December 2016

Enhancing the Draft Assembly with Minhash

Saju Varghese
University of Nevada, Las Vegas, sajuv725@gmail.com

Repository Citation

Varghese, Saju, "Enhancing the Draft Assembly with Minhash" (2016). *UNLV Theses, Dissertations, Professional Papers, and Capstones*. 2910.
https://digitalscholarship.unlv.edu/thesesdissertations/2910

ENHANCING THE DRAFT ASSEMBLY WITH MINHASH

By

Saju Varghese

Bachelor of Science - Computational Physics
Bachelor of Science - Computer Science
University of Nevada, Las Vegas
2013

A thesis submitted in partial fulfillment of
the requirements for the

Master of Science in Computer Science

Department of Computer Science
Howard R. Hughes College of Engineering
The Graduate College

University of Nevada, Las Vegas
December 2016

Thesis Approval

The Graduate College
The University of Nevada, Las Vegas

November 17, 2016

This thesis prepared by

Saju Varghese

entitled

Enhancing the Draft Assembly with Minhash

is approved in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science
Department of Computer Science

Kazem Taghva, Ph.D.
Examination Committee Chair

Ajoy Datta. Ph.D.
Examination Committee Member

Laxmi Gewali, Ph.D.
Examination Committee Member

Emma Regentova, Ph.D.
Graduate College Faculty Representative

Kathryn Hausbeck Korgan, Ph.D.
Graduate College Interim Dean

Abstract

In this thesis, we report on the use of minhash techniques to improve the draft assembly of a genome mapping. More specifically, we use minhash to compare the scaffolds of sea urchin and sea cucumber genomes.

One of the main contributions of this thesis is the implementation of minhash with the Message Passing Interface (MPI) utilizing Intel Phi co-processors. It is shown that our implementation significantly reduces the processing time for identification of k-mer similarities.

Acknowledgements

"I would like to thank my advisory committee, particularly Dr. Taghva, for their support, patience and guidance during my studies. I also want to give special thanks to John Kowalski, Ron Young and Switch for allowing me to use their resources towards my project. I would like to especially thank my family, my other family at Slickdeals and my friends who have supported me constantly these past three years and have been there for me as a source of encouragement, inspiration and support."

Saju Varghese

University of Nevada, Las Vegas

December 2016

Table of Contents

List of Tables

List of Figures

Table of Listings

Chapter 1

Introduction

According to https://www.hgsc.bcm.edu/other-invertebrates/sea-urchin-genome-project, "Echinoderms occupy an important evolutionary position with respect to vertebrates and humans: they, along with their sister phylum hemichordates, are the closest known relatives to chordates." The sea urchin genome mapping project as reported by Sea Urchin Genome Sequencing Consortium http://science.sciencemag.org/content/314/5801.toc is one of the well-studied topics in marine ecosystems. As a part of a bigger project for mapping of genome of sea cucumber, we were interested in alignment of certain scaffolds of sea cucumber and sea urchin.

Our efforts in mapping of sea cucumber genome began with approximately 450 million paired-end reads which were prepared at University of Southern California. In this thesis, we will report on discovery of similarities between the scaffolds of the genomes of sea cucumber and sea urchin using the minhash technique. The second chapter covers the background on genome mapping - using FastQC to check the quality of the reads followed by Trimmomatic to trim out bad reads then running String-Graph Assembler to form contigs which fed to SSPACE and GapFiller forming scaffolds.

The third chapter goes over the background of the minhash technique invented by Andrei Broder [11], a widely-used technique to find similarity between images, documents, and web pages that utilizes shingles and hashing with a distance measure to compute the similarity between two entities. The chapter will cover the basics of minhash technique.

The fourth and fifth chapters introduce our experiments and implementation of minhash technique. We discuss the data preparations and various genome post processing tools that are referred to in Chapter 1. We then provide the details of our implementation of minhash technique to compute the similarity between sea cucumber and sea urchin scaffolds.

Chapter six discusses the results of our experiments. At last, Chapter seven is our conclusions and future work associated with the assembly of sea cucumber.

Chapter 2

Genome Mapping

The deoxyribonucleic acid (DNA) is the building block of a living creature, it is unique to each being and guides its development. The DNA is a double helix composed of four bases Adenine (A), Cytosine (C), Guanine (G), and Thymine (T). These bases are attached by a backbone [1]. Genes are formed from specific sections of a DNA, and control different characteristics such as eye color and height [4].

The `genome` is an organism's complete set of DNA, including all of its genes [5]. Each strand of DNA can be viewed as a long string on the alphabet { A, C, G, T }. The size of this string is dependent on the living species, for example, the length of human genome is 3.2 billion base pairs or roughly 3GB in raw text, while the length of a virus' DNA is a few million base pairs. In order to build a complete genome, scientists have to first generate millions of reads/sequences from small fragments of DNA using sequencing machines such as those built by Illumina. The task of genome mapping is the process of creating the complete DNA from these reads. There are two basic approaches to genome mapping: Reference based or De novo. In reference based approach, if a closely related species' DNA (a reference) is known, one can use this reference as a template to build the genome. In the De novo sequencing approach, one basically arranges the reads without any reference.

A good analogy would be to compare the Illumina machine as a jigsaw puzzle creator, to which you would feed in a picture that you've never seen before and watch it generate jigsaw pieces for you that you would have to put together. Using De novo method, you will be slowly placing each piece next to the one that you assume is the correct location for it. In a reference based mapping, you are given a copy of a closely related image. One can use this image to figure out where to place the pieces. In our case, we are trying to sequence the 450 million paired-end Sea Cucumber reads that we received from University of Southern California utilizing the Sea Urchin's genome as a reference.

2.1 FASTQ file format

Before we can assemble the reads into the complete genome, we first have to pass the reads through FastQC to get an understanding of the quality of the reads. The reads are stored in FASTQ file formats. Each read is composed of four lines as described below [13]:

1. Line 1 begins with a '@' character and is followed by a sequence identifier and an *optional* description
2. Line 2 is the raw sequence letters (151 characters)
3. Line 3 begins with a '+' character and is *optionally* followed by the same sequence identifier (and any description) again
4. Line 4 encodes the quality values for the sequence in Line 2, and must contain the same number of symbols as letters in the sequence

For example:

```
@SEQ_ID
GATTTGGGGTTCAAAGCAGTATCGATCAAATAGTAAATCCATTTGTTCAACTCACAGTTT
+
!''*((((***+))%%%++)(%%%%).1***-+*''))**55CCF>>>>>>CCCCCCC65
```

Listing 2.1: FASTQ file format

2.2 Phred Quality Score

A Phred quality score is a measure of the quality of the identification of nucleobases generated by automated DNA sequencing. A score is attached to each nucleotide base-call in automated sequencer traces. Phred quality score, Q, is defined as a property which is logarithmically related to the base-calling error probabilities, P [13]:

$$Q = -10 \log_{10} P \tag{2.1}$$

or

$$P = 10^{\frac{-Q}{10}} \tag{2.2}$$

A Phred quality score of 20 for a nucleotide base means the chances of the base being incorrect is 1 in a 100, or 99%, whereas a Phred quality score of 60 means, the chances of it being incorrect is 1 in a 1000000 or 99.9999%.

In a FASTQ file, the Phred quality score is represented as an ASCII value. The quality scores ranges from low quality ! to ~ :

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`
    abcdefghijklmnopqrstuvwxyz{|}~
```

Listing 2.2: Phred quality scores

The following Python code converts between ASCII value and Phred33 code:

```python
# Convert Q into Phred33 ASCII-encoded quality
def QToPhred33(Q):
    return chr(Q + 33)

