in range $[0, 1]$ and the value 1 indicates A and B are the same.

In the assembly problem, we regard a read as an unordered set of all its k-mers. A necessary condition for two reads to have a good suffix-prefix overlap is that their Jaccard similarity is large. Therefore we could see that two reads having high Jaccard similarity is a potential pair to be merged.

Now say that we want to pick out similar reads out of a pool containing n reads. It would take $O(n^2)$ time to compute the Jaccard similarity between all pairs of reads and we need a better way to do this.

Let us fix a set A for now. Among all other $n-1$ sets, those similar to A should constitute only a very small fraction (roughly a constant number). Thus it is natural to ask: could we compute "fingerprints" of all sets, such that similar sets have the same fingerprint?

We could do the fingerprinting in the following way [22]: we create a function h to get fingerprints for all sets. This function h has a candidate element pool coming from the union of all sets and it will try these candidate in a fixed order. To create a fingerprint for a set A, h will try, in its own inner ordering, each candidate and see if A contains it. And h will use the first candidate contained in A as A's fingerprint. We have the following conclusion:

$$P(h(A) = h(B)) = \mathrm{Jac}(A, B) \tag{2.2}$$

The possibility here is due to the various choices of h, assuming h could use any ordering of candidate elements uniformly randomly. Considering the picking process of h on A and B, it will not decide the fingerprint for either A or B until it goes to an element in $A \cap B$ or $A \Delta B$. Thus the probability for it to encounter an element in $A \cap B$ before it sees an element in $A \Delta B$ is $\frac{|A \cap B|}{|A \cap B| + |A \Delta B|}$, which is $\mathrm{Jac}(A, B)$.

If we compute a lot of h's, we will be able to estimate $\mathrm{Jac}(A, B)$. Still, we need a clever way to avoid $O(n^2)$ time to inspect every pair of sets.

Locality-sensitive hashing (LSH) is a linear time method that employs an idea of producing similar hash values for similar objects (which is somehow the opposite of typical hash).

Suppose for every set, we compute $N = B \times T$ different minhash h of it. Then we divide these N values into B blocks (so that in a block, every set has T h values). Now we focus on all sets in one block. Suppose for some pair of sets A and B, $r = \mathrm{Jac}(A, B)$. Since we compute T different minhash h for both A and B in this block, the probability for all T minhashes from A and B to be equal is r^T. On the contrary, the probability for there to be some differences is $(1 - r^T)$. Consequently, the probability for A and B to have entirely the same T hashes in some block is $P = 1 - (1 - r^T)^B$.

It is easy to observe from the above deductions that, if the Jaccard similarity of two sets r is big enough, these two sets is almost guaranteed to have the same T hashes in at least one block. So we could only examine the pairs of sets with same hashes in every block for their true similarity.

Let us now plug in our numbers to evaluate these probabilities. For two reads A, B both of length S, say we extract all k-mers from them. If they have an L-character overlap, $\mathrm{Jac}(A, B) = \frac{L-k+1}{2S-L-k+1}$. If they differ in two positions, at most $2k$ of the common k-mers will be corrupted. Thus $\mathrm{Jac}(A, B)$ could be as small as $\frac{L-3k+1}{2S-L-k+1}$. For $S = 100, L = 30, k = 10$, this value would diminish. But if we only allow one mismatch with two reads, this value could be $\frac{11}{161}$. If we select $S = 100, T = 2, B = 1000$, then $P = 0.9907$, meaning that under 99% of the times our method will pick out these pairs as candidate pairs. (For pairs of reads that have larger overlap, it is almost certain that they'll be pulled out using this method.) We expect the case where two mismatches happen to be very rare, therefore covered by looking at other reads.

2.1.2 Complexity and Quality Estimation

This section estimates the time and space complexity as well as the quality of the suggested Minhash approach.

A simplified analysis

Suppose we have n reads, each S base pairs long. And we would compute T hash functions within each of the B blocks. Assume that r_1 and r_2 are some read within the pool of reads and $\mathrm{Jac}(r_1, r_2) = r$. Let A denote the event that r_1 and r_2 is picked out using the technique. Now what we are interested in are two conditional probabilities $P(A \mid r)$ and $P(\bar{A} \mid r)$.

We have computed the probability for a pair not to be picked as follows in the last subsection:

$$P(\bar{A} \mid r) = (1 - r^T)^B \tag{2.3}$$

and thus

$$P(A \mid r) = 1 - P(\bar{A} \mid r) = 1 - (1 - r^T)^B \tag{2.4}$$

And in order to estimate the false positive and false negative probability, we need $P(A \mid r \leq r_0)$ and $P(\bar{A} \mid r \geq r_0)$, respectively.

The Jaccard similarity is actually discrete in our case, thus

$$P(A \mid r \leq r_0) = \frac{\sum_{r \leq r_0} P(A \mid r) P(r)}{\sum_{r \leq r_0} P(r)} \tag{2.5}$$

We would need $P(r)$ to continue the computation, namely the distribution of Jaccard similarity within all pairs. Let us ignore the effect of the reads close to the ends of the genome and the effect of high-variance coverage, and suppose that the numbers of pairs with different overlap length are roughly constant when the overlap length varies. (Note that we simply ignore the overlap due to chances as well as due to repeated regions for now and only focuses on overlaps due to the original positions.)

Suppose that overlap length of some pair of reads is denoted by l and their Jaccard similarity is $r = \frac{l}{2S-l}$. Since distribution on l is uniform, we have

$$\begin{aligned} P(A \mid l \leq l_0) &= \frac{\sum_{0 \leq l \leq l_0} (1 - (1 - (\frac{l}{2S-l})^T)^B) \frac{1}{S}}{l_0/S} \\ P(\bar{A} \mid l > l_0) &= \frac{\sum_{l_0 < l \leq S} (1 - (\frac{l}{2S-l})^T)^B \frac{1}{S}}{(S - l_0)/S} \end{aligned} \tag{2.6}$$

In order to calculate the false positive and false negative rates in the picked pairs, we need to calculate $P(A)$, which is the probability of picking a pair of reads given that the pair do have some overlap.

$$P(A \mid \text{pair having overlap}) = \sum_{1 \leq l \leq S} (1 - (1 - (\frac{l}{2S - l})^T)^B) \frac{1}{S} \tag{2.7}$$

Also, we need the percentage of pairs that do have overlap (let coverage be σ =

$\frac{nS}{\text{Genome Length}}$):

$$P(\text{is a pair having overlap}) = \frac{n\sigma}{\binom{n}{2}} = \frac{2\sigma}{n-1} \tag{2.8}$$

Thus $P(A) = P(A \mid \text{pair having overlap}) * P(\text{is a pair having overlap})$. And we have the false positive rate

$$\text{FPR} = P(l \leq l_0 \mid A) = \frac{\sum_{0 \leq l \leq l_0} (1 - (1 - (\frac{l}{2S-l})^T)^B) \frac{1}{S} n^2 \frac{2\sigma}{n-1}}{\sum_{1 \leq l \leq S} (1 - (1 - (\frac{l}{2S-l})^T)^B) \frac{1}{S} n^2 \frac{2\sigma}{n-1}} \tag{2.9}$$

And the false negative rate (the percentage of the missing pairs among the correct pairs) is just the same as Equation 2.6 or as follows:

$$\text{FNR} = P(\bar{A} \mid l > l_0) = \frac{\sum_{l_0 < l \leq S} (1 - (\frac{l}{2S-l})^T)^B \frac{1}{S}}{(S - l_0)/S} \tag{2.10}$$

Error analysis

In the above analysis, we completely ignore several things: the selection of hash functions and the bias it introduces, and the "overlapping" pairs which are not close to each other in terms of original positions (two reads could have a Jaccard similarity of more than 0 even if they are not adjacent). We now take these factors into consideration and perform a finer-grained analysis.

Still, suppose we have n reads (each L base pairs long) and they form $\binom{n}{2}$ pairs in total. We denote a suffix-prefix overlap pair by $< r_i, r_j >$, if a suffix of r_i is a prefix of r_j. We also call this an adjacent pair. And we use $\text{ov}(r_i, r_j)$ as the length of the common suffix/prefix between r_i and r_j.

For a fixed r_i, how many pairs are there whose first component is r_i? Since the reads are of same lengths, it is expected that there should be σ such pairs on average where $\sigma = \frac{nL}{\text{Genome Length}}$ is the average coverage. Also, for a fixed read r, there are expected to be σ/L r_j's that form $< r, r_j >$ pairs where $\text{ov}(r, r_j)$ equals a fixed value between 1 and L. Ignoring the effect of the end of the genome where there is no enough room to hold

complete pairs, there should be σn adjacent pairs in total. And since we have no reason to believe that longer overlaps are more common in adjacent pairs, we assume that these adjacent pairs distribute uniformly on different overlap lengths.

Let us now consider the process of breaking reads into k-mers and the effect of calculating the Jaccard coefficient based on common k-mers. First we would consider the probability for two random strings of length L to have a common substring of length Q. It should be $(L-Q+1)^2(\frac{1}{4})^Q$. If by chance, two irrelevant reads (not adjacent, also not associated with repeated region) have a common $(Q+p)$-mer, their Jaccard similarity is only increased by $p/(2L - (\text{overlap length}) - k + 1)$. Note that when $(Q+p)$ goes beyond 30 the probability becomes diminishingly small. Thus we would ignore this case and assume that all reasonably large Jaccard similarity is due to adjacency or repeated regions.

The following discussion is based on a classification of pairs according to their Jaccard similarity. We would call the adjacent pairs type I pairs, and the pairs concerning repeated regions type II pairs, and all other pairs fake pairs. In the strict sense, only the type I pairs with a long overlap (longer than a assigned threshold) should be regarded as true answer.

