University of the Witwatersrand

Masters Dissertation

A Machine Learning Approach to DNA Shotgun Sequence Assembly

Author:
Radu-Ionut Constantinescu

Supervisor:
Dr. Ling Cheng

A dissertation submitted in fulfillment of the requirements for the degree of Master of Science in Engineering

Engineering and the Build Environment
School of Electrical and Information Engineering
CeTAS

July 2015

Declaration of Authorship

I, Radu-Ionut Constantinescu, declare that this thesis titled, 'A Machine Learning Approach to DNA Shotgun Sequence Assembly' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: ____

Date: ____

UNIVERSITY OF THE WITWATERSRAND

Abstract

School of Electrical and Information Engineering
Engineering and the Build Environment

Master of Science in Engineering

A Machine Learning Approach to DNA Shotgun Sequence Assembly

by Radu-Ionut Constantinescu

DNA sequencing and assembly is becoming increasingly prevalent in the field of bioinformatics. It is used in a variety of fields such as forensics and genetic engineering in order to sequence DNA of a species or specific individuals. The high computational complexity associated with DNA sequencing and assembly makes the process expensive to implement. In order to help reduce this complexity, a read grouping machine learning approach, which breaks the problem of assembly into multiple smaller sub-problems, is proposed. The shotgun sequencing process was performed on a 50456 base pair portion of the Drosophila Melanogaster (fruit fly) genome. The sequencing and assembly process was simulated under varying conditions of read size, coverage depth and sequencing error rates. The greedy and de Bruijn algorithms were first implemented as stand-alone assemblers and their performance was compared. Thereafter, a neural network system was implemented together with each of the two assemblers in order to investigate the effects a read grouping approach has on assembly performance. The performance of each of the four assemblers was then compared in terms of computational complexity and assembly accuracy using information theoretic principles along with a proposed coverage metric. It was found that the simulation time of the stand-alone greedy assembler was significantly improved when combined with the neural network read grouping approach. However, due to the higher relative complexity associated with the neural network training and grouping process, the same can not be said about the de Bruijn assembler. In order for the de Bruin assembler to benefit from this "divide and conquer" approach, faster grouping techniques need to be implemented.

Acknowledgements

First and foremost, I would like to thank my research supervisor Dr. Ling Cheng. You inspired me to undertake this journey and your contribution to this research was invaluable. I will always be grateful for our many discussions and the advice you gave me. I owe so much to you.

I would like to thank CeTAS and the School of Electrical and Information Engineering at the University of the Witwatersrand for enabling my research by providing me with the valuable facilities and equipment.

I also like to acknowledge and thank the contribution of Edward Steere to this research, whose insight and input regarding the field of bioinformatics was immensely valuable.

Last but not least, I would like to thank my loving family. To Florina Constantinescu, who always encouraged and supported me during my studies, and to Emma Jane Walland, whose support and valuable input helped me get over the finishing line.

Contents

List of Figures

List of Tables

Symbols

a	neuron activation
A	adenine base pair
c	coverage depth
c_{min}	minimum normalised coverage depth
$\bar{c}$	normalised coverage depth
C	cytosine base pair
D	edit distance
E	error function
G	guanine base pair
G	target DNA sequence length
$H_{Shannon}$	Shannon entropy
H_{α}	Renji entropy
I	number of input neurons
J	number of hidden neurons
k	size of k-mer
K	number of output neurons
L	read length
$\bar{L}$	normalised read length
$M_{adjusted\ coverage}$	adjusted coverage metric
$M_{largest\ contig}$	largest contig metric component
$M_{squared\ average}$	squared average metric component
N	number of reads
N_{cov}	read number coverage lower bound
N_{min}	read number assembly lower bound
O	big O limiting function