# Convert Phred33 ASCII-encoded quality into Q
def Phred33ToQ(quality):
    return (ord(quality) - 33)
```

Listing 2.3: Converting between ASCII value and Phred33 code

2.3 FastQC

FastQC runs the reads through a series of tests to verify their quality, which includes [6]:

1. Per Base Sequence Quality - checks the range of quality values across all bases at each position in the FastQ file. In this box-whisker type plot, the elements are as follows:

    - Yellow box represents the inter-quartile range (25 - 75%)
    - Upper and lower whiskers represent the 10% and 90% points
    - Central red line is the median value
    - Blue line represents the mean quality

2. Per Sequence Quality Score - reports back if a subset of the sequences has universally low quality values possibly from being poorly imaged (should be a small percentage of the total sequences, unless there is a systematic problem)

3. Per Base Sequence Content - ideally the four bases (A, C, G, T) should be evenly distributed across the read, this test checks for that, if one of the base reports higher concentration, it could mean a possible contamination or bias when the reads were being generated

4. Per Base GC Content - reports on the GC content of each base position in the file, where overrepresentation could indicate a contamination or bias in the generation of the reads

5. Per Sequence GC Content - GC content should be roughly normally distributed across the sequences, where an unusual shaped distribution could indicate a contamination or bias in the generation of the reads

6. Per Base N Content - N is the default value the sequencer would produce if it couldn't determine with sufficient confidence what the base was

7. Sequence Length Distribution - checks to confirm if the length of sequences is uniform or not

8. Duplicate Sequences - this test checks to see if any sequence has an unusual concentration in the file, while it is possible to have some sequences duplicate, however unlikely for it to happen in high concentration

9. Overrepresented Sequences - typically the reads should be a diverse set wherein no individual sequence should make up a tiny fraction of the whole. Finding such a sequence(s) is indicative of - highly biologically significance, contamination, or the reads are not diverse as expected

10. Overrepresented K-mers - this test counts the enrichment of every 5-mer within the sequences. It compares it against an expected level at which the k-mer should have been based on the base content of the library as a whole and then uses the actual count to calculate an observed/expected ratio for that k-mer

2.4 Trimmomatic

When preparing DNA fragments for sequencing, one of the task is to orient adapters around them before, during and after enrichment of DNA fragments with PCR primer. Polymerase chain reaction

(PCR) is a technique used to amplify a single copy or a few copies of a DNA fragment across several orders of magnitude, generating thousands of copies of that fragment. This allows the sequencer to generate multiple copies of the fragments.

An example of an adapter is the TruSeq Universal Adapter: "5 AATGATACGGCGACCAC-CGAGATCTACACTCTTTCCCTACACGACGCTCTTCCGATCT 3", that of a primer is PCR Primer 2.0: "5 CAAGCAGAAGACGGCATACGAGAT 3". These help the sequencer to generate reads around the DNA fragments of interest [16]. These however are irrelevant information after the sequencing is done and could mislead FastQC's report on the quality of the reads, since the FASTA file would contain segments of these adapters and primers.

Trimmomatic removes the adapters and primers from the reads, and trims off bad reads or those not meeting the criteria using the following steps [9]:

- Cut adapter and other Illumina-specific sequences from the reads
- Perform a sliding window trimming, cutting once the average quality within the window falls below a threshold
- Cut bases off the start of a read, if below a threshold quality
- Cut bases off the end of a read, if below a threshold quality
- Cut the read to a specified length
- Cut the specified number of bases from the start of the read
- Drop the read if it is below a specified length

Trimmomatic has two modes of execution: Simple Mode and Palindrome Mode. Simple mode references to a single read file, whereas Palindrome mode references to reads that have a forward and reverse read files, such as our Sea Cucumber data [10]. Trimming out the adapters and primers from the reads along with bad reads based on threshold quality, length of reads and other factors, you end up with a subset of reads that are of higher quality such that you can focus on assembling.

Going back to the jigsaw example, Trimmomatic could be considered as a filtering process to weed out jigsaw pieces that do not meet the standard jigsaw puzzle criteria or are too blurry to make out the image on the piece.

Once the reads have been trimmed and cleaned up, the next step is to construct the DNA from these reads. As mentioned previously, we are interested in the De novo assembly of sea cucumber.

This assembly must align and merge the reads with overlap to form contigs. These contigs are then combined to form scaffolds. Many techniques have been developed to carry out the task of assembly. In general, these techniques involve traversal of overlap graphs that are built from the reads. The most widely used methods of assembly are OLC, De Bruijn, and SGA. The details of these methods are beyond the scope of this thesis. The following figure gives a pictorial representation of these techniques:

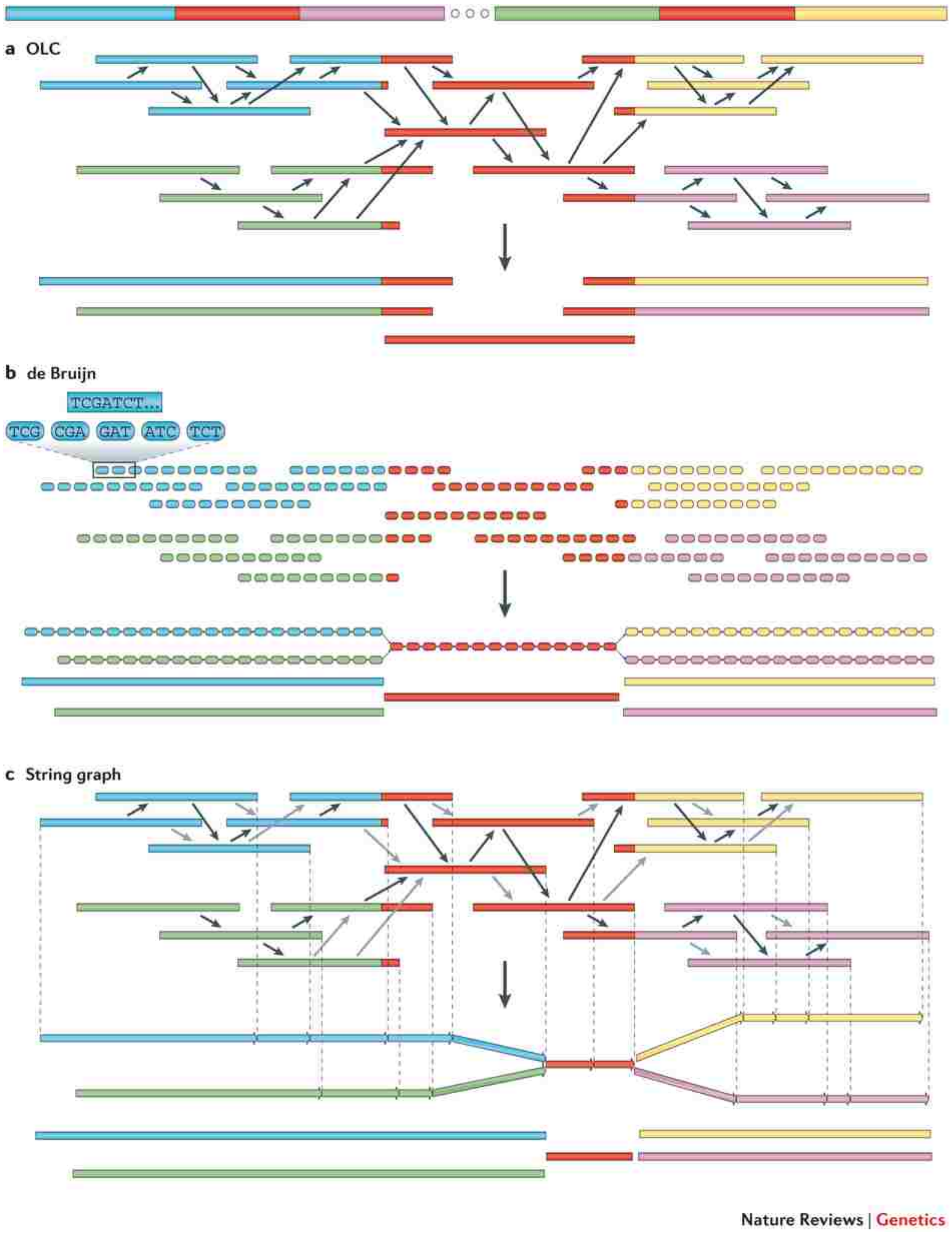


Figure 2.1: (Continued on the following page.)

Figure 2.1: “A genome schematic is shown at the top with four unique regions (blue, violet, green and yellow) and two copies of a repeated region (red). Three different strategies for genome assembly are outlined below this schematic. **a)** Overlap-layout-consensus (OLC). All pairwise alignments (arrows) between reads (solid bars) are detected. Reads are merged into contigs (below the vertical arrow) until a read at a repeat boundary (split colour bar) is detected, leading to a repeat that is unresolved and collapsed into a single copy. **b)** de Bruijn assembly. Reads are decomposed into overlapping k-mers. An example of the decomposition for k = 3 nucleotides is shown, although in practice k ranges between 31 and 200 nucleotides. Identical k-mers are merged and connected by an edge when appearing adjacently in reads. Contigs are formed by merging chains of k-mers until repeat boundaries are reached. If a k-mer appears in multiple positions (red segment) in the genome, it will fragment assemblies and additional graph operations must be applied to resolve such small repeats. The k-mer approach is ideal for short-read data generated by massively parallel sequencing (MPS). **c)** String graph. Alignments that may be transitively inferred from all pairwise alignments are removed (grey arrows). A graph is created with a vertex for the endpoint of every read. Edges are created both for each unaligned interval of a read and for each remaining pairwise overlap. Vertices connect edges that correspond to the reads that overlap. When there is allelic variation, alternative paths in the graph are formed. Not shown, but common to all three algorithms, is the use of read pairs to produce the final assembly product [12].”

In our experiments with sea cucumber, we used SGA followed by two post processing techniques of SSPACE and GapFiller. In the next section, we give a short summary of these techniques.

2.5 SGA

String Graph Assembler (SGA) is a *de novo* assembler designed to assemble large genomes from high coverage short read data [17]. SGA implements a set of assembly algorithms based on the FM-index. An SGA assembly has three distinct phases [18]:

1. First phase corrects base calling errors in the reads. An FM-index of the sequence reads is constructed; then base calling errors are identified by finding low-frequency k-mers in the reads.

2. Second phase assembles contigs from the corrected reads. An FM-index of the corrected sequence reads is constructed, where duplicate reads and low-quality reads after corrections are found and discarded with *sga filter* subprogram, while *sga fm-merge* merges together reads that can unambiguously assembled. Finally, *sga overlap* computes the structure of the string graph and contigs are built using *sga assemble.*

3. Third phase uses paired-end and/or mate pair data to build scaffolds from the contigs. This phase begins by re-aligning reads to the contigs built in the previous step. The copy number of each contig, and distances between contigs, are estimated from the resulting BAM files and used as input for *sga scaffold*, and the output from it is passed to *sga scaffold2fasta* to produce the FASTA file of the resulting scaffold sequences.

With the assembled genomes, there are a few other programs such as SSPACE and GapFiller that could be ran to further combine the scaffolds and fill some of the gaps in between them.

2.6 SSPACE

SSPACE (SSAKE-based Scaffolding of Pre-Assembled Contigs after Extension) is a stand-alone program for scaffolding pre-assembled contigs using NGS (Next-Generation Sequencing) paired-read data. SSPACE takes in the contigs assembled along with the paired-end data after non-ACTG reads have filtered and proceeds to loop through the following [7]:

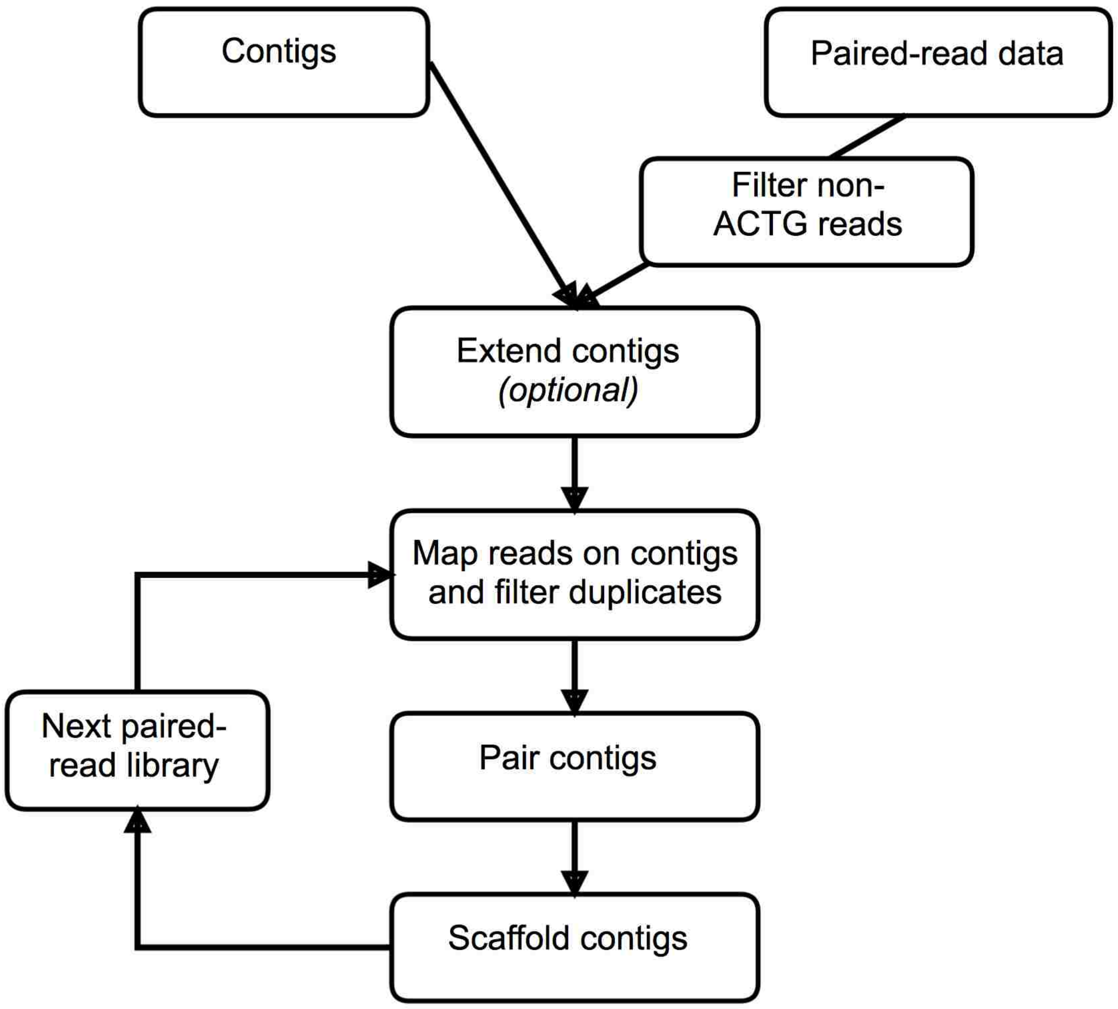


Figure 2.2: A schematic overview of the main steps of the SSPACE algorithm [7].

- The position and orientation of each pair that could be mapped is stored in a hash and duplicate read-pairs are filtered out.

- Putative contig pairs are computed based on the position of the paired reads on different contigs. These pairs are only considered if they satisfy the distance criteria set by the user-defined distance range.

- Scaffolds are formed iteratively combining contigs if a minimum number of read pairs support the connection, starting with the largest contig. The code also checks to see if there are alternative connections between the contigs and if found between the alternatives themselves, the algorithm seeks to place all alternatives in the correct order using the estimated insertion. Otherwise if the ratio that is calculated between the two best alternatives is below a threshold, a connection with the best scoring alternative is established. Extension of scaffolds is aborted if either a contig has no links with other contigs or the ratio for alternatives is exceeded.

- The scaffolding process is repeated until all contigs are incorporated into linear scaffolds.

2.7 GapFiller

GapFiller is used to close gaps within pre-assembled scaffold. "GapFiller seeks to find read pairs of which one member matches within a sequence region and the second member falls (partially) within the gap. The latter reads are then used to close the gap through sequence (k-mer) overlap. Gaps are entirely closed only if the size of the sequence insertion corresponds closely to the estimated gap size after scaffolding, which is based on the alignment of paired reads to the contigs. The process is iteratively repeated until no further gaps can be closed [8]."

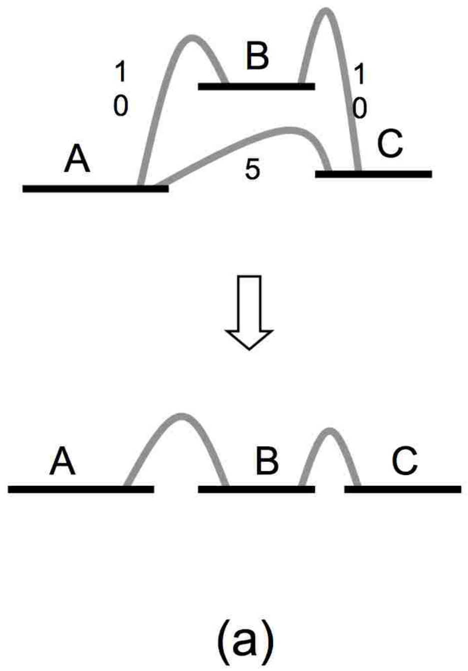


Figure 2.3: “Scaffolding contigs with alternative connections. SSPACE provides two scenarios for scaffolding contigs with alternative connections. In scenario **(a)** contig A has links with contigs B and C. Since also internal links exist between B and C, and the calculated distances between all contigs satisfy the distance-range, A, B and C are placed in one scaffold. In scenario **(b)** contig A has links with contigs B and C, but no internal links exist between B and C. Hence a ratio is calculated by dividing the number of links found between A and C (5) with those found between A and B (10). If this ratio (0.5) is below a user-specified treshold (parameter -a) contig A and B are placed in one scaffold [7].”

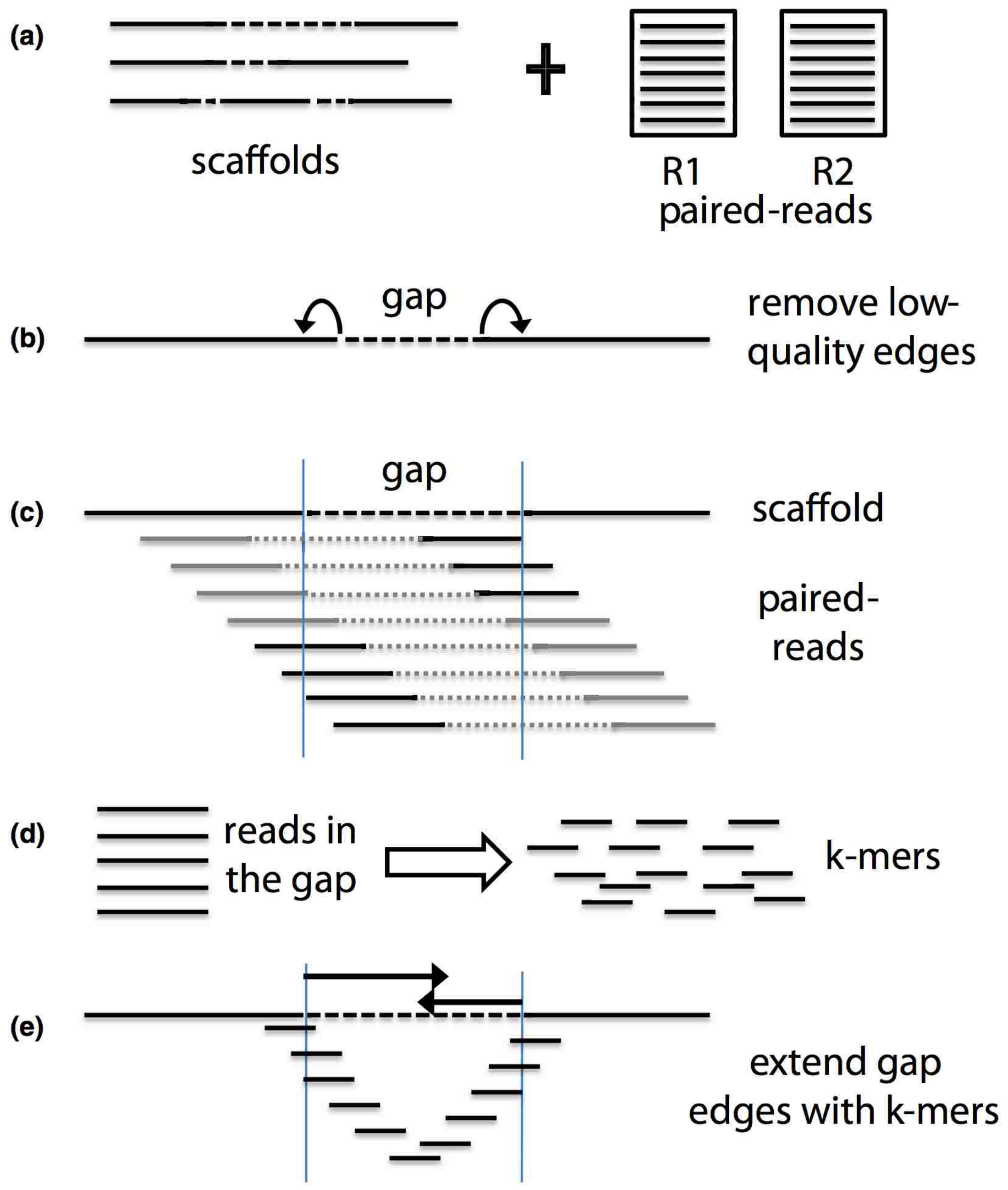


Figure 2.4: A schematic overview of the main steps of the GapFiller algorithm. “**(a)** The input data consist of a set of scaffold sequences containing gapped nucleotides and one or more sets of paired-end and/or mate-pair reads. **(c)** As a pre-processing step low quality nucleotides are removed from the sequence edges, thus enlarging the gap of ten nucleotides from each side. It should be stressed that the contig ends resulting from a draft assembly often contain misassemblies. **(c)** Paired-reads are aligned to the scaffolds and retained if one pair aligns to a scaffold sequence (dark grey) and one pair to a gapped region (black). **(d)** All pairs that are estimated to fall in the gapped regions are split into k-mers and used for gap filling. **(e)** The gap is closed from each edge by using k-mers that present a sequence overlap of size (k-mer - 1) and one nucleotide overhang. Gaps are closed if the right and left extensions can be merged and correspond to the estimated sequence gap [8].”

Chapter 3

Minhash

Suppose we have two sets, $A : \{a, b, c, d\}$ and $B : \{a, b, d, e\}$, and we want to find the similarity between the two sets. This can be achieved by taking the intersection between the sets over the size of the sets combined, in this case 3/5. This similarity is also known as Jaccard Similarity, also could be interpreted as *resemblance* $r(A, B)$, which is calculated for sets A and B by:

$$\frac{|A \cap B|}{|A \cup B|} \tag{3.1}$$

This fraction is a value that ranges between 0 and 1, where closer to 1 means that the sets are roughly the same.

Jaccard Distance $d(A, B)$, for sets A and B is defined as one minus Jaccard Similarity.

$$1 - \frac{|A \cap B|}{|A \cup B|} \tag{3.2}$$

This value also ranges from 0 to 1, where 0 means that the sets are identical. Jaccard Distance is a `metric measure` which satisfy the following four axioms [14]:

1. $d(x, y) \geq 0$ - no negative distance
2. $d(x, y) = 0$ if and only if $x = y$ - distance is positive, except for the distance from a point to itself
3. $d(x, y) = d(y, x)$ - distance is symmetric
4. $d(x, y) \leq d(x, z) + d(z, y)$ - triangle inequality

Other examples of metric measure are Euclidean distance, edit distance, Hamming distance, and cosine distance [14].

"How similar are these documents?" is a common question asked, whether be it in academia when checking for plagiarism or in business when checking for differences between 100 pages of legal documents or in search engines when trying to label how similar two webpages are. One could always compare each word in the document against each word in the other documents to compute the similarity between the documents. However, this is not computationally efficient. For example, to find similarity between a million documents one would have to do $\binom{1000000}{2}$ document comparisons which is $\approx \frac{n(n-1)}{2}$ which is on the order of n^2. This is where Minhash comes in. Minhash is a locality sensitive hashing scheme invented by Andrei Broder back in 1997 as part of the AltaVista search engine to find similar documents and compute similarity between documents [11]. The simplest representation of a document is a "bag of words" or `shingles`.

3.1 Shingles

The process of converting documents to sets is called Shingling. k-shingle or k-gram or k-mer is a sequence of k tokens that appears in a string. These tokens can be characters, words, sentences or something else depending on the application. For example, token when comparing sentences can be characters or words, whereas when comparing emails or documents could even be sentences. In our application, tokens will be strings of characters. For example, in the document, *to be or not to be*, the set of shingles can be {to be, be or, or not, not to, to be} or {to be, be or, or not, not to} after omitting duplicates. We can observe that the number of shingles of size k for a document of length n is $n - k + 1$. Of course, these shingles are not unique. Documents that have lots of shingles in common have similar text, even if the text appears in different order, so k must be large enough, or most documents will have most shingles [14]. The choice of k depends on the application. For example, for emails similarity $k = 5$ is better, while $k = 10$ is better for similarity of long documents. For our assembly application, we choose shingles of length 16.

Part of Minhash process is to compress long shingles so that it takes (say) 4 bytes. This allows us to represent a document by the set of hash values of its k-shingles. The following Python code shows how compressed k-shingles generated from a document:

```python
import binascii

# Generate k-sized shingles from document doc
# k    - shingle length
# doc - document
def genKShingles(k, doc):
    lenDoc = len(doc)
    shingles = []

    # Iterate from 0 to length of doc - k + 1
    for i in xrange(lenDoc-k+1):
        shingles.append(binascii.crc32(doc[i:i+k]) & 0xffffffff) # Get the 32-
            bit integer hash for the k-mer

    return shingles
```

Listing 3.1: Python code to generate k-mers

3.2 Matrix Representation of Sets

Once we have shingles generated from all the documents, one way to compare the documents against each other is to build a matrix with shingles as your rows and documents as your columns. This allows one to see which documents have the same shingles. Also, keep in mind, this matrix will be a sparse matrix.

For example, assume the universal set of our shingles is $\{a, b, c, d, e\}$ and our documents are $A = \{b, d, e\}$, $B = \{a, b, d\}$, $C = \{c\}$, $D = \{a, c, e\}$ and $E = \{d, e\}$, then the matrix would be:

Shingle	A	B	C	D	E
a	0	1	0	1	0
b	1	1	0	0	0
c	0	0	1	1	0
d	1	1	0	0	1
e	1	0	0	1	1

Table 3.1: Matrix representation of the shingles and documents, where 0 means the document doesn't have the shingle and 1 means the document does.

This matrix is also called the *characteristic matrix* for the shingles and their documents [14]. While a sparse matrix is not the ideal data structure to store data in, it helps visualize the data.

3.3 Minhash

"To *minhash* a set represented by a column of the characteristic matrix, pick a permutation of the rows. The minhash value of any column is the number of the first row, in the permuted order, in which the column has a 1 [14]."

For example, the matrix shown above in table 3.1, say permute the rows to form *bdeac*, this defines a minhash function h that maps sets to rows.

Shingle	A	B	C	D	E
b	1	1	0	0	0
d	1	1	0	0	1
e	1	0	0	1	1
a	0	1	0	1	0
c	0	0	1	1	0

Table 3.2: Matrix formed by permuting rows of Matrix 3.1 to form *bdeac*.

To compute the minhash value of set A according to h, we must find the first row for set A that has 1, which is b. Thus, $h(A) = b$, $h(B) = b$, $h(C) = c$, $h(D) = e$, and $h(E) = d$.

3.3.1 Minhash and Jaccard Similiarity

"The probability that the minhash function for a random permutation of rows produces the same value for two sets equals the Jaccard similarity of those sets [14]." The values for rows for sets A and B can be broken down into 3 categories:

1. Rows that have 0 in both columns (L)
2. Rows that have 1 in both columns (M)
3. Rows that have 0 and 1 (N)

In a spare matrix, most of the rows would be of type L, however it is the ratio of type M and N that determines both Jaccard similarity of A and B, and the probability that $h(A) = h(B)$. Jaccard similarity of A and B can be calculated by $m/(m+n)$ where, m is rows of type M and n is the rows of type N. m is represents $A \cap B$, and $(m+n)$ represents $A \cup B$.

3.3.2 Minhash Signatures

A minhash signature for column representing set S is the vector $[h_1(S), h_2(S), h_3(S), \ldots, h_n(S)]$, where $h_1, h_2, h_3, \ldots, h_n$ is n random permutations of the rows of the matrix M. The signatures themselves are placed into a matrix, called a signature matrix, in which i^{th} column is replaced by the minhash signature for (the set of) the i^{th} column [14].

Note that the signature matrix has only n rows, but has the same number of columns as matrix M.

3.3.3 Computing Minhash Signatures

It is not a feasible task to permute the characteristic matrix, even picking random permutations of millions or billions of rows is time-consuming, and the necessary sorting of the rows would take even more time. Thus, permuted matrices like in table 3.2 are not implementable [14].

However, it is possible to simulate the effect of a random permutation by a random hash function that maps row numbers to as many buckets as there are rows [14]. A hash function that maps integers $0, 1, 2, \ldots, k-1$ to bucket numbers to 0 through $k-1$ typically will map some pairs of integers to the same bucket and leave other buckets unfilled [14]. However, the difference is unimportant as long as k is large and there are not too many collisions [14].

Therefore, instead of picking n random permutations of rows, we will pick n randomly chosen hash functions $h_1, h_2, h_3, \ldots, h_n$ on the rows. The signature matrix is constructed by considering each row in their given order [14].

Taking the matrix in table 3.1, we are going to apply two hash functions, $h_1(x) = x + 1 \bmod 5$ and $h_2(x) = 3x + 1 \bmod 5$, from Chapter 3 of Mining of Massive Datasets. We have also replaced the letters in the shingles columns to be integers from 0 to 4. The values of these two functions applied to the row numbers are given in the last two columns in the matrix below:

Row	A	B	C	D	E	$(x+1) \bmod 5$	$(3x+1) \bmod 5$
0	0	1	0	1	0	1	1
1	1	1	0	0	0	2	4
2	0	0	1	1	0	3	2
3	1	1	0	0	1	4	0
4	1	0	0	1	1	0	3

Table 3.3: Matrix with hash functions computed for the rows.

These hash functions are true permutations of the rows, which is only possible since there are five rows and five being a prime number. If two rows get the same hash, collisions will occur.

We can compute the minhash signature from table 3.3 by doing the following:

1. Compute $h_1(r), h_2(r), \ldots, h_n(r)$
2. For each column c do the following:
   (a) If c has 0 in row r, do nothing.
   (b) If c has 1 in row r, then for each $i = 1, 2, \ldots, n$ set $SIG(i, c)$ to the smaller of the current value of $SIG(i, c)$ and $h_i(r)$.

Build a signature matrix with all row values set to ∞:

hash	A	B	C	D	E
h_1	∞	∞	∞	∞	∞
h_2	∞	∞	∞	∞	∞

First we consider row 0 in table 3.3, where $h_1(0) = 1$ and $h_2(0) = 1$. The row numbered 0 has 1's in the columns for sets B and D, so only these columns of the matrix can change. As 1 is less than ∞, the values in columns B and D can be updated:

hash	A	B	C	D	E
h_1	∞	1	∞	1	∞
h_2	∞	1	∞	1	∞

For row numbered 1, $h_1(1) = 2$ and $h_2(1) = 4$, we can set $SIG(1,1)$ to 2 and $SIG(2,1)$ to 4 for column A, but since column B has 1 which is less than 2 ($SIG(1,2)$) and 4 ($SIG(2,2)$), we would not be updating it.

hash	A	B	C	D	E
h_1	2	1	∞	1	∞
h_2	4	1	∞	1	∞

For row numbered 2, $h_1(2) = 3$ and $h_2(2) = 2$, we can set $SIG(1,2)$ to 3 and $SIG(2,2)$ to 2 for column C, but since column D has 1 which is less than 3 ($SIG(1,3)$) and 2 ($SIG(2,3)$), we would not be updating it.

hash	A	B	C	D	E
h_1	2	1	3	1	∞
h_2	4	1	2	1	∞

For row numbered 3, $h_1(3) = 4$ and $h_2(3) = 0$, we can $SIG(1,3)$ to 4 and $SIG(2,3)$ to 0 for column E. We can also update column A and B for $h_2(3)$ since 0 is less than 1.

hash	A	B	C	D	E
h_1	2	1	3	1	4
h_2	0	0	2	1	0

Finally, for row numbered 4, $h_1(4) = 0$ and $h_2(4) = 3$, column A, D and E will have 0 for $h_1(4)$ since 0 is less than their respective values - 2, 1, and 4. Therefore our final signature matrix is:

hash	A	B	C	D	E
h_1	0	1	3	0	0
h_2	0	0	2	1	0

Based off the final signature matrix, we can estimate the Jaccard similarities of the underlying sets. For example, column A and E are identical, so we can guess that the $SIM(A, E) = 1$, but if we look at table 3.3, we can see that the true Jaccard similarity of A and E is 2/3. For additional similarities:

x	y	$SIM(x, y)$	JS
A	B	0.5	0.5
A	C	0	0
A	D	0.5	0.25
B	C	0	0
B	D	0	0.25
B	E	0.5	0.25
C	D	0	1/3
C	E	0	0
D	E	0.5	0.25