We now discuss the selection of hash functions and its effect on the minhash technique. Rabin-Karp fingerprinting hash with a polynomial family $h(s) = \sum_{i=0}^{m-1} s_i c^i \mod q$ on random strings of length m has a bounded collision probability less or equal to $1/q$ [23]. By [22], we see that the minhash technique ideally wants a random permutation from the set of all possible k-mers to a set of integers with same size. We take $q = 2^{64}-1$ and the set of all possible k-mers with $k \leq 31$ could fit into the space $\{0, \ldots, 2^{64}-1\}$. [22] has proved that under this circumstance, this approximation can achieve very high accuracy. Actually, if we have a set S of k-mers and apply the Rabin-Karp hash h to transform k-mers into numbers, we have

$$\mathbf{E}(|h(S)|) = (q+1)(1-(1-\frac{1}{q})^{|S|}) \tag{2.11}$$

Since $|S|$ is small and q is large, we assert that applying h will not affect out calculations with high probability.

By [22], for two reads A and B we have

$$\mathbf{P}(\text{minhash}_c(A) = \text{minhash}_c(B)) = \text{Jac}(A, B) \tag{2.12}$$

in which c denotes that we pick smallest c values within hashes of all k-mers in A or B. Picking larger c can help eliminate the false positives due to chance. (We will revisit this result in the next section and prove it in a different way.)

It is hard to distinguish type I pairs and type II pairs if they both have a long overlap.

2.2 Algorithm and Parallelization Strategy

We present an algorithm that detects potential overlapping pairs formally based on previous discussions in Algorithm 1.

Algorithm 1 Finding candidate pairs using many blocks minhash.

Input: Read set R, number of blocks B, number of hash functions within one block T, hash function matrix H that is a $B \times T$ matrix.
Output: Set of potential overlapping pairs S.

$S = \emptyset$
for $i = 1 \dots B$ **do**
  arr $= \emptyset$ {auxiliary array that acts as a reverse lookup table}
  for $r \in R$ **do**
    for $j = 1 \dots T$ **do**
      let P be the set of all k-mers in r
      compute $K_{ij}(r) = \min_{l \in P} H_{ij}(l)$ {compute T minhashes for each read within this block}
    end for
    key$[r] = (K_{i1}, K_{i2}, \dots, K_{iT})$
    arr.append(tuple(key$[r], r$))
  end for
  sort arr according to the first component in each tuple
  for each segment in the sorted arr in which all reads share the same key **do**
    add every pair of read in this segment to S {these pairs are the potential overlapping pairs identified by our algorithm since they are hashed to the same value by T hash functions}
  end for
end for

We use the following parallelization method to do the computation, supposing we run the algorithm for n reads on a p processor computer without shared memory:

1. Load n/p reads into each processor's memory.

2. Each processor executes line 4-10 in Algorithm 1 independently.

3. Use parallel sorting to execute line 12 in Algorithm 1.

4. Now each processor will contain a sequence of reads with consecutive key values. If, when executing line 13 we found that a segment is separately stored in 2 processors, we move the entire bucket to the lower-ranked processor.

5. Repeat the above steps for B times, which corresponds to the outmost loop in Algorithm 1.

6. Refine the results by filtering out non-overlapping read pairs.

The time complexity is $O(BTnl/p)+O(B\times\text{ParallelSort}(nT,p))$, where ParallelSort$(m,p)$ is the time needed to sort m objects on p processors.

We implemented the proposed algorithm in C++ using MPI for the collective communication operations. We used KmerInd library to load the file data in blocks and to generate all k-mers. We used a version of distributed sample sort to perform the parallel sorting. For all the $T\times B$ hash functions, we choose MurMurHash with different seed values, which are big primes distributed in range. MurMurHash functions are known to behave min-wise independently and this property is useful in deriving probabilistic bounds discussed in the next section. We use the implementation of SMHasher (https://github.com/aappleby/smhasher) library for computing MurMurHash values for k-mers generated from the reads.

2.3 Experimental Results and Discussion

We ran our experiments on an Intel Xeon Infiniband cluster. Each node has two 2.0 GHz 8-core Intel E5-2650 processors and 128GB of main memory. Experiments were conducted on up to 64 nodes, totaling 1,024 cores. We evaluated our algorithm on three different datasets, shown in Table . Each dataset is a set of simulated Illumina reads derived from a

known genome using SimSeq, a read simulator designed to simulate Illumina short reads while taking into account the sequencer specific error models.The reason for using simulated reads instead of using a true Illumina dataset is to be able to know true suffix-prefix overlaps without running an alignment algorithm for every pair of reads, which would not be feasible computationally.

We use D1 and D2 to demonstrate the quality and scalability of our algorithm. Using D3, we demonstrate the ability of our algorithm to handle big genomic datasets. All the reads are of length 100.

	D1	D2	D3
Organism	*E. coli*	*H. sapiens*	*H. sapiens*
Source	Genome	chr1	Genome
#reads	1.75×10^6	87.5×10^6	1.25×10^9
Coverage	37.7X	35.4X	38.64X

Table 2.1: Datasets used in validating the many-block LSH algorithm.

To the best of our knowledge, no current software specifically targeting overlap graphs can estimate the graph for datasets with billions of short reads. However, in addition to standalone performance evaluation of our method, we also compared it against Minimap [17], a method that uses the MinHash technique but designed for the different problem of mapping and comparing long erroneous reads produced by PacBio and Oxford Nanopore Technologies sequencers. A recent survey [21] showed that among the currently available methods for finding the pairs of sequences with sufficient suffix-prefix overlap, Minimap performs better compared to other methods.

2.3.1 Quality assessment

Let F_e be the estimated FNR, F_o denote the observed FNR, and Ω denote the the ratio FP/TP.

We observe that F_e and F_o are reasonably close in most of the cases for D1, indicating that assumptions made while estimating the errors are reasonable. The estimates are partic-

Table 2.2: Experimental results on dataset D1, using many-block LSH algorithm.

Measure	K,T,B	Overlap Threshold (l_{min})		
		60	50	40
F_e	15,3,334	0.0018	0.0401	0.1511
F_o		0.0171	0.0392	0.0824
Ω		0.7968	0.4668	0.2779
F_e	13,3,334	0.0003	0.0173	0.1011
F_o		0.0092	0.0248	0.0609
Ω		0.8618	0.5097	0.3043
F_e	11,3,334	0.0001	0.0006	0.0606
F_o		0.0040	0.0139	0.0417
Ω		1.1094	0.7002	0.4586
F_o and Ω with Minimap				
F_o	11	0.2486	0.1882	0.1492
Ω		0.1003	0.1061	0.1120
F_o	9	0.2295	0.1737	0.1377
Ω		0.1047	0.1129	0.1288

Table 2.3: Experimental results on dataset D2, using many-block LSH algorithm.

Measure	K,T,B	Overlap Threshold (l_{min})		
		60	50	40
F_e	19,3,200	0.0558	0.1840	0.3146
F_o		0.1482	0.1914	0.2549
Ω		3.6302	2.8925	2.5146
F_e	15,3,300	0.0028	0.0471	0.1609
F_o		0.1049	0.1309	0.1769
Ω		7.1090	5.6551	4.8542
F_e	17,2,300	0.0001	0.0014	0.0410
F_o		0.0909	0.1001	0.1185
Ω		10.2986	8.1076	6.7351
F_o and Ω with Minimap				
F_o	15	0.4566	0.3888	0.3372
Ω		0.1091	0.1289	0.1591
F_o	13	0.4313	0.3646	0.3141
Ω		0.2340	0.3258	0.5078

ularly accurate for $(k, l_{\min}) = (15, 50), (13, 50), (11, 40)$ indicating a need for k, $l_{\min}$ to be proportional to each other for better estimates. For reasonable estimates, the value k needs to be balanced between the two extremes. A very small value of k could lead to too many spurious matches, whereas a large k can cause a single error in the overlap region to miss any common representatives between otherwise similar reads. For dataset D2, we typically underestimate the FNR by at most 0.1. In all the cases, the extra computing cost Ω due to false positives is a small constant, implying that our filter does not add enormous amount of extra overhead.

Minimap uses the minimum hash values of minimizers to compute the candidate overlapping pairs of sequences, a lower k value in general tends to favor lower observed FNR values. Therefore for Minimap experiments, we typically select the k-mer size lower than that used for evaluation of our method. Results of these runs are listed in Table 2.2 and Table 2.3. For both the datasets, our proposed method shows much lower FNR values in all the cases compared to Minimap. While Minimap shows lower Ω values compared to the proposed method, it does so at the cost of missing a significant percentage of the true suffix-prefix overlapping pairs of

2.3.2 Scalability of our algorithm

Table 2.4: Scalability results on dataset D1 and D2 for the many-block LSH algorithm.

	#cores	Run time (s)	Relative Speedup	Max memory per core (MB)
Dataset D1	16	648.86	1.00X	183.45
	32	328.56	1.97X	111.29
	64	169.49	3.82X	69.56
	128	121.31	5.34X	46.97
Dataset D2	64	2874.36	1.00X	2950.77
	128	1810.17	1.59X	1525.93
	256	908.33	3.16X	782.89
	512	528.61	5.43X	433.62

Results show that the run-time and the maximum per core memory scale up to 128 cores for D1 and up to 512 cores for D2, in line with their relative sizes. For any parallel algorithm, if the data size is fixed and the number of processors is continually increased, diminishing returns set in at some juncture. In our algorithm, this limit transpires due to two factors. First, the pair generation for a bucket takes time proportional to the square of the size of the bucket, and therefore, there is an imbalance in the amount of work done by the processors when the partitioning is too fine-grained. Second, our algorithm relies on parallel sorting, and communication costs dominate when the per processor data size becomes too small.