p	probability
P	message probability set
s	sub-sequence
S	sequence
t	output neuron target
T	thymine base pair
T	minimum overlap threshold
$\mathbf{T}$	neural network training set
w	neuron connection weight
$\mathbf{W}$	neural network weight vector
x	input neuron input
$\mathbf{X}$	neural network input set
$\mathbf{X_A}$	adenine input mapping
$\mathbf{X_T}$	thymine input mapping
$\mathbf{X_G}$	guanine input mapping
$\mathbf{X_C}$	cytosine input mapping
y	output neuron output
$\mathbf{Y}$	neural network output set
$\mathbf{Y_A}$	adenine output mapping
$\mathbf{Y_T}$	thymine output mapping
$\mathbf{Y_G}$	guanine output mapping
$\mathbf{Y_C}$	cytosine output mapping
z	hidden neuron activation
α	scaling factor
β	sigmoid function gradient
δ	error gradient
ϵ	error tolerance
η	learning rate
γ	sequence transformation function
ϕ	fractional overlap
σ	sigmoid function
τ	neural network grouping threshold
θ	time epoc

Chapter 1

Introduction

Deoxyribonucleic acid (DNA) is a molecule which encodes for genetic information in biological organisms. It achieves this by using a sequence of nucleotides, namely; Adenine, Thymine, Guanine and Cytosine, referred to by the letters A, T, G and C respectively [3]. In modern day medicine, DNA plays a major role in a number of important fields such as forensics, bioinformatics and genetic engineering. With the completion of the Human Genome Project and other similar projects aiming to sequence genomes for a variety of organisms, a major challenge for the past decade has been to digitise genetic information. This digitisation is achieved through the process of DNA sequencing and assembly and it opens the possibility to explore a new set of medical applications previously inaccessible. These applications include discovering genetic variations across different species, or members of the same species. It is possible to sequence an individual's genome and identifying genes affected by mutations caused by cancer. The benefits of such applications introduce new areas to the field of medicine, such as the ability to identify predispositions for certain diseases in individuals and the ability to produce tailor made personalised medicine based on an individual's genetics.

Current DNA sequencing and assembly technologies are unable to sequence entire genomes in one go. As a result the sequencing of large genetic information is split into two steps namely; the sequencing and assembly steps. The sequencing step is responsible for generating, or sequencing, multiple short length clones of the target being sequenced using gel-electrophoresis technologies [4, 5]. The assembly step is then responsible for piecing together these digital clones, also referred to as reads, in order to reconstruct the target sequence. Some examples of the DNA sequencing technologies used to generate the reads include the Sanger platform, 454 machines, Illumina and SOLiD [6, 7]. These technologies produce reads uniformly distributed across the target DNA sequence and which vary in length based on the sequencing technology used to generate them. Due to the distribution of reads across the target sequence, this technique of sequencing is referred to as shotgun sequencing [8]. In shotgun sequencing, enough reads are produced such that there exists a large degree of overlap between reads. This overlap then introduces a redundancy of information which is available to reconstruct the target sequence. The assembly step makes use of this redundancy in order to correctly assemble the pieces together. Figure 1.1 (a) shows the system block diagram of the DNA sequencing and assembly process. Figure 1.2 (a) shows an example of the shotgun sequencing procedure.

For the case of human whole genome sequencing, target sequences have in the order of 10^9 nucleotides (or base pairs) and the DNA sequencing technologies generate reads ranging between $100 - 1000$ base pairs in length [6–8]. For this reason, depending on the average length of reads, DNA assembly is required to piece together somewhere around 10^8 reads in order to reconstruct the original DNA target. Traditional assembly techniques compare reads and merge them if they are found to overlap with one another. More modern techniques, which use the overlap-layout-consensus paradigm,