As shown, the fraction of rows that agree in the signature matrix is only an estimate of the true Jaccard similarity. Remember, this example is much too small for the law of large numbers to assure that the estimates are close [14].

3.3.4 False Positives & False Negatives

Hashing can be compared to bucketing items into individual buckets based on their similarities. For example, geometric objects can be compared according to dimensions and shape. If not diverse enough, collisions can occur where a circle and sphere would end up in the same bucket, because the two are the same dimensionally. This can be remedied by picking numbers that are larger than the population size, ideally a prime number.

In addition, we could drop squares and rectangles together into the same bucket as well, this is known as *false positive.* Whereas *false negative* occurs when squares are not bucketed in the same buckets. By using a large number of hash functions when generating the minhash signature we can hope to reduce the chances of dissimilar items being in the same bucket and increase the chances of similar items being in the same buckets, and hope only a small fraction end up being false positives and false negatives.

3.4 Locality-Sensitive Hashing

In order to compare a million documents, say for example, as computed early on in this chapter, we would have to do roughly half a trillion comparisons. This is only if our goal is to find the similarity between every pair of documents, however most of the time we only want the all pairs that are above a certain threshold of similarity. We can shift our focus to such pairs by performing *locality-sensitive hashing* (LSH) or *near-neighbor search* [14].

3.4.1 Banding

With the minhash signatures we've created, an effective way to pick the hashings is to divide the signature matrix into b bands consisting of r rows each. For each band, there is a hash function that takes vectors of r integers (the portion of one column within the band) and hashes them to some large number of buckets. The same hash function is used for all the bands, but a separate bucket array for each band, so that columns with same vector in different bands are not hashed to the same bucket [14].

		1 0 0 0 2	
b_1	...	3 2 1 2 2	...
		0 1 3 1 1	
b_2			
b_3			

Table 3.4: Dividing a signature matrix into four bands of three rows per band [14].

The probability that minhash signatures for any two documents to agree in any one particular row of the signature matrix is s. Then we can calculate the probability that the signatures of these documents of becoming a candidate pairs as follows [14]:

1. The probability that the signatures agree in all rows of one particular band is s^r.
2. The probability that the signatures disagree in at least one row of a particular band is $1 - s^r$.
3. The probability that the signatures disagree in at least one row of each of the bands is $(1-s^r)^b$.
4. The probability that the signatures agree in all the rows of at least one band, and therefore become a candidate pair, is $1 - (1 - s^r)^b$.

The function $1 - (1 - s^r)^b$ forms a *S-curve* given b and r. The steepest part of the curve is where our threshold for finding candidates is, the value of s at which the probability of becoming a candidate is 0.5. The pairs with similarity above the threshold are very likely to become candidates, while those below the threshold are unlikely to become candidates [14].

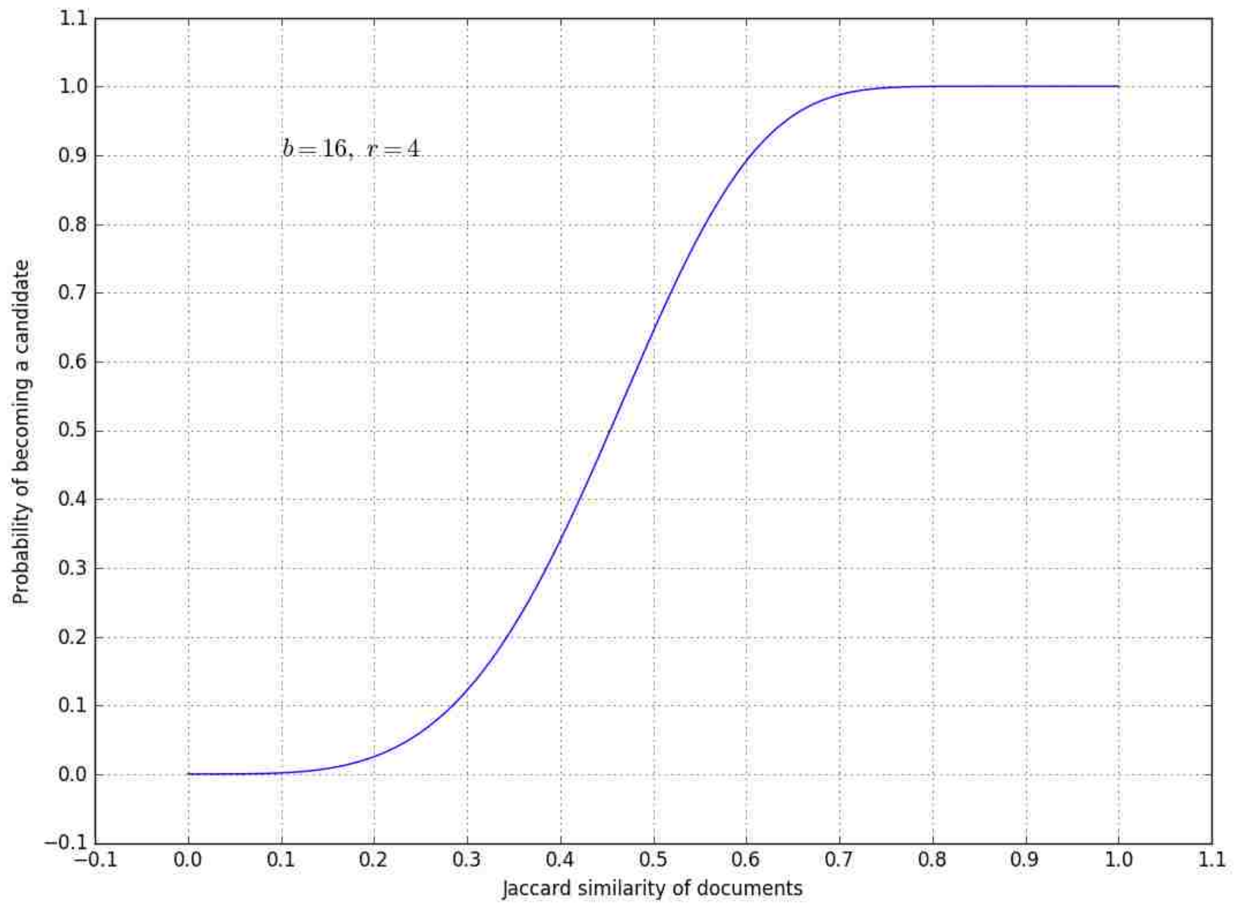


Figure 3.1: For $b = 16$ and $r = 4$, this is the curve generated by $1-(1-s^r)^b$. The threshold is ≈ 0.45, which can also be approximated by taking $(1/b)^{(1/r)} = (1/16)^{(1/4)} = 1/2$.

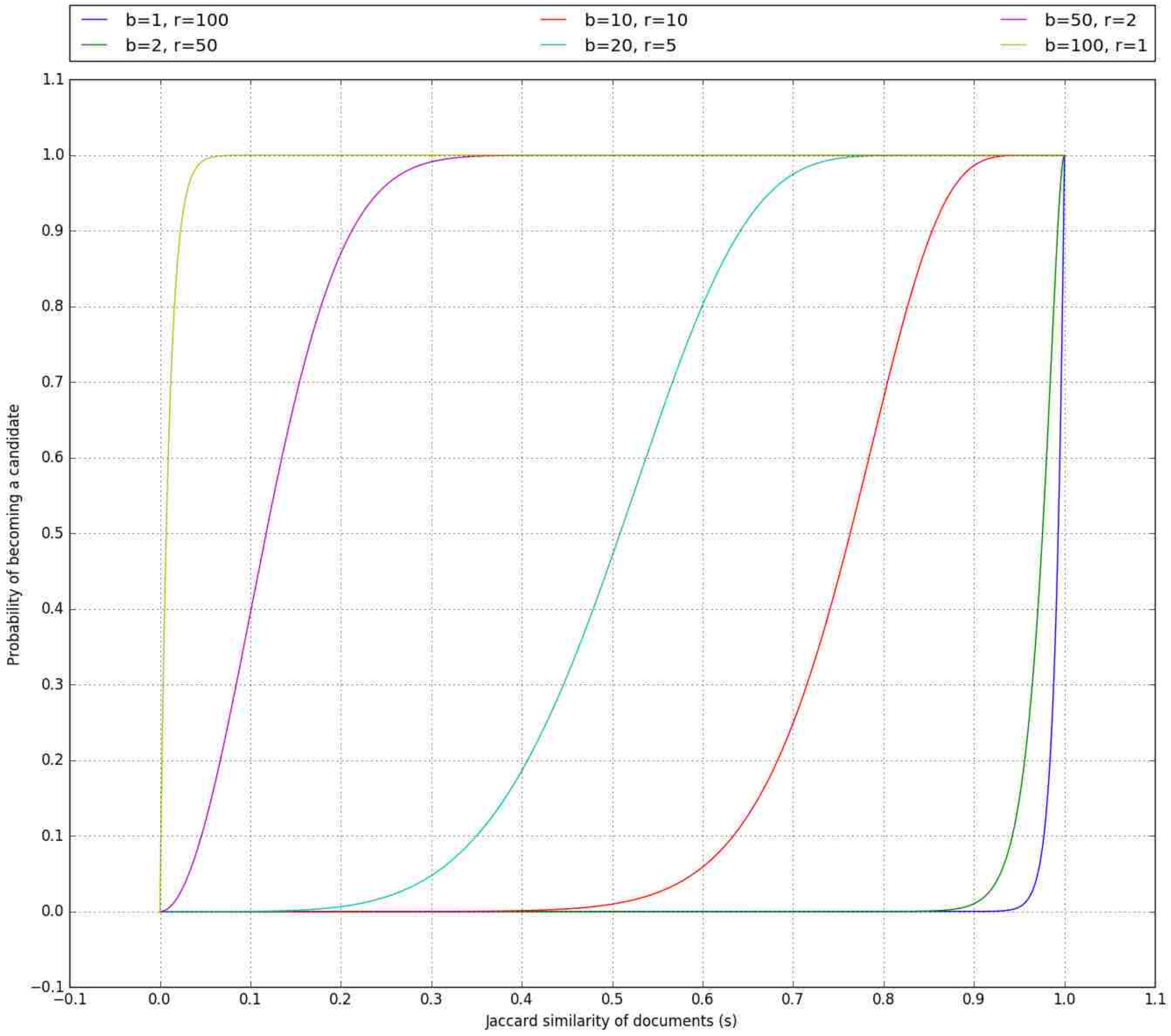


Figure 3.2: Given 100 rows, we can band them 6 different ways. Here we have plotted the various combinations to show how the curve reacts to the values of b and r picked. The area of the graph to the left of your selected s would be your false positives, while the area to the right would be your false negatives.

Consider $s = 0.8$, for $b = 20, r = 5$ for d_1 and d_2:

- Probability d_1, d_2 are identical in one particular band is: $(0.8)^5 = 0.32768$
- Probability d_1, d_2 are not similar in all of the 20 bands: $(1 - 0.32768)^{20} = 0.000356$
- Probability d_1, d_2 are identical in all rows of at least one of the 20 bands, and become a candidate pair is: $1 - 0.000356 = 0.999644$.
  - About one in 3000 pairs that are as high as 80% are false negatives (we will miss them)
  - We would find 99.9644% pairs of truly similar documents

3.5 Summary

We can summarize minhash with locality-sensitivity hashing procedure as follows:

1. We pick a k value for generating k-mers from each document depending on the content type
2. We pick n hash functions to hash the k-mers through to form the minhash signature for each document
3. We then pick a threshold s that defines how similar documents have to be in order for them to be a candidate pair. Next we pick a number of bands b and a number of rows r such that $b * r = n$, and the threshold s is approximately $(1/b)^{1/r}$ [14]:
   - If avoidance of false negatives is important, then $(1/b)^{1/r}$ must be lower than s
   - If speed is important and avoidance of false positives is important, then $(1/b)^{1/r}$ must be higher than s
4. Retrieve candidate pairs using locality-sensitive hashing technique
5. Examine the candidate pairs' signatures before examining the documents themselves

Chapter 4

Experiment

As previously mentioned in chapter 2, we are working with Sea Cucumber paired-end reads prepared at University of Southern California.

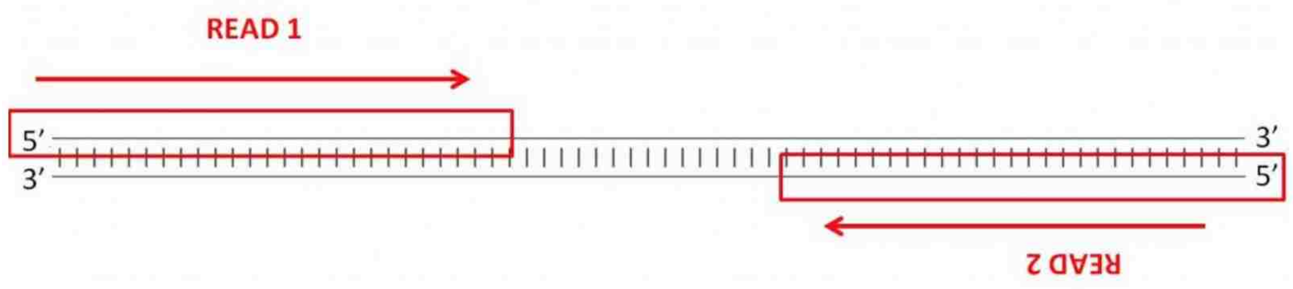
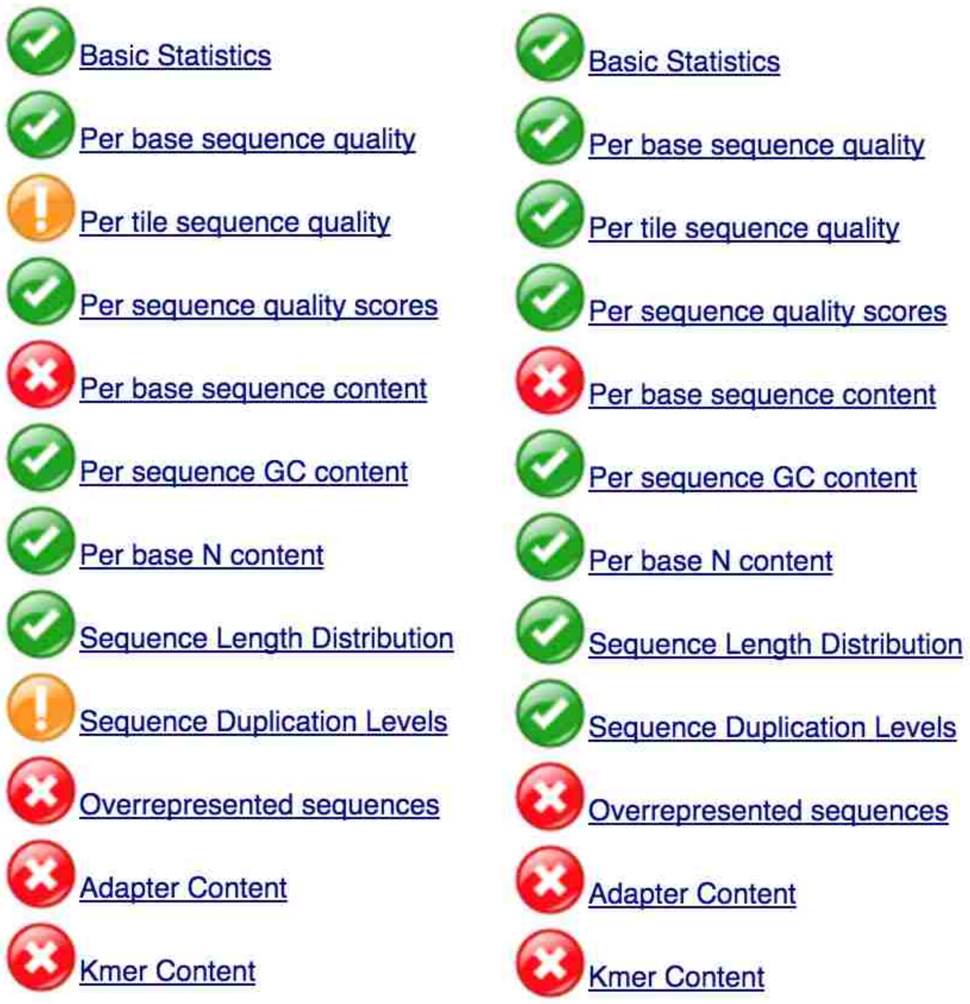


Figure 4.1: A pair of reads, the sequence you see are, conceptually, *pointing towards each other on opposite strands*. When you align them on the genome, one read should align to the foward strand, and the other should align to the reverse strand, at a higher base pair position than the first one so that they are pointed towards one another. This is known as a "FR" read - forward/reverse, in that order [15].

The reads consists of 414,981,154 forward (LP001_R1.fastq) and reverse (LP001_R2.fastq) reads. Our first task was to verify the quality of the reads using FastQC.



✓ Basic Statistics
✓ Per base sequence quality
! Per tile sequence quality
✓ Per sequence quality scores
✗ Per base sequence content
✓ Per sequence GC content
✓ Per base N content
✓ Sequence Length Distribution
! Sequence Duplication Levels
✗ Overrepresented sequences
✗ Adapter Content
✗ Kmer Content

(a) LP001_R1

✓ Basic Statistics
✓ Per base sequence quality
✓ Per tile sequence quality
✓ Per sequence quality scores
✗ Per base sequence content
✓ Per sequence GC content
✓ Per base N content
✓ Sequence Length Distribution
✓ Sequence Duplication Levels
✗ Overrepresented sequences
✗ Adapter Content
✗ Kmer Content

(b) LP001_R2

Figure 4.2: FastQC report on our Sea Cucumber reads.

As shown, FastQC has flagged the reads with few alerts and warnings. In order to fix some of the warnings, we ran the reads through Trimmomatic (paired-end mode), with the following criteria:

- Remove adapters (ILLUMINACLIP:adapter.fa:2:30:10)
- Remove leading low quality or N bases (below quality 3) (LEADING:3)
- Remove trailing low quality or N bases (below quality 3) (TRAILING:3)
- Scan the read with a 4-base wide sliding window, cutting when the average quality per base drops below 15 (SLIDINGWINDOW:4:15)
- Drop reads below the 151 bases long (MINLEN:151)