To test the applicability of our parallel algorithm for large datasets, we used dataset D3 containing 1.25 billion reads. We were able to process D3 in 58 minutes using 1024 cores. This demonstrates that our implementation is able to process big genomic datasets in reasonable time. For running Minimap, we used a machine with four 2.1 GHz 18-core Intel Xeon E7-8870 processors and 1TB of main memory. We used this large shared memory machine for Minimap runs because Minimap cannot take advantage of distributed memory but is only capable of utilizing cores available in a single machine via shared-memory threads. Though runtimes on machines with different capabilities are not directly comparable, we provide the runtimes for Minimap for the sake of completeness. Using 32 threads, Minimap took 0.9 and 37 minutes for datasets D1 and D2 respectively. For D3, even after consuming 24 hours of its allocated job time, Minimap failed to complete its run with 32 threads.

2.4 An Improvement Using One-permutation Hashing

In previous sections, we had obtained a feasible method that reduces the time needed to find similar reads from linear to sub-linear. However, the pre-processing time needed is large because we have to compute $O(|L|BT)$ hashes for each read. As demonstrated by [24], we don't actually need much of these computations. In order to adapt the results in [24] to the

assembly scenario, we reprodeduce some of the formulas in their paper as below.

Lemma 2.4.1 (Success probability of minhash)**.** *If set A and B satisfy $\mathrm{Jac}(A, B) = R$, the original minhash schema will successfully pick this pair up with a probability R among all choices of ideal hash functions π. Or*

$$\mathbf{Pr}_{\pi}(\textit{minhash}_c(A) = \textit{minhash}_c(B)) = \mathrm{Jac}(A, B) \tag{2.13}$$

Proof. Note that an idea hash function serves as a permutation π. Let the set of objects that A or B may contain be $D = \{1, 2, \ldots, n\}$. Then we could write A or B as a binary string, with i-th bit indicating whether the set contains object i.

Recall that the minhash scheme will apply π to both binary strings, and decide whether to pick up this pair based on whether the minimum non-zero bits in $\pi(A)$ and $\pi(B)$ match.

If we set $f_1 = |A|, f_2 = |B|, a = |A \cap B|$, then the probability for the minhash of A and B to agree is given by

$$\sum_{1 \le i \le n} \frac{\binom{a}{1}\binom{n-i}{f_1+f_2-a-1}(f_1 + f_2 - a - 1)!(n - f_1 - f_2 + a)!}{n!} \tag{2.14}$$

$$= \frac{a\frac{n!}{(f_1+f_2-a)!(n-f_1-f_2+a)!}(f_1 + f_2 - a - 1)!(n - f_1 - f_2 + a)!}{n!} \tag{2.15}$$

$$= \frac{a}{f_1 + f_2 - a} = R = \mathrm{Jac}(A, B) \tag{2.16}$$

Say the matched minhash is at the i-th digit for both A and B. We first choose one from a common elements and let it map to the i-th position in our permutation π. Then we found $(f_1 + f_2 - a - 1)$ positions for other bits that represent objects in $A \cup B$, among the $(n - i)$ available spots after i-th bit. We of course also permute these bits. Then the only task left is to permute the 0s that represent objects neither in A nor in B (where there are $(n - f_1 - f_2 + a)!$ ways). We sum up the probability for all i to get the desired equation.

□

2.4.1 A new one-permutation scheme

Suppose that we want to estimate the similarity of read A and B. We now adapt the following procedures:

1. Uniformly randomly select a permutation π that permutes k-mers in a set. Remember that it actually permutes the bits in a 0-1 assignment string.

2. Apply the same π to both A and B to obtain $\pi(A)$ and $\pi(B)$, where π : read $\mapsto \{0,1\}^D$ is a permutation of the k-mer spectrum of reads.

3. Divide the 0-1 strings $\pi(A)$ and $\pi(B)$ into k equally long blocks (or substrings, more precisely). Compute signatures $s(A)$ and $s(B)$ to be the position of smallest non-zero 1 bit in each block.

4. Go back to step 1 and randomly choose another permutation π'. Do this for l times.

Next, we reproduce some results from [24] but with a slightly different approach. The point in doing this is that we want to point out the assumptions made by [24] and discuss their applicability in the assembly use case.

Theorem 2.4.2 (Distribution of the numbers of mutually empty bins and matched bins)**.** *Suppose that N_{empty} denotes the number of blocks in which both $\pi(A)$ and $\pi(B)$ have no 1 bit, k is the number of blocks, D is total dimension, $f = |A \cap B|$. Then*

$$\mathbf{Pr}(N_{\text{empty}} = j) \tag{2.17}$$

$$= \frac{1}{D!}\binom{k}{j} A_{D-f}^{Dj/k} \left[(D - Dj/k)! + \sum_{s=1}^{k-j} (-1)^s \binom{k-j}{s} A_{(k-j-s)D/k}^{f} \right] (D - Dj/k - f)! \tag{2.18}$$

$$= \sum_{s=0}^{k-j} (-1)^s \frac{k!}{j!s!(k-j-s)!} \prod_{p=0}^{f-1} \frac{D(1-(j+s)/k) - p}{D-p} \tag{2.19}$$

Thus expected percentage of empty bins is

$$\mathbf{E}(N_{\text{empty}})/k \tag{2.20}$$

$$= \sum_{j=0}^{k-1} \frac{j\mathbf{Pr}(N_{\text{empty}} = j)}{k} \tag{2.21}$$

$$= \prod_{j=0}^{f-1} \frac{D(1-1/k)}{D-j} \tag{2.22}$$

$$\approx (1-\frac{1}{k})^f \tag{2.23}$$

Also, the expected percentage of matched blocks is given by

$$\mathbf{E}(N_{\text{match}})/k \tag{2.24}$$

$$= R\left[1 - \frac{\mathbf{E}(N_{\text{empty}})}{k}\right] \tag{2.25}$$

$$\approx R \tag{2.26}$$

Proof. First we show the combinatorial meaning of each term appeared in Equation 2.18. $\binom{k}{j}$ is the number of ways we choose and permute j empty blocks among k total blocks. $A_{D-f}^{Dj/k}$ is the number of ways we choose $\frac{D}{k}j$ elements to put into these j empty blocks. $(D - \frac{D}{k}j - f)!$ to the right of the square bracket represents ways to permute all elements except those put into the j empty blocks and those belongs to $A \cup B$.

Inside the square bracket, we use Inclusion-Exclusion principle to calculate the number of ways ensuring that every non-empty block (total number is $k-j$) has at least one element from $A \cup B$.

These yield Equation 2.19 after some simplifications.

Next, we look at the deduction of Equation 2.22. Actually the probability for any 1 out of k blocks to be empty is exactly $\frac{\binom{D(1-1/k)}{f}}{\binom{D}{f}}$, which is number of ways to put f items in $D(1-1/k)$ slots divided by the number of ways to put f items in D slots. Thus

$$E(N_{\text{empty}}) = k\frac{\binom{D(1-1/k)}{f}}{\binom{D}{f}} \tag{2.27}$$

$$= k \cdot g(1) \tag{2.28}$$

in which $g(x) = \prod_{t=0}^{f-1} \frac{D(1-x/k)-t}{D-t}$.

Another way to do this is by manipulating Equation 2.19.

$$= \sum_{j=0}^{k-1} \frac{j\mathbf{Pr}(N_{\text{empty}} = j)}{k} \tag{2.29}$$

$$= \sum_{j=0}^{k-1} \sum_{s=0}^{k-j} \frac{j}{k}(-1)^s \frac{k!}{j!s!(k-j-s)!} g(j+s) \tag{2.30}$$

$$= \sum_{j+s=l\leq k} \sum_{s\leq l} g(l) \frac{k!}{j!(k-j)!} \frac{(k-j)!}{s!(k-j-s)!} \frac{l-s}{k}(-1)^s \tag{2.31}$$

$$= \sum_{l\leq k} g(l) \frac{(k-1)!}{(k-l)!(l-1)!} \sum_{s\leq l-1} \binom{l-1}{s}(-1)^s \tag{2.32}$$

$$= \sum_{l\leq k} \binom{k-1}{l-1}(-1)^{l-1} g(l) \tag{2.33}$$

$$= g(1) \tag{2.34}$$

The last step above is due to mathematical induction on k.

To estimate the number of the expected empty blocks, we notice that

$$g(l) = \prod_{t=0}^{f-1} \frac{D(1-l/k)-t}{D-t} \tag{2.35}$$

$$= \prod_{t=0}^{f-1} \frac{D-t-Dl/k}{D-t} \tag{2.36}$$

$$= \prod_{t=0}^{f-1} \left(1 - \frac{l}{k(1-\frac{t}{D})}\right) \tag{2.37}$$

$$\approx (1-\frac{l}{k})^f \tag{2.38}$$

From this, it is easy to derive the expected percentage of matched blocks. □

Theorem 2.4.2 has shown that the percentage of matched blocks is approximately the same as Jaccard similarity of two sets.

Theorem 2.4.3 (Variance of the expected matched blocks, from [24])**.**

$$var(N_{match}) < kR(1-R) \tag{2.39}$$

Theorem 2.4.3 shows that the variance of N_{match} is not very large when the two sets are very similar, and it's hard for sets that have high Jaccard similarity to have no matched blocks.

Theorem 2.4.4 (An Estimator of Jaccard similarity R, from [24])**.** *The number of matched blocks and the number of mutually empty blocks could be used to estimate the Jaccard similarity R of two reads:*

$$\hat{R} = \frac{N_{match}}{k - N_{empty}} \tag{2.40}$$

$$E(\hat{R}) = R \tag{2.41}$$

$$var(\hat{R}) \approx \frac{R(1-R)}{k} \leq \frac{1}{4k} \tag{2.42}$$

where k is the number of blocks we divide the permutations into.

Theorem 2.4.4 shows that we could estimate the Jaccard similarity of two sets quickly by looking at their matched blocks and mutually empty blocks. Also, the confidence of our estimation could be enhanced by increasing the number of blocks k we use. Also notice that by Theorem 2.4.3, the denominator in $\hat{R}$ can hardly be empty if R is sufficiently large.

Theorem 2.4.4 could also provide overlap length hints to the assemblers [20, 17].