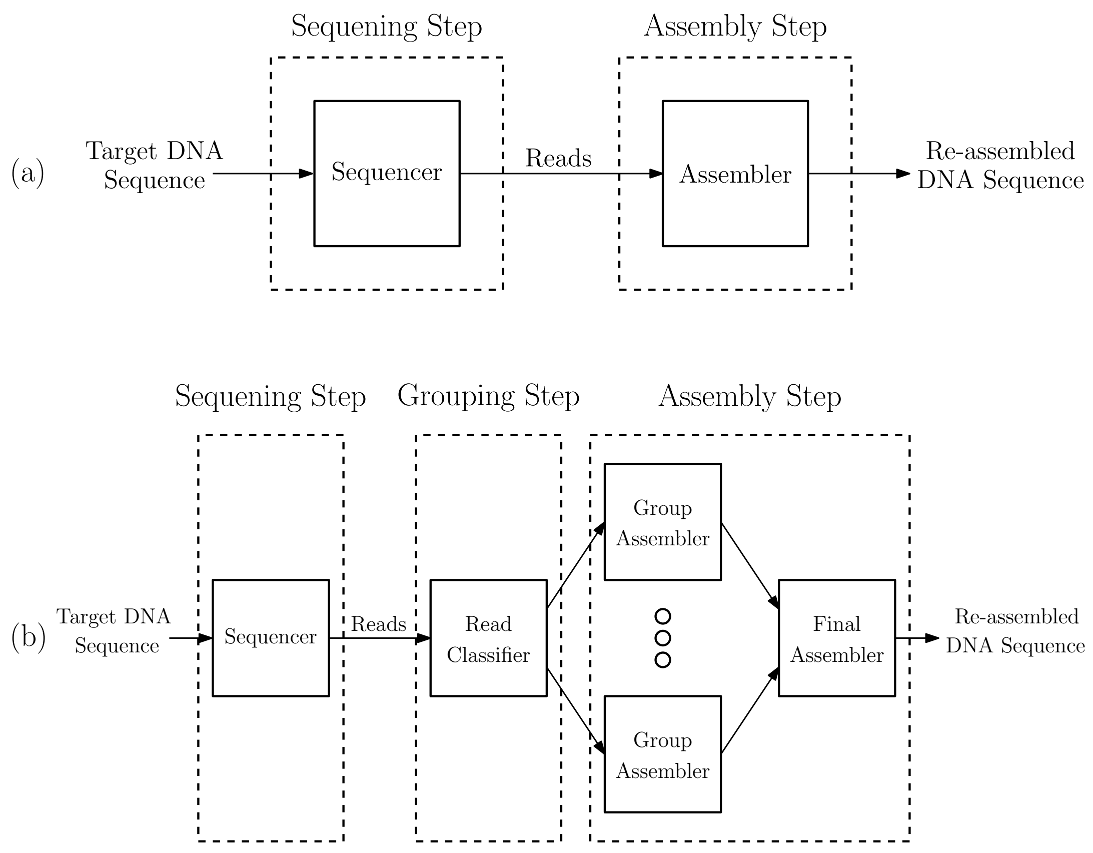


FIGURE 1.1: (a) The regular DNA sequencing and assembly step. The sequencing step generates small read sequences which the assembly step assembles together; (b) The DNA sequencing, grouping and assembly step. The extra grouping step classifies reads generated from the sequencing step into separate groups. Assembly is then applied to each group and then again on the outputs from each group.