```
java -jar ../programs/Trimmomatic-0.35/trimmomatic-0.35.jar PE -trimlog
    trimlog.txt LP001_R1.fastq LP001_R2.fastq LP001_forward_paired.fastq
    LP001_forward_unpaired.fastq LP001_reverse_paired.fastq
    LP001_reverse_unpaired.fastq ILLUMINACLIP:adapter.fa:2:30:10 LEADING:3
    TRAILING:3 SLIDINGWINDOW:4:15 MINLEN:151
```

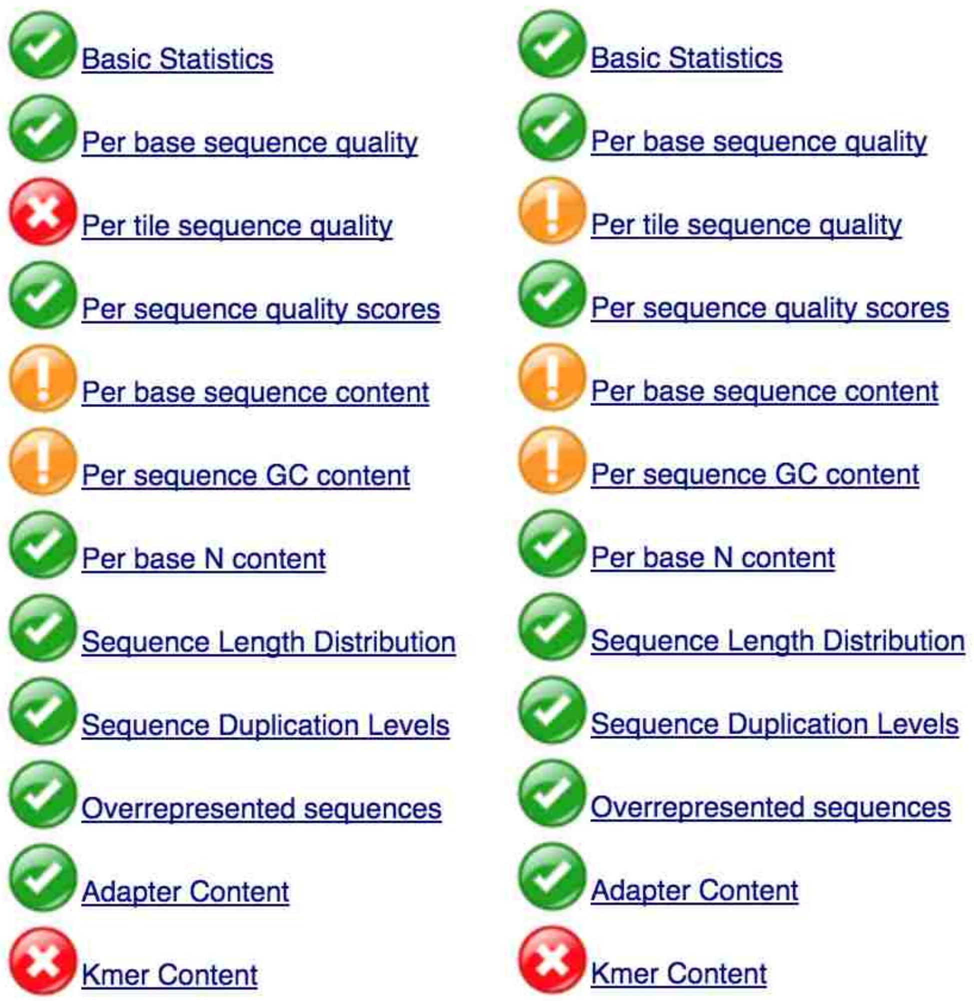
Listing 4.1: Running trimmomatic pair-end mode