Theorem 2.4.5 (False negative rate estimation)**.** *We have the following conditional probabilities:*

$$\mathbf{Pr}(\textit{minhash match in one particular block}|R) = \frac{R}{k} \tag{2.43}$$

$$\mathbf{Pr}(\textit{not match in any block}|R) = (1 - \frac{R}{k})^k \tag{2.44}$$

$$\mathbf{Pr}(\textit{not match in any block for M permutations}|R) = (1 - \frac{R}{k})^{kM} \tag{2.45}$$

given R is the Jaccard similarity of two sets.

Proof. We have shown in Section 1 that the probability for the minhash of two sets to match is R. This R could lie in any of the k blocks, and these k blocks are symmetric with respect to all permutations. □

The use of Theorem 2.4.5 is to establish a bound on missing rate. This result is not from [24].

We now consider implementing the permutations as hash functions. Suppose the range of the hash function we choose contains 2^c numbers and we divide the range into k blocks. Thus there are $m = 2^c/k$ numbers within each block. In contrary, a permutation should contain 4^G numbers, where k-mer size is G and should contain $4^G/m = k2^{2G-c}$ blocks. But we could see that we may use hash functions with a larger range, say $c = 64$ to make this effect nearly eligible. And if the hash function scatters the k-mers well into its range (e.g., smhasher), this will not lead to much performance degradation.

2.5 An Overlap Finder Based on One Permutation Hashing

In this section we will present an algorithm for finding overlapping read pairs based on theoretical basis laid out in the previous section.

2.5.1 Algorithm

Algorithm 2 is a sequential version of the overlapping pair finding algorithm using one-permutation hashing technique.

Algorithm 2 Finding candidate pairs using one-permutation hashing.

Input: Read set R, Hash functions set H, number of blocks k.
Output: Candidate overlapping pairs set S, estimated Jaccard similarity of all candidate paris.

for hash function $h \in H$ **do**
  set up reverse look-up table array $A[1 \dots k]$ (each $A[i]$ is a dictionary that maps a hash value back to the list of reads r it come from)
  for $r \in R$ **do**
    set up minhash array $m[1 \dots k]$ to be the largest possible value the hash functions can take in each range block
    for kmer $\in$ getKmer(r) **do**
      $v = h$(kmer)
      let k' denotes the block that hash value v lies in
      if $v < m[k']$ **then**
        $m[k'] = v$ {$m[k']$ stores smallest hash in block k'}
      end if
    end for
    for $i = 1 \dots k$ **do**
      if $\exists$ kmer $\in r$ such that h(kmer) lies in block i **then**
        $A[i][m[i]]$.append(r) {for read r, the smallest hash in block i is $m[i]$}
      end if
    end for
  end for
  for $i = 1 \dots k$ **do**
    for $v \in$ keyset($A[i]$) **do**
      add every pair in list $A[i][v]$ to candidate pairs set S
      update the Jaccard similarity estimation of every pair using Theorem 2.4.4: $\hat{R} = N_{\text{match}}/(k - N_{\text{empty}})$ {the variance should reduce as more hash functions pick the same pair out}
    end for
  end for
end for

2.5.2 Theoretical Performance Analysis

The following analysis assumes that we are dividing the range of the hash function into k blocks and we take the union of the pairs found by calculating M hash functions.

False negative (missing) rate

Let A denote the event that a pair is picked by our algorithm. According to Theorem 2.4.5, the probability for the algorithm to miss a pair with Jaccard similarity R is:

$$P(\bar{A}|R) = (1 - \frac{R}{k})^{kM} \tag{2.46}$$

Thus the false negative rate for all pairs with Jaccard similarity greater than or equal to R is

$$P(\text{miss}|R \geq R_0) = \frac{\sum_{R_0 \leq R \leq 1} P(\bar{A}|R)P(R)}{\sum_{R_0 \leq R \leq 1} P(R)} \tag{2.47}$$

where $P(R)$ is the distribution of pairs against Jaccard similarity.

False positive rate

In estimating the false positive rate, we ignore the effect of using hash functions rather than permutations. Also, we ignore the cases where the hash values collide coincidentally since these probabilities are small. The false positive rate is given by:

$$r_{FP} = P(R < R_0|A) = \frac{P(R < R_0, A)}{P(A)} \tag{2.48}$$

$$= \frac{P(A|R < R_0)P(R < R_0)}{P(A)} \tag{2.49}$$

$$= \frac{\sum_{R < R_0} P(A|R)P(R)}{\sum_R P(A|R)P(R)} \tag{2.50}$$

Time and space complexity

Suppose read coverage is σ, the read set is R, the read length is L, and we compute M hash functions for each read in total, whose ranges are divided into k blocks. Also assumes that our algorithm has very little false negative rate and a false positive rate of r_{FP}.

Thus the total running time is given by

$$|R|L + \frac{\sigma|R|}{1 - r_{FP}} \tag{2.51}$$

We could see that the computation of processing each block are completely independent of other blocks. Thus the computation is easily parallelizable.

The total space consumption is

$$|R|L| + 2k|R| + \frac{\sigma|R|}{1 - r_{FP}} \tag{2.52}$$

assuming we are using a shared-memory environment.

2.5.3 Results and Discussions

In this section, we compare the new one-permutation based pair filtering approach to the old many-block minhash technique. The 10000 reads are simulated from an excerpt from *E.coli*'s genome. In all experiment settings, the trusted overlap length is set to 60. The many-block scheme uses parameters $B = 300, T = 3, K = 7$. The one-permutation scheme uses $k = 10$ blocks and maps each k-mer into range $[0, 10^8]$.

Table 2.5: Experiment results on overlap graph quality measurements, from the one-permutation LSH method and the many-block LSH method.

Measure	Many-block LSH	One-permutation, $\|M\|$ hashes			
		$\|M\| = 2$	$\|M\| = 3$	$\|M\| = 5$	$\|M\| = 8$
r_{FN}	0.03%	0.08%	0.08%	0.04%	0.04%
r_{FP}	37.18%	45.79%	47.64%	61.68%	63.03%
Running time (s)	8287.5	135.3	179.6	314.2	490.6

From Table 2.5, we observe that the suggested one-permutation technique outperforms the previous many-block method significantly in terms of running time. This is also the main contribution of the one-permutation technique.

Also, we should note that we did not lose much on the false negative rates. This actually

matches our expectation and theoretical deductions. In addition, there are some false negatives that are not necessarily false negatives since the simulator created erroneous reads. The 0.03% false negative rate given by 300 block method demonstrates this well since the theoretical FN rate is several magnitudes smaller than this.

The false positive rates shown here could be improved by implementing hash functions with a larger range. We discussed the use of hash functions and its effect on our method's performance in Section 2, and we assumed the range to be 2^{64}. But in this initial verification of ideas I used 2^{31}. The many-block scheme suffers from this less since it gets to use the full range of the hash functions, while the one-permutation scheme could only use $1/k$ of the range. Thus we expect if larger hash function ranges are adapted, the FP rate of one-permutation scheme should be comparable or even better than that of the many-block approach, making the running time overhead due to FP differ little from the many-block approach.

2.5.4 Future Work

Though we have shown in the previous sections that the sequential version of the suggested overlap finding algorithm is much faster than the sequential version of existing methods, more experiments are to be conducted using a parallelized implementation of the suggested algorithm.

CHAPTER 3
AN APPROXIMATION ALGORITHM FOR K-MER COUNTING

3.1 Problem Formulation

Given a set of reads R taken from genome s, and a k-mer (substring) p of length k, we want to count how many times p appears in s (all lengths are in terms of base pairs).

The problem is inherently hard. Since any read could come from multiple locations in the original genome if it involves repeated regions in the genome, we must find a way to cluster the reads coming from nearby locations together.

To address the location ambiguity problem, we consider using high-coverage Illumina paired reads, where each read pair consists of two reads of length l separated by distance d. We assume that d is greater than the lengths of the majority of the repeated regions, since d could be more than 2000 base pairs long [12].

3.2 Algorithm and Analysis

The idea proposed by Prof. Vazirani is inspired by a classic approximation algorithm to find the minimum length superstring. It is based on the fact that the read pair distances in the Illumina paired reads have a small variance (less than 10%). Consider read pairs $r_1 = (r_{11}, r_{12})$ and $r_2 = (r_{21}, r_{22})$. If r_{11} and r_{21} have a good overlap, and r_1 and r_2 do come from nearby locations, then r_{12} and r_{22} should also come from nearby locations in the genome, by the small variance in read pair distance assumption. Even if r_{12} and r_{22} do not directly overlap with each other, we could always find a series of "intermediate" reads that could fill in the gap, provided we are using high-coverage reads.

An important concept used in the algorithm is the suffix-prefix overlap. A suffix-prefix overlap between any two strings s_1 and s_2 means that there exists s' that is a suffix of s_1

and also a prefix of s_2. A maximum suffix-prefix overlap refers to the s' with maximum length.

The naïve algorithm is given as Algorithm 3.

Algorithm 3 Counting k-mers by greedily merging read pairs.

Input: Read pair set R, K-mer p.
Output: Number of occurrences of p in the original genome.

1: $R' = \{r : (r \in R) \cap (r \text{ contains } p)\}$ {relevant read set}
2: **while** true **do**
3: overlap score set $S = \emptyset$
4: **for** $r \in R'$ **do**
5: **for** $r' \in R'$ and $r' \neq r$ **do**
6: calculate overlap score ov(r, r')
7: add ov(r, r') to S
8: **end for**
9: **end for**
10: sort S in decreasing order
11: **if** $\max S <$threshold **then**
12: break
13: **end if**
14: **for** $s \in S$ **do**
15: **if** $s <$threshold **then**
16: break
17: **end if**
18: suppose that $s = \text{ov}(r, r')$, remove r, r' from R'
19: add $r_{\text{new}} = \text{merge}(r, r')$ to R'
20: **end for**
21: **end while**
22: occurrences of p in the original genome is round($\frac{|R'|}{2}$)

In line 6, we calculate the overlap score between read pair $r = (r_1, r_2)$ and $r' = (r'_1, r'_2)$. Suppose that o_1 is the maximum suffix-prefix overlap between r_1 and r'_1, and o_2 is the maximum suffix-prefix overlap between r_2 and r'_2. The overlap score is a symmetric function of o_1 and o_2 that takes reversed complementary matchings into account. We have used the maximum product of overlap lengths in our exploratory implementation.