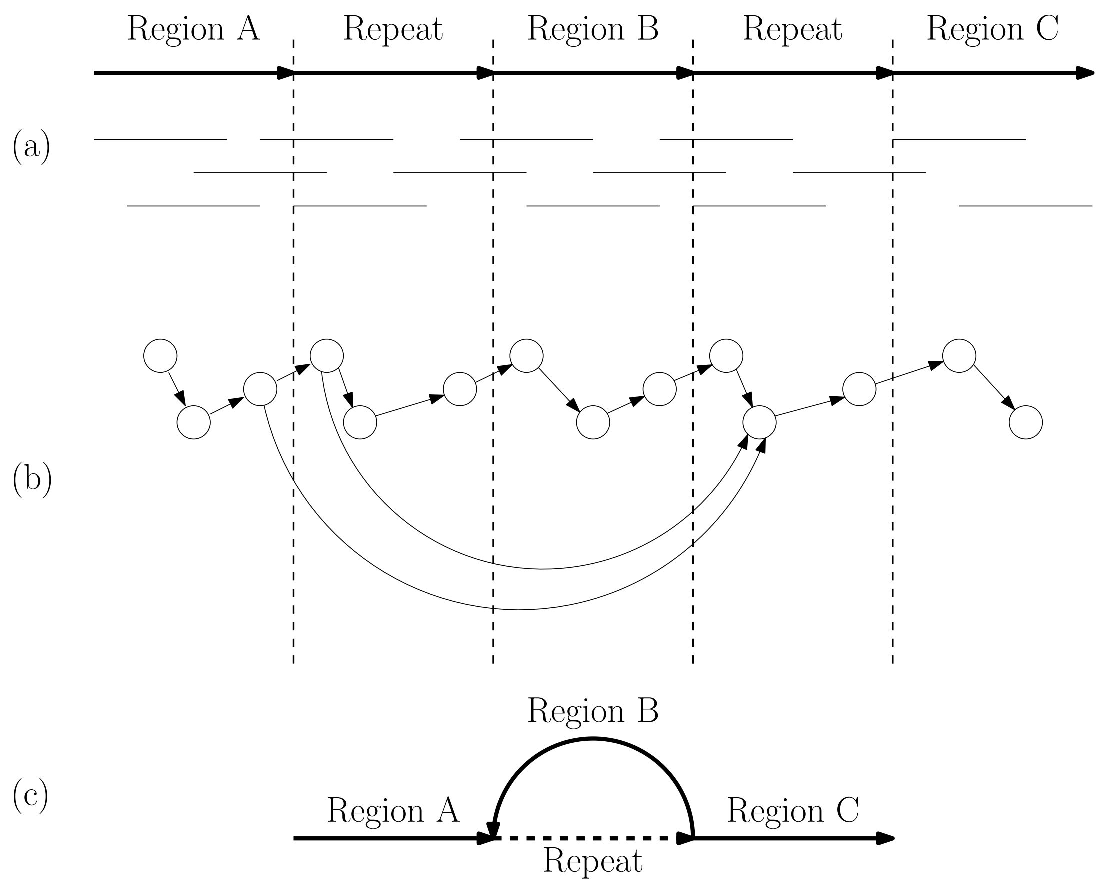


FIGURE 1.2: An example target sequence with repeat regions. (a) Shows the reads generated by the shotgun sequencing process; (b) Shows the layout overlap graph created from the reads; (c) Shows the de Bruijn graph created from the reads. This example is adapted from Pevzner et al [1].

convert this assembly process into a graph problem and solve it using graph theoretic principles [1]. An example of an overlap graph can be found in Figure 1.2 (b). These techniques typically use a greedy approach, or greedy algorithm, for assembling reads together by finding reads with the largest matching overlap and merging them [8–10]. However, due to the extremely large number of reads needed to sequence targets of such a magnitude, the assembly problem becomes incredibly complex and computationally expensive to solve in this manner. These techniques therefore only remain viable options for localised sequencing cases where only specific and much smaller areas of a genome or chromosome are sequenced.

The most recent techniques implement graph theoretic principles in order to solve the assembly problem. They make use of de Bruijn graphs, where vertices within these graphs are created by breaking reads into smaller pieces known as k-mers. Vertices are then connected together by edges if they are found to overlap with one another. This interconnectivity between vertices is what makes up the de Bruijn graph and allows one to re-assemble a sequence. The assembly problem is solved by using the Eulerian algorithm which finds a path, also known as the Eulerian path, through the de Bruijn graph. A Eulerian path is a path which visits every edge of the graph exactly once [1]. An example of a simplified de Bruijn graph can be found in Figure 1.2 (c). The de Bruijn graph technique solves the assembly problem in a more efficient manner which reduces the time complexity of the assembly process.