```
>TruSeq3_IndexedAdapter
GATCGGAAGAGCACACGTCTGAACTCCAGTC
>TruSeq3_UniversalAdapter
ACACTCTTTCCCTACACGACGCTCTTCCGATCT
```

Listing 4.2: adapter.fa file used for Trimmomatic

Trimmomatic for a paired-end mode run, outputs four files - forward and reverse, paired and unpaired reads. The paired reads are the ones that met the criteria we set for the run, while the unpaired ones did not make the cut. Number of paired reads that survived is 184,016,240.

The following is the FastQC's report on the result from Trimmomatic:



✓ Basic Statistics	✓ Basic Statistics
✓ Per base sequence quality	✓ Per base sequence quality
✗ Per tile sequence quality	! Per tile sequence quality
✓ Per sequence quality scores	✓ Per sequence quality scores
! Per base sequence content	! Per base sequence content
! Per sequence GC content	! Per sequence GC content
✓ Per base N content	✓ Per base N content
✓ Sequence Length Distribution	✓ Sequence Length Distribution
✓ Sequence Duplication Levels	✓ Sequence Duplication Levels
✓ Overrepresented sequences	✓ Overrepresented sequences
✓ Adapter Content	✓ Adapter Content
✗ Kmer Content	✗ Kmer Content

(a) LP001_forward_paired (b) LP001_reverse_paired

Figure 4.3: FastQC report on the forward and reverse paired reads outputed from Trimmomatic.

Next we fed the survived reads to SGA to build a draft assembly. SGA produced:

Scaffolds merged	434,385
Unplaced Scaffolds	1,671,716
Total Scaffolds	2,106,101

a total of 2,106,101 scaffolds, out of which 1,671,716 were unplaced scaffolds. These scaffolds were then passed through SSPACE to combine some of the scaffolds. This resulted in a total of 2,073,163 scaffolds. These scaffolds are then passed through GapFiller to finally form 2,072,983 scaffolds. These scaffolds will be used in our implementation of Minhash where we will be using Sea Urchin's scaffolds as a reference genome.

The California Purple Sea Urchin (*Strongylocentrotus purpuratus*) has been sequenced and annotated by the Sea Urchin Genome Sequencing Consortium led by the Human Genome Sequencing Center (HGSC) at Baylor College of Medicine [2]. We will be using *Spur_4.2* from https://www.ncbi.nlm.nih.gov/assembly/GCF_000002235.4/ as our reference genome to compare against our Sea Cucumber's scaffolds.

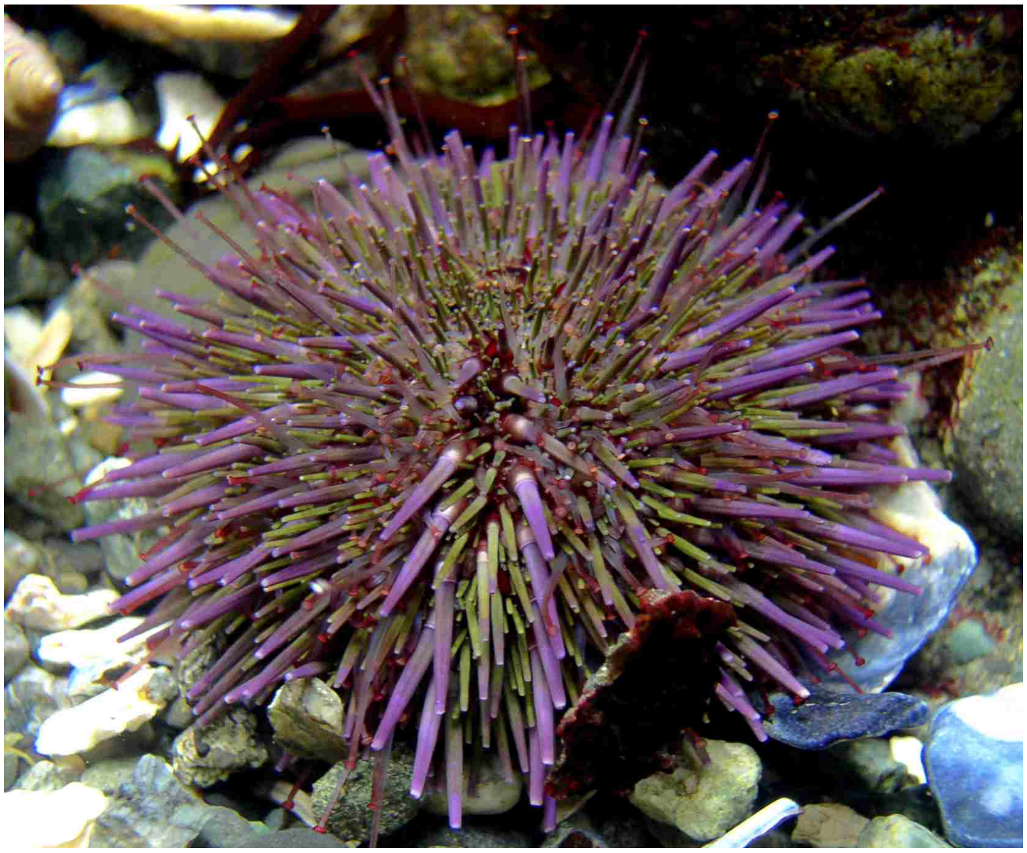


Figure 4.4: The California Purple Sea Urchin aka *Strongylocentrotus purpuratus*. Courtesy of Kirt L. Onthank.

Sea cucumber and sea urchin are Echinoderms, a member of the phylum Echinodermata of marine animals. Sea urchin being of the same phylum is what makes it a good candidate for a reference genome for sea cucumber. Sea urchins genome has been shown to be surprisingly related to humans' https://www.nsf.gov/news/news_summ.jsp?cntn_id=108174. One of the objectives of assembly is identification and annotation of genes. In the case of sea cucumber, researchers want to identify the gene or genes that are responsible for healing properties.

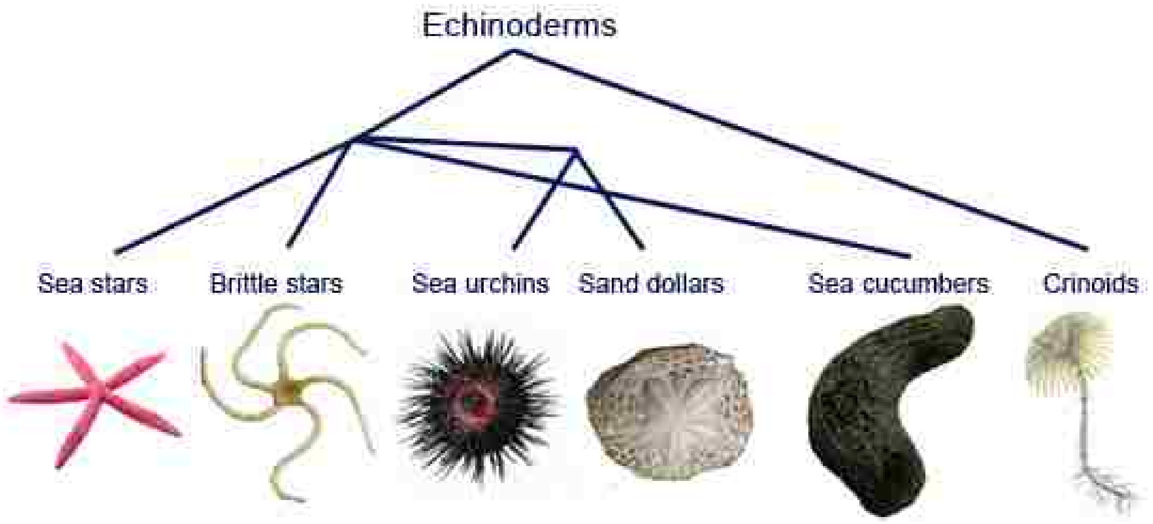


Figure 4.5: The different groups of echinoderms. Courtesy of MESA (http://www.mesa.edu.au/echinoderms/default.asp).

Chapter 5

Minhash Implemenation

Our implementation of minhash is broken into the following five components:

1. Reading in sea urchin and sea cucumber scaffolds
2. Generating k-mers from the scaffolds
3. Hashing the k-mers
4. Collecting the minhashes to form signatures
5. Finding similarity between sea urchin and sea cucumber signatures using Hamming distance

The rest of this chapter is divided into two sections. The first section gives an overview of our implementation. The second section provides the details on improving the efficiency of our implementation with concepts from parallel programming.

5.1 Implementation Overview

We used the sea urchin scaffolds (*Spur*_4.2) generated by HGSC at Baylor College of Medicine and the sea cucumber scaffolds from the output from GapFiller. The scaffold file is read in and k-mers (where $k = 16$) are generated using the code mentioned in chapter 3.1. The k-mers are stored as integers to save space. Sea urchin has 31,896 scaffolds which generated about 990,436,922 16-mers, while sea cucumber which has 2,072,983 scaffolds, generated 1,094,181,998 16-mers. These large numbers point to the difficulty of the comparison of sea cucumber and sea urchin scaffolds. The brute force technique requires a total of $2,072,983 * 31,896 = 66,119,865,768$ scaffold comparisons.

Once the k-mers are generated, our next piece of code generates a family of independent hash functions to map the k-mers to produce the minhash signature vectors for each of our scaffolds. The family of independent hash functions we generate are of the form $(ax+b)\%c$ where a and b are random values between 0 and $maxNum$ (in our case $maxNum$ is $2^{32}-1$) that only occurs once (this is guaranteed by Python's random.sample function), x is the k-mer in integer form and c is the prime larger than $maxNum$ so that we have fewer or no collisions happening. The following Python code shows how we generate our family of independent hash functions:

```
import random

# Generate a list of n unique random numbers
def genRandomNums(n):
    # Maximum number allowed is 2^32 - 1 cause of our hash limit
    maxNum = 2**32 - 1

    # Check if n is valid
    if n < 0 or n > maxNum:
        raise Exception("Invalid number of random number requested")

    # Use random's sample call to generate list of n unique random numbers
        between 0 and maxNum
    return random.sample(xrange(0, maxNum), n)