In line 19, we create $r_{\text{new}} = \text{merge}(r, r')$. r_{new} is also a read pair whose first read is the result of merging the first components of r and r', and whose second read coming from merging the second components of r and r'.

The time complexity of Algorithm 3 is $O(|R| + l^2n^2)$, where n is the number of reads

that contain the target k-mer p. The all-vs-all overlapping score calculation can be optimized using suffix trees or the one-permutation hashing technique, resulting in an overall complexity of $O(|R| + ln)$, if we only update the scores affected by line 18 and 19 in each iteration. But given that n is often very small compared to the read set size $|R|$, the unoptimized version is computationally feasible.

3.3 Results and Discussions

Table 3.1: Estimating k-mer occurrences with Algorithm 3 in a substring of the *E. coli* genome. Only high frequency k-mers are listed.

k-mer spectrum	reported occrrences	true occrrences
k-mer 1	7	6
k-mer 2	6	5
k-mer 3	7	6
k-mer 4	7	6
k-mer 5	8	7
k-mer 6	5	5
k-mer 7	5	4
k-mer 8	16	12
k-mer 9	13	10
k-mer 10	6	5

We performed a preliminary empirical study of the effectiveness of the proposed algorithm. We used a length 200000 substring of the *E. coli* genome and perform queries on about 100 k-mers. We simulate our read set with a coverage of 40 X.

Since *E. coli* genome is known to be not repeat-rich, we manually implant several repeat regions of different lengths in the genome and query k-mers within the implanted regions. Some repeat region implantations are adjacent, while some are scattered in the genome. The repeat region length varies from smaller than read lengths l (about 10 base pairs) to smaller than read distance d (about 200 base pairs).

From the experiment results, we found that our algorithm will almost always yield correct answers while querying low-frequency k-mers (that only appear once or twice in the genome). Also, our algorithm works well on the small repeat region lengths. Thus

we only record the errors our algorithm make when queried with high-frequency k-mers in Table 3.1.

The results suggest that our algorithm provide a quite stable upper bound estimation of k-mer occurrences. The algorithm would often return answers slightly larger than the true occurrences since it will not be able to correctly merge all read pairs due to read errors (false negative merging). We expect the upper bound will be more accurate if we error-correct all reads before running our algorithm. The only case it would return with an answer lower than the true occurrences is when it makes wrong merges (false positive merging) in the relevant read set R'. This could be a result of highly repetitive regions, or very long repeat regions (much larger than read distance d). In either case we cannot improve much, given that the information on repeat regions is already lost when sequencing the original genome. But we expect that sequencing technologies that offer longer reads and/or longer read pair distances [6, 9] will help address this problem.

3.4 Future Work

We hope to implement a parallel version of Algorithm 3 in the future so that we could run experiments on larger genomes and larger read sets. Also, we would like to add a read error correcting stage when constructing the relevant read set R' in Algorithm 3.

Appendices

APPENDIX A

SOURCE CODE

A.1 K-mer Counting Algorithm

Listing A.1: Building the genome and simulating the reads

```
#!/usr/bin/python3.4
from __future__ import print_function
# import random, string, numpy as np, os, sys, matplotlib.pyplot as
      plt;
from myutil import *;
from read_sequencing import *;

if False: '''

TODO LIST:
1. Add verbose information as the probram proceeds through stages
2. When reading genome from file, don't take num of lines as the
      first param
'''

if (len(sys.argv) != 3):
   print("Correct usage: ", sys.argv[0], " <config_file> <out_dir>");
   exit();

print("Reading params from config file.....");

# open config file
fin = open(sys.argv[1], 'r');
# read the config
input_array = []
while (True):
   line = fin.readline();
   if (line == ''):
      break;
   if (line[0] == "#"):
      continue;
   input_array.append(line);

i = 0;

# Read the genome from file or generate
read_genome_from_file = input_array[i];
read_genome_from_file = read_genome_from_file[0:-1];
i = i + 1;

# Input file for the cases where the genome and reads have to be read
      from file
# instead of being randomly generated
f_inputread = '';

f_genome = '';
# if the genome has to be read from a file, open that file
if (not read_genome_from_file == "none"):
   f_genome = open(read_genome_from_file, 'r');
# add sanity check

# open output directory and check it's existance
out_dir = sys.argv[2];
if (not os.path.isdir(os.path.join(os.getcwd(), out_dir))):
   print("Error: no directory named ", out_dir);
   exit();

cp_command = "cp -f " + sys.argv[1] + " " + sys.argv[2];
if (not cp_command[-1] == "/"):
   cp_command += "/";
cp_command += "copy_" + sys.argv[1];
os.system(cp_command);

# open all needed files in output directory
os.chdir(out_dir);
# f_walkin=open("walk_input.txt", 'w');
f_log = open("log_DbGraphBuild.txt", 'w');
f_err = open("error_DbGraphBuild.txt", 'w');
# f_reread=open("reread.txt", 'w');
f_genome_out = open("DB_build_genome.txt", 'w');

# copy the config file in the folder
pass_file_objects(fin, f_log, f_err);

# read other parameters from config file
# Genome length
L = int(input_array[i]);
i = i + 1;

# Number of reads
N = int(input_array[i]);
i = i + 1;

# Read length
l = int(input_array[i]);
i = i + 1;

# Mate pair distance, i.e. the number of hops required to get from
      one to the other.
d = int(input_array[i]);
i = i + 1;

# read method
read_method_param = int(input_array[i]);
i = i + 1;

per_line_param = int(input_array[i]);
i = i + 1;

# some name, checking if coverage plot is needed
q3r = int(input_array[i]);
i = i + 1;
covg_hist_plot_needed = True;
if (q3r == 0):
   covg_hist_plot_needed = False;

lmer_merge_check = input_array[i];
i = i + 1;
# Config file read

# initialize vars
G = '';
readset = [];
read_metadata_set = [];
```

```
coverage_hist = [];
mutation_array = [];

# Generate/read a random genome
if (not read_genome_from_file == "none"):
   print("Reading genome from file...");
   num_lines = f_genome.readline();
   num_lines = int(num_lines);
   for i in range(num_lines):
      line = f_genome.readline();
      if (line == ''):
         break;
      line = line[0:-1];
      G += line;
   L = len(G);
else:
   print("Generating genome...");
   G = ''.join([random.choice(alphabet) for p in range(L)])

# Create long repeated regions
print("Creating repeated regions in G...")
sample_pos = 400
rep_len = 70
rep_pos_list = [800, 870, 940]
G = create_repeats(G, sample_pos, rep_len, rep_pos_list)

sample_pos = 470
rep_len = 30
rep_pos_list = [1500, 3000, 4500, 6000, 15000, 18000]
G = create_repeats(G, sample_pos, rep_len, rep_pos_list)


# print("Testing repeats:")
# print(G.count(rep_seq))

# Initialize the coverage histogram
print("Generating reads...");
for i in range(L):
   coverage_hist.append(0);
if (read_method_param == 1):
   read_method1(G, coverage_hist, readset, read_metadata_set, N, l,
         d);
elif (read_method_param == 2):
   read_method2(G, coverage_hist, readset, read_metadata_set, N, l,
         d);
else:
   print("Invalid read method parameter");
   exit();
# read method wiped out from here
# now generage biological mutation array (this needs to go in above
      else part)
# mutation_pos = random.sample(range(0,len(G)-1), num_mutations);
# insert_mutations(mutation_array, mutation_pos, readset,
      read_metadata_set, G);
# insert_errors(error_percentage, readset, read_metadata_set);

# Input read/generated

##############Now writing read and genome data to the log
      file##########

print("Writing read and genome data to the log file...");
# f_log.write(
#   "Read data:\nThe first two columns are the reads\nThe third
      column gives flags about forward/reverse read, mutations and
      errors\nFourth and fifth column are self explanatory\nSixth
      and seventh column give the mutation and error data
      respectively\nThe mutation and error data are list of
      triples\n\tThe first entity giving out the read in the current
      pair\n\tThe second entity is the position\n\tThe third
      position represents the originally present base\n\n");
f_log.write(str(len(readset)) + "\n")
for i in range(len(readset)):
   f_log.write(readset[i][0] + "\t\t" + readset[i][1] + "\t\t" +
         read_metadata_set[i][0] + "\tStart pos=(" +
            read_metadata_set[i][1] + "," + read_metadata_set[i][2] +
                  ")\tDist=" + read_metadata_set[i][3]);
   for j in range(4, len(read_metadata_set[i])):
      print("\t", read_metadata_set[i][j], end='', file=f_log);
      f_log.write("\n");

f_log.write("\n\n\nGenome and Coverage\n");
for i in range(int(math.ceil(float(len(coverage_hist)) /
      float(per_line_param)))):
   f_log.write(G[i * per_line_param:(i + 1) * per_line_param] + "\n");
   for j in range(i * per_line_param, min((i + 1) * per_line_param,
         len(coverage_hist))):
      f_log.write(str(coverage_hist[j]) + " ");
   f_log.write("\n\n");

f_log.write("\n\n\nGenome as one string\n");
f_log.write(G);
f_log.write("\n\n");

f_log.write("\n\n\nGenome on the opposite strand\n");
f_log.write(complement(G[::-1]));
f_log.write("\n\n");

print("1", file=f_genome_out);
print(G, file=f_genome_out);

# plotting the coverage histogram
if (covg_hist_plot_needed):
   plt.plot(coverage_hist);
   plt.xlabel('Genomic position');
   plt.ylabel('Coverag');
   plt.show();

fin.close();
f_log.close();
f_err.close();
# f_reread.close();
f_genome_out.close();
if (not read_genome_from_file == "none"):
   f_genome.close();

# cluster_main_with_log(G, readset, read_metadata_set)
cluster_for_repeats(G, readset, read_metadata_set)
```