Another approach to reducing the time complexity of the assembly process is to first place reads into similar groups by introducing an additional step between the sequencing and assembly steps. In this pre-assembly step, a clustering machine learning approach for grouping reads is introduced. This grouping technique, inspired by Angeleri et al [11], attempts to reduce complexity for the assembly techniques by implementing a "divide and conquer" approach. Figure 1.1 (b) shows how the overall sequencing and assembly process is modified with the introduction of this new step. The research explores this grouping approach and implements it together with existing assembly algorithms.

1.1 Research Aims and Objectives

The aim of this research is to tackle the problem of DNA sequence assembly by combining a machine learning approach with commonly used assembly techniques. In particular, an artificial neural network system is implemented which divides reads into separate groups with the aim of reducing the overall complexity for the assembly process. Both overlap and de Bruijn graph assembly techniques are implemented with and without the machine learning grouping approach and the performance of each assembly approach is then compared.

The research purpose is to compare the performance of these assembly approaches using a thorough analysis from an information theoretic perspective. The research will determine if the "divide and conquer" approach, by grouping reads, will reduce computational complexity and improve the performance of DNA assembly techniques. The performance of each approach is presented using a new metric which takes into account both the number and accuracy of the assembled outputs. This will be achieved with the following objectives in mind:

- Establish a good foundation based on information theoretic principles for analysis of the assembly techniques;
- Simulate the overlap assembly technique using the greedy algorithm;
- Simulate the k-mer assembly technique using the de Bruijn graph and a path finding algorithm;

- Develop a clustering artificial neural network which places reads into separate groups and then implement it together with the two assembly techniques;
- Using an appropriate measure of performance, compare each assembly approach in terms of computational complexity (time complexity) and performance (accuracy) of assembly.

According to the standards defined by the National Human Genome Research Institute, for a reconstruction to be deemed successful, at least 95% of the original DNA target sample sequence must be reconstructed [12]. With this in mind, and with the aid of an information theoretic analysis, the research question attempts to determine the following: for a given DNA sample, which of the two proposed assemblers (greedy and de Bruijn) achieves the most accurate assembly and at what computational complexity cost? Additionally, how does a read grouping neural network assembly scheme affect the overall assembly process and the outputs of each of the assemblers?

1.2 Research Resources, Scope and Dissertation Layout

The assembly techniques, along with the machine learning approach, are implemented by means of a C++ simulation. In order to speed up the simulation process a number of simulation machines have been provided by the CeTAS research group at the school of Electrical and Information Engineering. The target DNA sequence data has been obtained from the FlyBase database [13]. This data consists of an already sequenced portion of the Drosophila Melanogaster (fruit fly) genome, however some preprocessing is required in order to convert this data into a usable format. The shotgun sequencing process is then simulated on this data in order to generate the multiple reads. The assembly approaches are simulated and the resulting reconstructed DNA sequence is verified according to the known FlyBase sequenced genome. The shotgun sequencing simulations are implemented with the following assumptions:

- All reads are generated from the same single DNA strand;
- All reads are generated at the same specified size.

These assumptions are implemented in order to limit the scope and complexity of the research.

The rest of this dissertation is structured as follows. A literature review covering important sources is performed in Chapter 2 in order to better establish the research context. Chapter 3 provides more fundamental detail on the concepts being researched and implemented. Chapter 4 covers the implementation of existing assembly techniques. Chapter 5 covers the implementation of the machine learning approach to the DNA sequencing and assembly problem. In Chapter 6 all results are provided and analysed. Finally, Chapter 7 will conclude the research.