# Generates h independent hash functions of the format: h(x) = (a*x + b) % c
def genHashFuncs(h):
    # Set c to number larger than maxNum
    c = 4294967311

    # Generate 2 list of values for a and b
    aList = genRandomNums(h)
    bList = genRandomNums(h)

    # Initialize hash function array
    hashFuncs = []

    # Generate h random hash functions
    for i in xrange(h):
        hashFuncs.append(lambda x, a=aList[i], b=bList[i], c=c: (((a * x) + b)
            % c))

    return hashFuncs
```

Listing 5.1: Python code to generate a family of independent hash functions

In our experiments, we generate 200 independent hash functions. We then use these independent hash functions to generate the minhash signature for each scaffold by finding the lowest value computed when passing each k-mer through the independent hash functions. The following C code displays how the minhash signature is computed for each scaffold:

```
unsigned long int *computeMinhashSignature(unsigned long int *scaffold,
    unsigned long int scaffoldLength, struct hashFuncs hFuncs)
{
    unsigned long int i = 0, j = 0, hash = 0;
    unsigned long int *vec = (unsigned long int *)malloc(sizeof(unsigned long
        int) * hFuncs.length);

    for (i=0; i<hFuncs.length; ++i)
    {
        vec[i] = hFuncs.c;
        for (j=0; j<scaffoldLength; ++j)
        {
            // (((a * x) + b) % c)
            hash = (((hFuncs.funcs[i].a * scaffold[j]) + hFuncs.funcs[i].b) %
                hFuncs.c);

            if (vec[i] > hash)
            {
                vec[i] = hash;
            }
        }
    }
    return vec;
}
```

Listing 5.2: C code to compute the minhash signature from a scaffold of given length and a family of independent hash functions

Once the minhash signatures are computed for each scaffold, we can finally compare the scaffolds of sea cucumber and sea urchin against each other using Hamming distance.

Hamming distance is a distance measure metric that satisfies the axioms as explained in Chapter 3. Hamming distance between two strings of equal length is the number of positions at which the corresponding symbols are different. For example, the hamming distance between the strings *abbc* and *abcc* is 1, because the third position is different between the two strings.

In our case, the strings for Hamming distance would be the minhash signature for each scaffold. We calculate the Hamming distance a bit differently, where instead of counting the difference between the two signatures, we compute the similarity of them. The following C code shows how we calculate Hamming distance between two minhash signatures. It is worth mentioning that we divide the distance by the length of the signature array to normalize the value to fall between 0 and 1.

```c
double calHammingDistance(unsigned long int *a, unsigned long int *b, int
    length)
{
    double dist = 0;
    int i = 0;

    for (i=0; i<length; ++i)
    {
        if (a[i] == b[i])
        {
            ++dist;
        }
    }

    // divide by length to set the distance between 0 and 1
    return dist / length;
}
```

Listing 5.3: C code to compute the Hamming distance between two arrays of the same length

With our interpretation of Hamming distance, a distance value closer to 1 means the two scaffolds are very similar (1 being the same scaffold), while a value closer to 0 means they are very different.

5.2 Implementation Efficency

Given our data size, implementing certain sections sequentially is not ideal, while doable, the sheer amount of calculations involved would make the time taken impractical. In order to speed up the computation, we employed the use of not just MPI (Message Passing Interface) but also utilized Intel Xeon Phi Coprocessors hosted by Cherry Creek 2.0 supercomputer at Switch.

5.2.1 MPI & OpenMP

MPI in short is a message passing system that networks multiple machines for computation. This allows communications between the machines, therefore one can send data to and from machines or simply use them to work on the problem in parallel. We utilize MPI purely to launch our job across multiple machines. While we could utilize it to send data to the machines, given the nature of shared disk on Cherry-Creek this was not necessary.

OpenMP is a specification for a set of compiler directives, library routines, and environment variables that can be used to specify high-level parallelism in Fortran and C/C++ programs. OpenMP is used to spin up threads in parallel to parallelize the computations involved. We specifically used `#pragma omp for` call to generate n threads (where n is 220) when running `for` loops so that we could fully utilize the Intel Xeon Phi Coprocessor's computational resources. Threads unlike MPI processes, are isolated to the host machine, however unlike MPI processes that uses multiple machines and has distributed memory, threads share memory with other threads on the same machine. This can be both a boon and a bane, a boon in the aspect no special instructions or calls are required to access memory utilized by another thread, the bane being if one is not careful, this could introduce race conditions. Race condition happens when more than one thread contests for the same memory location at the same time with operations that could lead to incorrect result if not performed sequentially.

5.2.2 Intel® Xeon Phi™ Coprocessor

Coprocessor is a computer processor used to supplement the functions of the primary processor. In essence it "offloads" the work from the processor and speeds up the overall performance of the machine. "The original Cherry Creek (1.0) had 48 nodes. Each node had 2 Intel Xeon E5 - 2697v2 (12 cores each), 128Gb Ram, and 3 Intel Xeon Phi 7120P coprocessors (with 61 cores each). In addition to the 48 nodes above, Cherry Creek 2.0 has an additional: 48 Penguin Computing Relion nodes each with 2 Xeon E5 - 2640v3 (8 cores each), 128Gb Ram, and 4 Intel Xeon Phi 31S1P (with 57 cores each). It also contains 24 Intel manufactured nodes with 2 Xeon E5 - 2697v2 (12 cores each), 192Gb Ram, and 2 Intel Xeon Phi 7120P coprocessors (with 61 cores each). Cherry Creek 2.0 currently has the following capabilities: theoretical peak speed of 495 TFlops/s (Trillion Floating-Point operations per second), total Memory: 32.470 TB (TeraBytes), and total scratch storage of 46.32 TB" [3].

5.2.3 Using an Intel® Xeon Phi™ Coprocessor

Intel Xeon Phi coprocessor is essentially just like a regular processor, but it has its own set of rules and regulations on how to use it, and even its own Linux operating system. While typical contemporary processors clock about 2-3GHz, Intel Xeon Phi coprocessors clock about 1GHz [19]. Intel Xeon Phi coprocessors are best used as pure computational entities; they can handle anywhere from 220 to 240 threads in parallel while a regular processor is limited to 8 to 24 threads in parallel. They have their own memory (8 to 16GB GDDR5) that hosts the data used for computation, which is not much for large dataset problems and as such data will need to be transferred from host to coprocessor and vice-versa via the System BUS [19]. Intel Xeon Phi coprocessors can also be used as part of MPI jobs since they have their own hostname.

5.2.4 Transferring functions and data to an Intel® Xeon Phi™ Coprocessor

As mentioned in the above subsection 5.2.3 Intel Xeon Phi coprocessor has its own Linux operating system and supports traditional Linux services including SSH [19]. There are three ways to run code on an Intel Xeon Phi coprocessor:

- Native Applications - Directly compile and run code from the coprocessor without needing to involve the host processor.

```
#include <stdio.h>
#include <unistd.h>

int main()
{
    printf("Hello world! I have %ld logical cores.\n",
    sysconf(_SC_NPROCESSORS_ONLN));
}
```

Listing 5.4: C hello world code that can be ran on host as well as on Intel Xeon Phi Coprocessor [19].

```
user@host% icc hello.c
user@host% ./a.out
Hello world! I have 32 logical cores. user@host%
```

Listing 5.5: Compiling and running the "Hello World" code on the host. [19].

To compile the C code to run on Intel Xeon Phi coprocessor, `-mmic` flag must be used. Note that once compiled with this flag, host machine cannot execute the code.

```
user@host% icc hello.c -mmic
user@host% ./a.out
-bash: ./a.out: cannot execute binary file
user@host% scp a.out mic0:~/
a.out 100% 10KB 10.4KB/s 00:00
user@host% ssh mic0
user@mic0% pwd
/home/user
user@mic0% ls
a.out
user@host% ./a.out
Hello world! I have 240 logical cores.
```

Listing 5.6: A native application for Intel Xeon Phi coprocessors cannot be run on the host system, so we have to transfer and run a native application on an Intel Xeon Phi coprocessor direclty. [19].

- Explicit Offload Model - With this model, special instructions are wrapped around code and data that needs to be transferred from host to the coprocessor and back. These special instructions generate special code for the coprocessor to execute.

```
// __attribute__((target(mic))) is a declaration qualifier that indicates
    that the declared object (variable or function) must be compiled into
    the target code
__attribute__((target(mic))) int data[1000];

int __attribute__((target(mic))) CountNonzero(const int N, const int* arr
    )
{
    int nz=0;
    for (int i = 0; i < N; i++) if (arr[i] != 0) nz++;
    return nz;
}

// #pragma offload_attribute(push, target(mic)) and #pragma
    offload_attribute(pop) can be used instead of the qualifier
    __attribute__((target(mic))) when multiple consecutive elements in a
    source file need to be included in the offload code
#pragma offload_attribute(push, target(mic))

double* ptrdata; // Apply the offload qualifier to a pointer-based array
void MyFunction(); // a function
#include "myvariables.h" // or even a whole file

#pragma offload_attribute(pop)

// #pragma offload_transfer target(mic) requests that certain non-scalar
    data must be copied to the coprocessor. This pragma takes a number of
    clauses to specify data traffic
#pragma offload_transfer target (mic:0) in(ptrdata : length(N)) alloc_if
    (1) free_if(0)
```

Listing 5.7: Some of the special instructions used to generate code specifically for Intel Xeon Phi coprocessors [19].

- MYO (Virtual-Shared) Memory Model - An alternative to the offload model, a virtual-shared memory approach called MYO, Mine Yours Ours, eliminates the need for data marshaling. That is, `#pragma offload` is replaced with a software emulation of memory shared between multi-core and many-core processors residing in a single system. MYO refers to the software abstraction that shares memory within a system for determining current access and ownership privileges. This is controlled by two keywords `_Cilk_shared` and `_Cilk_offload` [19].

```
#include <stdio.h>

#define N 1000

_Cilk_shared int ar1[N];
_Cilk_shared int ar2[N];
_Cilk_shared int res[N];

void initialize()
{
    for (int i = 0; i < N; i++)
    {
        ar1[i] = i;
        ar2[i] = 1;
    }
}

_Cilk_shared void add()
{
    #ifdef __MIC__
        for (int i = 0; i < N; i++) res[i] = ar1[i] + ar2[i];
    #else
        printf("Offload to coprocessor failed!\n");
    #endif
}

void verify()
{
    bool errors = false;

    for (int i = 0; i < N; i++)
        errors |= (res[i] != (ar1[i] + ar2[i]));