Listing A.2: k-mer counting functions

```
#!/usr/bin/python3.4
from __future__ import print_function
import math, sys
# import matplotlib.pyplot as plt
import re, os, heapq

alphabet = ['A', 'C', 'G', 'T']
fin = ''
# f_walkin='';
f_log = ''
f_err = ''
kmer_k = 7

main_log = open("mainlog.txt", 'a')
err_log = open("errlog.txt", 'w')
readlen = 40
# TODO: add more stable threshold function
threshold = (readlen * 2 - kmer_k) * 2


# return number of occurences of pattern p in string s
# note that in this method, it will count overlapping patterns as
      twice
# while in s.count() method, overlapping patterns will be skipped
def cnt_recurrence(s, p):
   poslist = []
   for m in re.finditer('(?=' + p + ')', s):
      poslist.append(int(m.start()))
```

```
    return len(poslist)


def create_repeats(G, sample_pos, rep_len, rep_pos_list):
    rep_seq = G[sample_pos:sample_pos + rep_len]
    print('Repeated region is:')
    print(rep_seq)

    for i in range(len(rep_pos_list)):
        G = G[:rep_pos_list[i]] + rep_seq + G[rep_pos_list[i] +
              rep_len:]

    return G


# complement a string (kmer)
# string is immutable, so we have to create new strings by
      concatenating old ones
def complement(S):
    for i in range(len(S)):
        if (S[i] == 'A'):
            S = S[:i] + 'T' + S[i + 1:];
        elif (S[i] == 'C'):
            S = S[:i] + 'G' + S[i + 1:];
        elif (S[i] == 'G'):
            S = S[:i] + 'C' + S[i + 1:];
        elif (S[i] == 'T'):
            S = S[:i] + 'A' + S[i + 1:];
    return S;


# data structure for readpair
# also acts as data structure for clustered readpairs
# TODO: add a variable counting how many original reads are merged
      into the current read pair
class Readpair:
    # pos1, pos2 denote the starting positions of the reads
    # first and second store the actual reads
    # dist is read distance, not used
    def __init__(self, isReverse, pos1, pos2, first, second, dist):
        self.isReverse = isReverse
        self.startpos1 = pos1
        self.startpos2 = pos2
        self.first = first
        self.second = second
        self.dist = dist
        self.group = -1

    # find out which kmer it belongs to
    def findGroup(self, kmer_pos_list):
        g1 = find_kmer_ord(kmer_pos_list, self.startpos1,
              len(self.first))
        g2 = find_kmer_ord(kmer_pos_list, self.startpos2,
              len(self.second))
        if g1 != -1:
            self.group = g1
        elif g2 != -1:
            self.group = g2
        else:
            self.group = -1

    # calculate the overlap score of this readpair with another
          readpair
    # currently the score is aligned_overlap_length *
          suffix_prefix_overlap_length
    def overlapScore(self, other, kmer):
        maxscore = -1
        s1 = self.first
        s2 = self.second
        t1 = other.first
        t2 = other.second
        s1p = complement(s1[::-1])
        s2p = complement(s2[::-1])
        t1p = complement(t1[::-1])
        t2p = complement(t2[::-1])

        # four possible cases for the double alignment
        anstuple1 = doubleAlignment(s1, s2, t1, t2, kmer)
        anstuple = anstuple1
        if anstuple1[0] > maxscore:
            maxscore = anstuple1[0]
            anstuple = anstuple1

        anstuple2 = doubleAlignment(s1, s2, t2p, t1p, kmer)
        if anstuple2[0] > maxscore:
            maxscore = anstuple2[0]
            anstuple = anstuple2

        anstuple3 = doubleAlignment(s2p, s1p, t1, t2, kmer)
        if anstuple3[0] > maxscore:
            maxscore = anstuple3[0]
            anstuple = anstuple3

        anstuple4 = doubleAlignment(s2p, s1p, t2p, t1p, kmer)
        if anstuple4[0] > maxscore:
            maxscore = anstuple4[0]
            anstuple = anstuple4

        return anstuple

    def __str__(self):
        s = ''
        if not self.isReverse:
            s += 'F '
        else:
            s += 'R '
        s += str(self.group)
        s += ' ' + self.first + ' ' + self.second + ' '
        s += str(self.startpos1) + ' ' + str(self.startpos2) + ' ' +
              str(self.dist)
        return s


# compute overlap score from the value of aligned overlap and
      suffix-prefix overlap
def overlap_score(aligned_overlap, sufpref_overlap):
    if sufpref_overlap < 5 or aligned_overlap < kmer_k:
        return 0
    else:
        return aligned_overlap * sufpref_overlap


# return overlapping lengths of string pairs (s1,s2) and (t1,t2),
      aligning their common k-mer
# all possibilities are considered, but we only allow (s1 and t1
      aligned) or (s2 and t2 aligned)
def doubleAlignment(s1, s2, t1, t2, kmer):
    maxscore = -1

    # note that overlap2one and overlapone functions are ordered, so
          params have to be fed with 4 different orders
    aligned_ov1a, string11a = overlap2one(s1, t1, kmer)
    sufpref_ov1a, string12a = overlapone(s2, t2)
    anstuple1a = (overlap_score(aligned_ov1a, sufpref_ov1a),
          string11a, string12a)
    if anstuple1a[0] > maxscore:
        maxscore = anstuple1a[0]
        anstuple = anstuple1a

    aligned_ov1b, string11b = overlap2one(s1, t1, kmer)
    sufpref_ov1b, string12b = overlapone(t2, s2)
    anstuple1b = (overlap_score(aligned_ov1b, sufpref_ov1b),
          string11b, string12b)
    if anstuple1b[0] > maxscore:
        maxscore = anstuple1b[0]
        anstuple = anstuple1b

    aligned_ov1c, string11c = overlap2one(t1, s1, kmer)
    sufpref_ov1c, string12c = overlapone(s2, t2)
    anstuple1c = (overlap_score(aligned_ov1c, sufpref_ov1c),
          string11c, string12c)
    if anstuple1c[0] > maxscore:
        maxscore = anstuple1c[0]
        anstuple = anstuple1c

    aligned_ov1d, string11d = overlap2one(t1, s1, kmer)
```

```
    sufpref_ov1d, string12d = overlapone(t2, s2)
    anstupleld = (overlap_score(aligned_ov1d, sufpref_ov1d),
           string11d, string12d)
    if anstupleld[0] > maxscore:
       maxscore = anstupleld[0]
       anstuple = anstupleld

    #
           ======================================================================

    aligned_ov1a, string11aa = overlap2one(s2, t2, kmer)
    sufpref_ov1a, string12aa = overlapone(s1, t1)
    anstuplelaa = (overlap_score(aligned_ov1a, sufpref_ov1a),
           string11aa, string12aa)
    if anstuplelaa[0] > maxscore:
       maxscore = anstuplelaa[0]
       anstuple = anstuplelaa

    aligned_ov1b, string11bb = overlap2one(t2, s2, kmer)
    sufpref_ov1b, string12bb = overlapone(s1, t1)
    anstuplelbb = (overlap_score(aligned_ov1b, sufpref_ov1b),
           string11bb, string12bb)
    if anstuplelbb[0] > maxscore:
       maxscore = anstuplelbb[0]
       anstuple = anstuplelbb

    aligned_ov1c, string11cc = overlap2one(s2, t2, kmer)
    sufpref_ov1c, string12cc = overlapone(t1, s1)
    anstuplelcc = (overlap_score(aligned_ov1c, sufpref_ov1c),
           string11cc, string12cc)
    if anstuplelc[0] > maxscore:
       maxscore = anstuplelcc[0]
       anstuple = anstuplelcc

    aligned_ov1d, string11dd = overlap2one(t2, s2, kmer)
    sufpref_ov1d, string12dd = overlapone(t1, s1)
    anstupleldd = (overlap_score(aligned_ov1d, sufpref_ov1d),
           string11dd, string12dd)
    if anstupleldd[0] > maxscore:
       maxscore = anstupleldd[0]
       anstuple = anstupleldd

    return anstuple


# suffix-prefix overlap in fixed order: s1 comes before s2
# no other combinations are tried
def overlapone(s1, s2):
    # print('Finding suffix-prefix overlap between:')
    # print(s1)
    # print(s2)
    l = min(len(s1), len(s2))
    # s1p = complement(s1[::-1])