Chapter 2

Literature Review

Large advances have been made in the field of DNA sequencing and assembly which has helped in reducing the time and cost associated with the process. This chapter investigates literature related to DNA sequencing and assembly. Both legacy and current state-of-the-art techniques in DNA sequencing and assembly are investigated. This chapter also explores the Information theory and machine learning in the context of the DNA sequencing and assembly problem and helps to lay down the foundation of the research. Through analysis of the literature it is shown that the feasibility of accurate assembly depends largely on the complexity of the target and the size and coverage depths used in the sequencing process. Additionally, it is found that a gap in the literature remains with regards to combining the clustering machine learning approach with more modern k-mer assembly techniques.

2.1 The Origin of DNA Sequencing

The field of DNA sequence assembly has been created due to the increasing need for sequencing genetic information. The Human Genome Project and other similar projects have created this demand. They seek to increase the rate of DNA sequencing while reducing its cost. Obtaining the genetic information of a particular species or individual plays a vital role in a number of applications. It is an important tool which helps in the biological analysis of organisms and the prevention of diseases. This genetic information is expressed by genes located at various regions within chromosomes. These chromosomes typically consist of three regions namely; heterochromatic, euchromatic and centromere regions [3]. While all three regions play a role in the encoding of genetic information, the euchromatic region is regarded as the most gene rich as it consists of around 93% of the human genome [14] and around 66% of the fly genome [15]. Further study of the human and fly genome has revealed the presence of repeated regions. These regions consist of perfect, or near perfect, repeats of regions within the genome. Repeat regions mostly exist within heterochromatic regions of the genome however a small amount of such regions, around $5-6\%$ of the human genome and significantly less for the fly genome, have also been found in the euchromatic regions [15, 16]. These regions present a problem to the sequencing process as it introduces ambiguity in the assembly process leading to the highest potential for causing mis-assembly. It is therefore critical for modern day assemblers to overcome the presence of repeats within the genome of a target sequence.

One of the early predecessors to modern day DNA sequencing was a technique known as sequencing by hybridisation. This technique made use of DNA arrays, or sequencing chips, to determine the sequence of a single stranded DNA target. These arrays consist of small DNA l-tuples, also known as probes, which bind (or hybridise) to the complimentary sub-strings if they exist within the target sequence [17]. Once all the l-tuples present in the target DNA are identified this data can then be used to assemble the target sequence. This assembly problem is also referred to as the super-string problem which uses the overlap-layout-consensus paradigm [1] and has been shown to be an NP-complete problem [18]. This high complexity cost associated with solving the super-string problem prompted the need for more efficient ways of solving the DNA sequence assembly problem. This was achieved by P. A. Pevzner whose proposed solution laid the foundations for the modern day assembly techniques covered later in this chapter [19]. His approach solved the problem by constructing de Bruijn graphs and finding Eulerian cycles through them in order to reconstruct the target sequence.

Other approaches to generating sequence data using polymerase chain reactions and electrophoresis techniques led to the creation of the shotgun sequencing process [5, 6]. This process generates clones, or reads, of the target sequence instead of using DNA arrays as in the sequencing by hybridisation process. The size and number of these reads varies depending on the sequencing technology being used and is further discussed in Section 2.2 of this chapter.

As the field of DNA sequence assembly grew a common set of symbols and lexicon has become established within the field. Based on popular literature the following list of symbols have become synonymous with DNA sequence assembly;

- G = Target DNA sequence length (in base pairs);
- L = Read length (in base pairs);
- N = Number of reads taken;

- $c = \frac{LN}{G}$ = Coverage depth (redundancy of coverage).

The coverage depth c is an important term associated with DNA sequence assembly which greatly affects the performance and ability of assemblers to reconstruct the target sequence. It also plays an important role and is a critical component in the information theoretic analysis performed in Section 2.5 of this chapter. The term "contig" is used in popular literature to refer to a contiguous fragment which consists of the combination of one or more sequenced reads [8]. In the context of this research the term contig is used to refer to an output from an assembler in the DNA assembly step. Typically, output contigs consist of multiple merged reads, however in some situations they may consist of only a single read. It is also common to produce multiple output contigs from a single assembly round [20]. In such cases the contigs are disjoint from one another and are referred to as "islands". The gaps between these contigs, which are gaps in the sequenced target, are known as "oceans".