    printf("%s\n", (errors ? "ERROR" : "CORRECT"));
}

int main(int argc, char *argv[])
{
    initialize();
    _Cilk_offload add(); // Function call on coprocessor, ar1, ar2 are
        copied in, res copied out
    verify();
}
```

Listing 5.8: Example of using the virtual-shared memory and offloading calculations with _Cilk_shared and _Cilk_offload of the function call. Note that, even though data are not explicitly passed from the host to the coprocessor, function compute_sum(), executed on the coprocessor, has access to data initialized on the host. [19].

5.2.5 Using Multiple Intel® Xeon Phi™ Coprocessor

Cherry-Creek has the Intel Xeon nodes configured to have anywhere from two to four Intel Xeon Phi coprocessors connected to it. Which allows a user to further distribute their data and spin up more threads in parallel to work on the data. Sending data to multiple coprocessors can be blocking or non-blocking or via MPI. We went with spinning up threads using `pragma` on the host machine equal to the number of coprocessors attached to the node to split up the data and communicate that to their respective coprocessor.

5.3 Summary

Our code can be summarized as follows (as of right now):

1. Generate a family of h independent hash functions
2. The scaffolds are read in by a Python script that then produces n output files evenly distributed with k-mers that are converted to integers 3.1

3. We then launch a C code that immediately launches n MPI-processes that does the following:
   (a) Read in the n^{th} file output file from the previous step
   (b) Read in the h independent hash functions
   (c) Split the scaffolds further based on c coprocessors allocated for the job
   (d) Offload the data to the coprocessor for computing the minhash signature for each scaffold using the independent hash functions 5.2
   (e) Retrieve the minhash signature from the coprocessor and write them into a file
4. Launch a second C code that then does the following:
   (a) Reads in the output from the previous step for sea cucumber and sea urchin
   (b) Compares the signatures to find similar scaffolds using Hamming distance 5.3
   (c) Write the result from comparisons - scaffold # of sea cucumber, scaffold # of sea urchin, number of matches

Chapter 6

Result

To test out the code, we first ran sea urchin against itself. A Python multi-process version (8 processes in parallel) took roughly 90 hours to compute the similarities between the sea urchin scaffolds. We then updated the code to have Python generate a file with the k-mers since it is easier to do string manipulation in it, and wrote a C OpenMP version (8 threads) which then took roughly 20 minutes to compute the similarities. Once we introduced a single coprocessor (220 threads), then time taken dropped down to 13 minutes. Total scaffolds compared in this scenario was $(31,896 * 31,896/2) = 508,677,408$.

We further modified the Python code to generate n k-mer files for sea cucumber, and updated our C code to now launch n MPI process (1 per node) which then utilizes 4 coprocessors on each node (each coprocessor spawning 220 threads), so essentially if $n = 4$, we will be spawning $4 * 4 * 220 = 3520$ threads. The execution time to compute the minhash signatures for all sea cucumber scaffolds for $n = 8$ was roughly 3 minutes. Finally, a C sequential code was written to compute the similarities between sea cucumber and sea urchin which took 31 hours.

About 5,855,468 single matches was found between sea cucumber and sea urchin scaffolds, followed by 249,344 for 2 matches, till the max number of matches we found was 28 matches for 4 sea cucumber scaffolds against the same sea urchin scaffold.

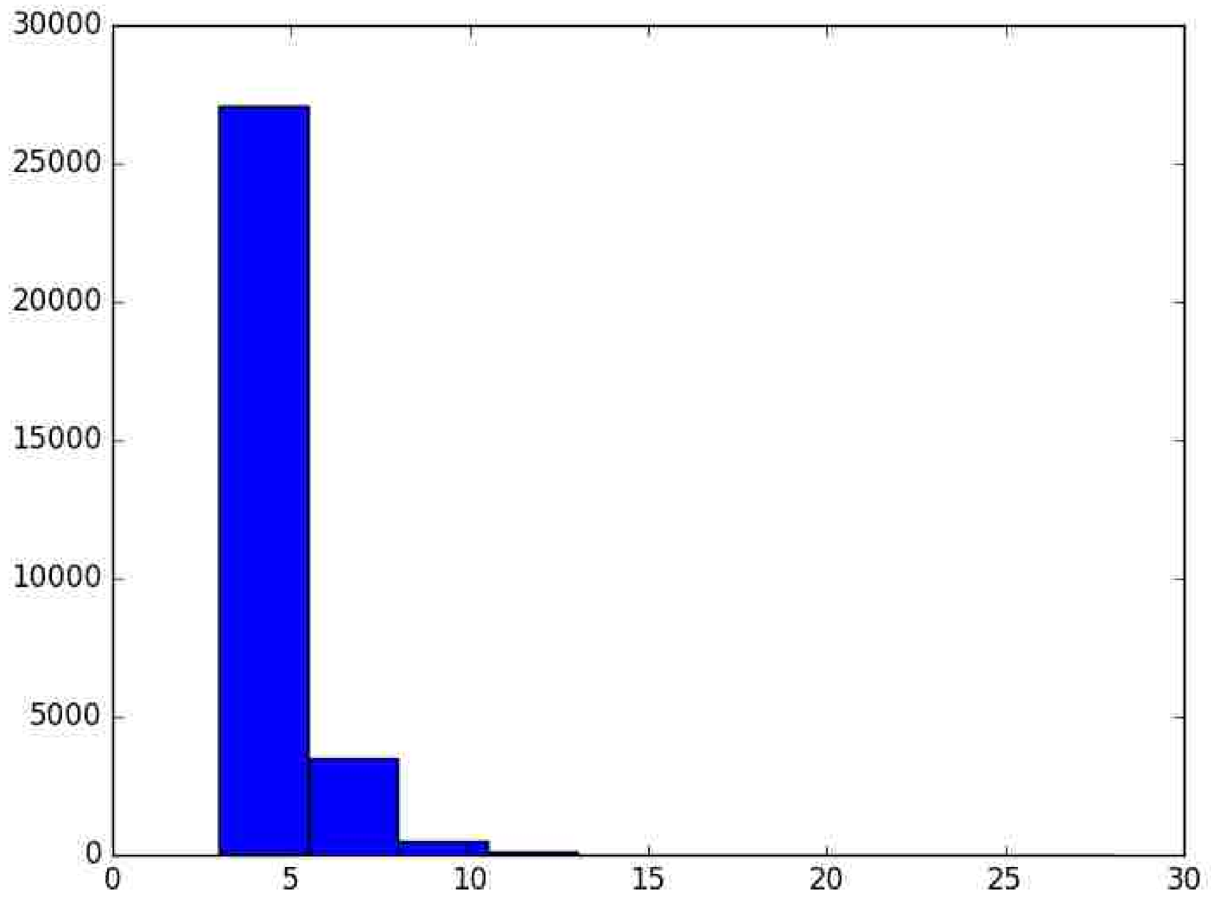


Figure 6.1: Histogram on number of sea cucumber scaffolds that had more than 2 matches against sea urchin. The x-axis represents the number of matches and the y-axis represents the total number of sea cucumber scaffolds that matched.

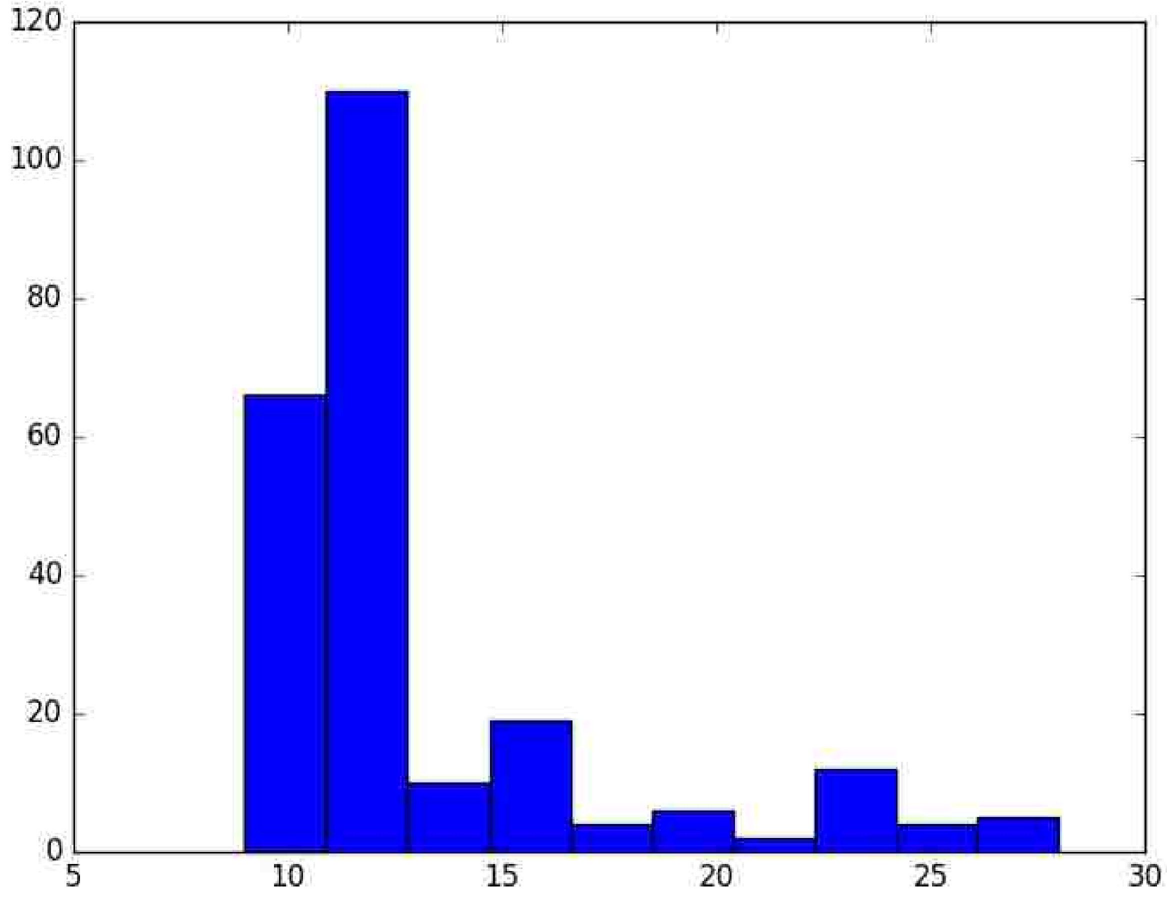


Figure 6.2: Histogram on number of sea cucumber scaffolds that had more than 8 matches against sea urchin. The x-axis represents the number of matches and the y-axis represents the total number of sea cucumber scaffolds that matched.

Chapter 7

Conclusion & Future Work

We took University of California's sequenced reads for sea cucumber and cleaned it up to generate scaffolds using various genome tools such as FastQC, Trimmomatic, SGA, SSPACE and GapFiller. We then attempted to match the scaffolds against HGSC's sea urchin scaffolds using minhash technique to find similar scaffolds.

Our future work involves the following:

- Trying out various other genome tools to further clean up the sea cucumber reads
- Improving our code to be more stable and efficient depending on job resources allocated for the job
- Running data used by other assemblers through ours to compare the results

Bibliography

[1] What's a genome, Feb 2002.

[2] Sea urchin genome project, Apr 2012.

[3] Unlv national supercomputing institute, Jun 2015.

[4] What is a genome?, Jan 2016.

[5] What is a genome?, Oct 2016.

[6] Simon Andrews et al. Fastqc: A quality control tool for high throughput sequence data. *Reference Source*, 2010.

[7] Marten Boetzer, Christiaan V Henkel, Hans J Jansen, Derek Butler, and Walter Pirovano. Scaffolding pre-assembled contigs using sspace. *Bioinformatics*, 27(4):578–579, 2011.

[8] Marten Boetzer and Walter Pirovano. Toward almost closed genomes with gapfiller. *Genome biology*, 13(6):1, 2012.

[9] A Bolger and F Giorgi. Trimmomatic: a flexible read trimming tool for illumina ngs data. *URL http://www.usadellab.org/cms/index.php*, 2014.

[10] Anthony M Bolger, Marc Lohse, and Bjoern Usadel. Trimmomatic: a flexible trimmer for illumina sequence data. *Bioinformatics*, page btu170, 2014.

[11] Andrei Z Broder. On the resemblance and containment of documents. In *Compression and Complexity of Sequences 1997. Proceedings*, pages 21–29. IEEE, 1997.

[12] Mark JP Chaisson, Richard K Wilson, and Evan E Eichler. Genetic variation and the de novo assembly of human genomes. *Nature Reviews Genetics*, 16:627–640, 2015.

[13] Peter JA Cock, Christopher J Fields, Naohisa Goto, Michael L Heuer, and Peter M Rice. The sanger fastq file format for sequences with quality scores, and the solexa/illumina fastq variants. *Nucleic acids research*, 38(6):1767–1771, 2010.

[14] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. *Mining of massive datasets*. Cambridge University Press, 2014.

[15] Eric Vallabh Minikel. Forward and reverse reads in paired-end sequencing, Dec 2012.

[16] James Schiemer. Illumina truseq dna adapters de-mystified. 2011.

[17] Jared T Simpson. Sga. https://github.com/jts/sga/wiki, 2011.

[18] Jared T Simpson. Sga. https://github.com/jts/sga/wiki/SGA-Design, 2011.

[19] A Vladimirov, R Asai, and V Karpusenko. Parallel programming and optimization with intel xeon phi coprocessors. *Colfax International*, 2015.

Curriculum Vitae

Graduate College

University of Nevada, Las Vegas

Saju Varghese

Degrees:

Bachelor of Science in Computer Science 2013

Bachelor of Science in Computational Physics 2013

University of Nevada Las Vegas

Thesis Title: Enhancing the Draft Assembly with Minhash

Thesis Examination Committee:

Chairperson, Dr. Kazem Taghva, Ph.D.

Committee Member, Dr. Ajoy Datta, Ph.D.

Committee Member, Dr. Laxmi Gewali, Ph.D.

Graduate Faculty Representative, Dr. Emma Regentova, Ph.D.