    # s2p = complement(s2[::-1])

    ans = []
    newstr = []

    # deal with substrings
    if s1.find(s2) != -1:
       return l, s1

    # i,j
    for p in range(1, l, -1):
       if s1[-p:] == s2[0:p]:
          ans.append(p)
          newstr.append(s1 + s2[p:])
          break

    if len(ans) == 0:
       return 0, ""
    maxov = max(ans)
    # print('Results of the four alignments:')
    # print(ans)
    # print(newstr)
    return maxov, newstr[ans.index(maxov)]


# aligned overlap in fixed order
# no other flipped versions of s1 or s2 are tried
def overlap2one(s1, s2, kmer):
    # s1p = complement(s1[::-1])
    # s2p = complement(s2[::-1])

    allans = []
    allstr = []
    al, str1 = matched_overlap(s1, s2, kmer)
    allans.append(al)
    allstr.append(str1)

    # al, str1 = matched_overlap(s1p, s2, kmer)
    # allans.append(al)
    # allstr.append(str1)
    #
    # al, str1 = matched_overlap(s1, s2p, kmer)
    # allans.append(al)
    # allstr.append(str1)
    #
    # al, str1 = matched_overlap(s1p, s2p, kmer)
    # allans.append(al)
    # allstr.append(str1)

    if len(allans) == 0:
       return 0, ""
    maxov = max(allans)
    # print(ans)
    # print(newstr)
    return maxov, allstr[allans.index(maxov)]


# find out which kmer the read contains
# TODO: there could be reads containing multiple kmers
def find_kmer_ord(kmer_pos_list, readpos, readlen):
    for n in range(len(kmer_pos_list)):
       if readpos <= kmer_pos_list[n] < readpos + readlen:
          return n
    return -1


# construct a readpair set using original readset and the
      read_metadata_set
def construct_readpairset(readset, read_metadata_set):
    readpairset = []

    for i in range(len(readset)):
       isReversed = False
       read1 = readset[i][0]
       read2 = readset[i][1]
       startpos1 = int(read_metadata_set[i][1])
       startpos2 = int(read_metadata_set[i][2])
       dist = int(read_metadata_set[i][3])

       # if reversed, update the starting point information
       if read_metadata_set[i][0] == 'R':
          isReversed = True
          startpos1 -= len(read1) - 1
          startpos2 -= len(read2) - 1

       newReadPair = Readpair(isReversed, startpos1, startpos2, read1,
             read2, dist)
       readpairset.append(newReadPair)
    return readpairset


# doubly merging all readpairs
# TODO: finally we would want to ignore all groups with very few read
      pairs inside it
# TODO: Current implementation is a very slow O(n^3), with a lot of
      repeated computations
def merge_all_readpairs2(G, kmer, rpset):
    print('Beginning doubly merging all readpairs')
    t1 = t2 = rpset[0]
    i1 = i2 = 0

    while True:
```

```
        print('length of read pair set is:', len(rpset))
        maxoverlap = 0
        ans1 = ''
        ans2 = ''
        size = len(rpset)
        for r1 in range(size - 1):
            for r2 in range(r1 + 1, size):
                if r1 != r2:
                    ov, a1, a2 = rpset[r1].overlapScore(rpset[r2], kmer)
                    if ov > maxoverlap:
                        maxoverlap = ov
                        ans1 = a1
                        ans2 = a2
                        t1 = rpset[r1]
                        t2 = rpset[r2]

                    ov, a1, a2 = rpset[r1].overlapScore(rpset[r2],
                          complement(kmer[::-1]))
                    if ov > maxoverlap:
                        maxoverlap = ov
                        ans1 = a1
                        ans2 = a2
                        t1 = rpset[r1]
                        t2 = rpset[r2]

        print('In this iteration, max score is: ', maxoverlap)
        if (cnt_recurrence(G, ans1) == 0 and cnt_recurrence(G,
               complement(ans1[::-1])) == 0) or (
                    cnt_recurrence(G, ans2) == 0 and cnt_recurrence(G,
                          complement(ans2[::-1])) == 0):
            print('Wrong merge!')
            print(t1)
            print(t2)
            print('first: ', ans1, ' ', 'second: ', ans2)

        if maxoverlap > 150:
            # othersset += other
            rpset.remove(t1)
            rpset.remove(t2)
            startposlist = sorted([t1.startpos1, t1.startpos2,
                  t2.startpos1, t2.startpos2])
            newrp = Readpair(False, startposlist[0], startposlist[2],
                  ans1, ans2, -1)
            rpset.append(newrp)
        else:
            break
    print('Now printing final rpset')
    rec = 0
    for rp in rpset:
        print(rp.startpos1, ' ', rp.startpos2, ' ', rp.first, ' ',
              rp.second)
    return int((len(rpset) + 1) / 2)


def cluster_for_repeats(G, readset, read_metadata_set):
    print(G)
    casenum = 0
    error = []
    hasError = False
    L = len(G)
    if not os.path.exists("./mergelogs"):
        os.makedirs("mergelogs")
    # for i in range(0, L - kmer_k, 1):

    for i in range(400, 500, 11):
        kmer = G[i:i + kmer_k]
        kmerp = complement(kmer[::-1])
        # correctans = G.count(kmer) + G.count(kmerp)
        casenum += 1
        print("For the kmer: " + kmer + "\tstarting at position " +
              str(i) + "\t#case is " + str(casenum),
              file=main_log)
        print("For the kmer: " + kmer + "\tstarting at position " +
              str(i) + "\t#case is " + str(casenum))
        kmer_pos_list = []
        for m in re.finditer('(?=' + kmer + ')', G):
            kmer_pos_list.append(int(m.start()))
        for m in re.finditer('(?=' + kmerp + ')', G):
            kmer_pos_list.append(int(m.start()))

        # print(kmer_pos_list, file=lg)
        kmer_pos_list = sorted(kmer_pos_list)
        correctans = len(kmer_pos_list)
        print("It appears in G at following positions:", file=main_log)
        print("It appears in G at following positions:")
        print(kmer_pos_list, file=main_log)
        print(kmer_pos_list)

        # find # occurences of this kmer in genome
        print("So Correct answer is: " + str(correctans), file=main_log)
        print("So Correct answer is: " + str(correctans))

        # find out relevent reads and their corresponding metadata
        newreadset = []
        newreadmetaset = []
        for i in range(len(readset)):
            if readset[i][0].find(kmer) != -1 or
                    readset[i][0].find(kmerp) != -1 or
                    readset[i][1].find(kmer) != -1 or \
                         readset[i][1].find(kmerp) != -1:
                newreadset.append(readset[i])
                newreadmetaset.append(read_metadata_set[i])

        # construct readpair set
        rpset = construct_readpairset(newreadset, newreadmetaset)
        for rp in rpset:
            rp.findGroup(kmer_pos_list)
            # print(rp)
        ourans = merge_all_readpairs2(G, kmer, rpset)
        print('Our answer is: ' + str(ourans))
        error.append(ourans - correctans)
    print('Among all runs, the errors are like:')
    print(error)


# auxiliary function to facilitate sorting
def getOrd(item):
    return item[2]


# auxiliary function to print out a readset
def print_readset(readset, plotnum, order):
    fname = str(plotnum) + "_" + str(order) + ".txt"
    fout = open("./mergelogs/" + fname, 'w')
    sorted(readset, key=getOrd)
    for read in readset:
        fout.write(str(read[1]) + "\t" + str(read[2]) + "\t" + read[0]
              + "\n")
    fout.close()


def pass_file_objects(f1, f3, f4):
    global fin
    # global f_walkin;
    global f_log
    global f_err
    fin = f1
    # f_walkin=f2;
    f_log = f3
    f_err = f4
```