2.2 Shotgun Sequencing Technologies

The invention of newer sequencing technologies using polymerase chain reactions and electrophoresis has led to the introduction of the shotgun sequencing process for creating read data. The success of the shotgun sequencing methods is due to the larger read sizes which can be generated with the newer sequencing technologies. These technologies are able to create much larger reads when compared to the sequencing by hybridisation approach. Reads used in sequencing by hybridisation are in the range of $8 - 25$ base pairs [17] whereas newer technologies generate reads ranging between $25 - 1000$ base pairs [5–7]. By creating large sets of such reads much higher coverage depths can be achieved which significantly improves the accuracy for a given reconstruction.

The first generation of shotgun sequencing technologies known as the Sanger machines created reads ranging between $100 - 1000$ base pairs in length. As DNA sequencing and assembly evolved newer shotgun sequencing technologies began to focus on generating much smaller reads. The 454 machines produced an average read length of 250 base pairs while the Illumina and SOLID technologies produced reads in the range of $25 - 35$ base pairs [7]. The feasibility of using smaller reads in the sequencing process was analysed by Whiteford et al [21] where it was shown that reads ranging between $25 - 50$ base pairs in length were adequate in the sequencing of bacterial targets, however for larger and more complex targets, larger read sizes are required. Since every base pair increases the complexity of a sequence by a factor of four [22], observing a unique sequence significantly reduces when reducing the read length below 20 base pairs. Additionally, achieving maximal uniqueness within the read set depends on the size and complexity of the target being sequenced [21]. Recent technologies such as Velvet [23] have however improved upon the performance of short read length assemblers in the assembly of mammalian target sequences.

The shotgun sequencing process generates reads which are uniformly distributed across the target sequence [8] with a typical error rate of 1% per base pair and below [23, 24]. Typically, between $10 - 60$ times coverage depths is needed in order to ensure there are no gaps, or oceans, for a given target sequence [8]. For whole genome assembly, where large target sequences are used, shotgun sequencing generates around 27×10^6 reads [16]. This large number of reads introduces the complexity issues associated with the assembly step of DNA sequence assembly.

2.3 Assembly Techniques

There are two popular and widely accepted approaches when it comes to the assembly step of the DNA sequence assembly problem. The first compares overlapping regions between reads in order to establish a degree of similarity. Contigs are created and extended if the overlapping prefix or suffix regions between reads are found to be similar [8]. The second approach breaks reads into smaller k-mers which are used to create a de Bruijn graph. The problem of assembly is then solved using a graph theoretic approach making use of path finding algorithms to find Eulerian cycles through the graph [1]. These two assembly techniques are discussed in Section 2.3.1 and Section 2.3.2 respectively.

2.3.1 Overlap Assembly

The overlap assembly techniques solve the DNA sequence assembly problem by identifying similar reads using an appropriate similarity measure and merging them to create contigs. These techniques find a similarity between overlapping regions of reads by using an overlap score as defined by Lander and Waterman [20]. This overlap score is defined as the number of similar base pairs between similar regions of two reads. These regions correspond to either the prefix or suffix areas of reads and are required to be above some minimum overlap threshold value defined as

- T = Minimum amount of base pairs needed to detect an overlap;
- $\phi_{min} = \frac{T}{L}$ = Minimum read fraction needed to detect an overlap.