A.2 Minhash and One-permutation Overlap Finding Algorithms

Listing A.3: Sequential Implementation of Minhash and one-permutation overlap finding functions

```
from __future__ import print_function

from myutil import *
from read_sequencing import getUniform
import time


# TODO: Test this code and implement the O(n^2) straightforward method

class Methods:
   original, onePermutation = range(2)


M = 100000000
HA = 1664525
HC = 1442695040888963407
B = 300
T = 3
K = 7
pair_cnt = 0
R = 1

num_hash_funcs = 2
clustering_method = Methods.onePermutation


# generate a hash for a list containing T elements
def hash_for_T_list(list_of_T):
   ans = 0
   for n in list_of_T:
      ans = (ans * HA + n) % M
   return ans


# original hash function for a k-mer
def orig_hash(kmer):
   kmerp = complement(kmer[::-1])
   if kmerp <= kmer:
      s = kmerp
   else:
      s = kmer
   ans = 0
   for i in range(len(s)):
      ans = (HA * ans % M + ord(s[i]) - ord('A')) % M
   return ans


# create N=B*T hash functions
def hash_base_pool():
   bases = getUniform(0, M, B * T)
   for i in range(len(bases)):
      bases[i] = int(round(bases[i]))
   return bases


# use the t-th hash function within the b-th block on s
def hash_func(s, bases, b, t):
   return orig_hash(s) ^ bases[b * T + t]


# pick the min hash for a read using the t-th hash function within
      b-th block
def hash_for_read(read, bases, b, t):
   minhash = M + 1
   length = len(read)
   for i in range(length - K):
      cur = hash_func(read[i:i + K], bases, b, t)
      if cur < minhash:
         minhash = cur
   return minhash


# hashtable_for_rows is a dictionary
# keys are hash_for_T_list
# values are [[t-item-list, rows that have this t-item-list],
# [another list, rows that have this list]]
def update_hashtable(hashtable_for_rows, vals_in_b, rownum):
   h = hash_for_T_list(vals_in_b)
   if h not in hashtable_for_rows:
      hashtable_for_rows[h] = [[vals_in_b, rownum]]
      return
   nowval = hashtable_for_rows[h]
   found = False
   for thislist in nowval:
      if thislist[0] == vals_in_b:
         found = True
         thislist.append(rownum)
   if not found:
      nowval.append([vals_in_b, rownum])


# create clusters based on hash val, hash vals are not stored
def cluster_minhash(pairs_dic, N, singlereadset, bases):
   # clusterMat = [[0 for x in range(N)] for y in range(N)]
   for b in range(B):
      # if b % 100 == 0:
      print("Processing block %d of %d" % (b, B))
      hashtable_for_rows = {}
      for i in range(len(singlereadset)):
         vals_in_b = []
         r = singlereadset[i]
         for t in range(T):
            vals_in_b.append(hash_for_read(r, bases, b, t))
         update_hashtable(hashtable_for_rows, vals_in_b, i)
      allvals = hashtable_for_rows.values()

      for v in allvals:
         for lis in v:
            for ind1 in range(1, len(lis)):
               for ind2 in range(ind1 + 1, len(lis)):
                  minind = min(lis[ind1], lis[ind2])
                  maxind = max(lis[ind1], lis[ind2])
                  if not minind in pairs_dic:
                     pairs_dic[minind] = {maxind}
                  else:
                     pairs_dic[minind].add(maxind)


def one_perm_hash_func(s, l, bases):
   return (orig_hash(s) ^ bases[l]) % M


# one-permutation method
def one_permutation_clustering(pairs_dic, N, singlereadset, bases):
   range_len = M / 10
   for l in range(num_hash_funcs):
      print("Processing hash function %d of %d" % (l, num_hash_funcs))
      rlt = {}
      for i in range(len(singlereadset)):
         if i % 1000 == 0:
            print("processing", i, 'th read')
         read = singlereadset[i]
         minhashes = [int(i * range_len) for i in range(10 + 1)]
         touched = [False for i in range(10 + 1)]
         length = len(read)
         for j in range(length - K):
            cur = one_perm_hash_func(read[j:j + K], l, bases)
            # print(cur)
            b = int(cur / range_len) + 1
            if cur < minhashes[b]:
               minhashes[b] = cur
               touched[b] = True
         for k in range(10 + 1):
            if touched[k]:
               if not minhashes[k] in rlt.keys():
                  rlt[minhashes[k]] = [i]
               else:
```

```
                rlt[minhashes[k]].append(i)
        print('adding to pairs dictionaries ...')
        for hash_v in rlt.keys():
            lis = rlt[hash_v]
            for ind1 in range(1, len(lis)):
                for ind2 in range(ind1 + 1, len(lis)):
                    minind = min(lis[ind1], lis[ind2])
                    maxind = max(lis[ind1], lis[ind2])
                    if not minind in pairs_dic:
                        pairs_dic[minind] = {maxind}
                    else:
                        pairs_dic[minind].add(maxind)


def test(G, readset, read_metadata_set):
    newreadset = []
    newmetaset = []
    for i in range(len(readset)):
        read1 = readset[i][0]
        read2 = readset[i][1]
        startpos1 = int(read_metadata_set[i][1])
        startpos2 = int(read_metadata_set[i][2])

        # if reversed, update the starting point information
        # if read_metadata_set[i][0] == 'R':
        #    startpos1 -= len(read1) - 1
        #    startpos2 -= len(read2) - 1

        newreadset.append(read1)
        newreadset.append(read2)
        newmetaset.append(startpos1)
        newmetaset.append(startpos2)

    N = len(newreadset)
    print("Number of reads: ", N)

    # find out true pairs
    print('begin calculating true pairs')
    superreadset = []
    truedic = {}
    for i in range(len(newreadset)):
        superreadset.append((i, newmetaset[i]))
    superreadset.sort(key=lambda x: x[1])
    for p in range(len(superreadset)):
        # curset=[]
        curind = superreadset[p][0]
        j = p + 1
        endpos = superreadset[p][1] +
              len(newreadset[superreadset[p][0]])
        while j < len(superreadset) and superreadset[j][1] < endpos -
              60:
            otherind = superreadset[j][0]
            minind = min(curind, otherind)
            maxind = max(curind, otherind)
            if not minind in truedic:
                truedic[minind] = {maxind}
            else:
                truedic[minind].add(maxind)
            j += 1
    true_pair_cnt = 0
    truedicout = open('realpairs.txt', 'w')
    keys = truedic.keys()
    print(len(keys), file=truedicout)
    for k in keys:
        ansstr = ''
        ansstr += str(k) + ' :'
        for other in truedic[k]:
            ansstr += ' ' + str(other)
        true_pair_cnt += len(truedic[k])
        print(ansstr, file=truedicout)

    print("Number of true pairs is:", true_pair_cnt)

    pair_dic = {}

    bases = hash_base_pool()

    cluster_start_time = time.time()
    if clustering_method == Methods.onePermutation:
        one_permutation_clustering(pair_dic, N, newreadset, bases)
    else:
        cluster_minhash(pair_dic, N, newreadset, bases)
    cluster_stop_time = time.time()
    print('Clustering process time: %s seconds' % (cluster_stop_time -
          cluster_start_time))

    # write out pairs to file
    total_pairs_filtered = 0
    dicout = open('pairs.txt', 'w')
    # for i in range(N):
    #    for j in range(i + 1, N):
    #        total_pairs_filtered += clusterMat[i][j]
    keys = pair_dic.keys()
    print(len(keys), file=dicout)
    for k in keys:
        ansstr = ''
        ansstr += str(k) + ' :'
        for other in pair_dic[k]:
            ansstr += ' ' + str(other)
        print(ansstr, file=dicout)
        total_pairs_filtered += len(pair_dic[k])

    print("Total pairs we picked out: ", total_pairs_filtered)
    print("Out of", N * (N - 1) / 2, "possible pairs")

    # count number of false positives and false negatives
    false_pos = 0
    false_neg = 0
    for k in range(N):
        if not k in truedic and not k in pair_dic:
            continue
        if not k in truedic:
            false_pos += len(pair_dic[k])
            continue
        if not k in pair_dic:
            false_neg += len(truedic[k])
            continue
        false_pos += len(pair_dic[k].difference(truedic[k]))
        false_neg += len(truedic[k].difference(pair_dic[k]))

    print("False positive pairs:", false_pos)
    print("False negative (missed) pairs:", false_neg)

    print('False positive rate:', false_pos / total_pairs_filtered)
    print('False negative (miss) rate:', false_neg / true_pair_cnt)
```

REFERENCES

[1] F. Sanger, S. Nicklen, and A. R. Coulson, "DNA sequencing with chain-terminating inhibitors," *Proceedings of the National Academy of Sciences*, vol. 74, no. 12, pp. 5463–5467, Dec. 1977.

[2] R. D. Fleischmann, M. D. Adams, O. White, *et al.*, "Whole-Genome Random Sequencing and Assembly of Haemophilus Influenzae Rd," *Science*, vol. 269, no. 5, pp. 496–498, Jul. 1995.

[3] A. Kalyanaraman, S. J. Emrich, P. S. Schnable, *et al.*, "Assembling genomes on large-scale parallel computers," in *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, IEEE, 2006.

[4] E. Pettersson, J. Lundeberg, and A. Ahmadian, "Generations of sequencing technologies," *Genomics*, vol. 93, no. 2, pp. 105–111, Feb. 2009.

[5] J. R. Miller, S. Koren, and G. Sutton, "Assembly algorithms for next-generation sequencing data," *Genomics*, vol. 95, no. 6, pp. 315–327, Jun. 2010.

[6] K. Berlin, S. Koren, C.-S. Chin, *et al.*, "Assembling large genomes with single-molecule sequencing and locality-sensitive hashing," *Nature Biotechnology*, vol. 33, no. 6, pp. 623–630, Jun. 2015.

[7] J. F. Denton, J. Lugo-Martinez, A. E. Tucker, *et al.*, "Extensive Error in the Number of Genes Inferred from Draft Genome Assemblies," *Plos Computational Biology*, vol. 10, no. 12, Dec. 2014.

[8] S. M. Huse, J. A. Huber, H. G. Morrison, *et al.*, "Accuracy and quality of massively parallel DNA pyrosequencing," *Genome biology*, vol. 8, no. 7, 2007.

[9] L. Liu, Y. Li, S. Li, *et al.*, "Comparison of next-generation sequencing systems," *Journal of Biomedicine and Biotechnology*, 2012.

[10] E. Ukkonen, "Approximate String-Matching with Q-Grams and Maximal Matches," *Theoretical Computer Science*, vol. 92, no. 1, pp. 191–211, 1992.

[11] M. Schirmer, U. Z. Ijaz, R. D'Amore, *et al.*, "Insight into biases and sequencing errors for amplicon sequencing with the Illumina MiSeq platform," *Nucleic Acids Research*, vol. 43, no. 6, Jan. 2015.

[12] K. Nakamura, T. Oshima, T. Morimoto, *et al.*, "Sequence-specific error profile of Illumina sequencers," *Nucleic Acids Research*, vol. 39, no. 13, May 2011.

[13] J. T. Simpson, K. Wong, S. D. Jackman, *et al.*, "ABySS: A parallel assembler for short read sequence data," *Genome Research*, vol. 19, no. 6, pp. 1117–1123, Jun. 2009.

[14] P. A. Pevzner, H. Tang, and M. S. Waterman, "An Eulerian path approach to DNA fragment assembly," *Proceedings of the National Academy of Sciences*, vol. 98, no. 17, pp. 9748–9753, Aug. 2001.

[15] D. R. Zerbino and E. Birney, "Velvet: Algorithms for de novo short read assembly using de Bruijn graphs," *Genome Research*, vol. 18, no. 5, pp. 821–829, May 2008.

[16] I. MacCallum, D. Przybylski, S. Gnerre, *et al.*, "ALLPATHS 2: small genomes assembled accurately and with high continuity from short paired reads," *Genome biology*, vol. 10, no. 10, 2009.

[17] H. Li, "Minimap and miniasm - fast mapping and de novo assembly for noisy long sequences.," *Bioinformatics*, vol. 32, no. 14, pp. 2103–2110, 2016.

[18] G. Myers, "Efficient Local Alignment Discovery amongst Noisy Long Reads.," *WABI*, 2014.

[19] N. J. Loman, J. Quick, and J. T. Simpson, "A complete bacterial genome assembled de novo using only nanopore sequencing data," *Nature Methods*, vol. 12, no. 8, pp. 733–735, Aug. 2015.

[20] E. W. Myers, G. G. Sutton, A. L. Delcher, *et al.*, "A Whole-Genome Assembly of Drosophila," *Science*, vol. 287, no. 5, pp. 2196–2204, Mar. 2000.

[21] J. Chu, H. Mohamadi, R. Warren, *et al.*, "Overlapping long sequence reads: Current innovations and challenges in developing sensitive, specific and scalable algorithms," *bioRxiv*, Oct. 2016.

[22] A. Z. Broder, "On the resemblance and containment of documents," in *Compression and Complexity of SEQUENCES 1997*, IEEE Comput. Soc, 1997, pp. 21–29.

[23] G. Gonnet and R. Baeza-Yates, "An analysis of the Karp-Rabin string matching algorithm," *Information Processing Letters*, vol. 34, no. 5, pp. 271–274, May 1990.

[24] P. Li, A. Owen, and C.-H. Zhang, "One Permutation Hashing for Efficient Search and Learning," *arXiv.org*, Aug. 2012. arXiv: `1208.1259v1 [cs.LG]`.