Another popular measure of similarity is the edit distance metric [25]. This edit distance, also known as the Levenshtein distance, is defined as the minimum number of insertions, deletions or substitutions required to transform one string, or read, into another [26]. Hence, the lower the edit distance cost between two reads, the greater their degree of similarity. Edit distance comparison was first applied to protein sequences by Needleman and Wunsch [27] and has since been extended to DNA sequences. Work by Chang and Lawler [28] has shown how finding the edit distance between two strings can be done using Dynamic Programming [29]. Smith and Waterman [30] also discuss the use of a similarity matrix when comparing sequences. This matrix identifies areas within sequences which are considered most similar, taking into account insertions, deletions and mismatches between base pairs.

A number of popular assemblers implementing an overlap assembly approach have been developed. These assemblers find reads which are considered similar and merge them together to form contigs in a greedy manner. This approach has been implemented by assembler technologies such as the TIGR [31], PHRAP [8] and CAP series of assemblers [9, 10] which make use of a greedy strategy in order to piece together contigs in a parallel fashion. Some other overlap assemblers such as the SSAKE [22], VCAKE [32] and SHARCGS [24] assemblers use a sequential greedy strategy for assembling contigs. These assemblers are presented in the paragraphs that follow.

The TIGR assembler claims to overcome several obstacles associated with the shotgun sequencing assembly of large target sequences. The most important of which is its ability to deal with the presence of repeat regions in the target sequence. The TIGR assembler identifies repeat regions by separating reads into repeat and non-repeat categories. Reads are classified by measuring the possible overlaps with all other reads once a pairwise comparison of every read has been performed. A threshold is used to differentiate between repeat and non-repeat reads. This threshold is biased towards over-labeling reads as repeat reads in order to minimise the chance of erroneous merges with contigs. TIGR first creates and extends contigs by only using non-repeat reads until the ends of contigs only have potential overlaps with repeat reads. Lower coverage regions are assembled only if there are no other reads with stronger overlap data. These lower coverage regions occur due to the randomness of the shotgun sequencing process where some reads have overlap possibilities below the median of other reads. The TIGR assembler has been used to correctly assemble the Haemophilus influenzae and Mycoplasma genitalium genomes [31].

The CAP series of assemblers perform the assembly process in several steps. After a pairwise comparison between each read is completed, reads with erroneous or no overlaps are removed. A dynamic programming algorithm is also used to compute an overlap alignment score based on the Smith-Waterman algorithm [30]. This overlap score measures the degree of similarity between reads and is used in a greedy manner to merge similar reads into contigs [9, 10]. For the case of CAP3, an additional consensus step is performed which makes corrections to contigs. This consensus also evaluates the quality of the contigs by assigning a quality value to each base pair. These quality values are then used to remove poor quality areas from contigs when performing pairwise overlap comparison between reads and contigs [10].

The SSAKE assemblers are designed to operate on high coverage short length reads [22]. They are used in the sequencing of short length targets in the region of $30k$ base pairs. The SSAKE assembly algorithm makes use of a hash table which is keyed by the first 11 unique base pairs within each read, with the number of times they occur. A prefix tree is then used to sort the sequences according to their occurrence. Starting with an 11-mer base pair window, the ends of the assembly contig are compared with the ends of the reads populating the hash table. The search time for this process is significantly reduced using prefix tree sorting. When a match is found, the remaining unmatched portion of the read is appended to the assembly contig and the read is removed from the hash table and prefix tree. If no matches are found then the k-mer window is reduced in size. This process is repeated until a user defined minimum is reached for the k-mer window. A limitation of the SSAKE assembler is that it operates on only error-free reads. This is considered unrealistic due to the error rates associated with modern day sequencing technologies.

The VCAKE assembler improves upon the SSAKE algorithm by introducing a tolerance to error introduced in the shotgun sequencing process [32]. VCAKE achieves this by introducing two primary changes to the SSAKE algorithm. Firstly, the VCAKE assembler makes use of larger coverage depths compared to SSAKE. Secondly, matches between reads and the assembly contig are recorded and stored in an array instead of greedily merging the first match. A majority vote is then used to merge the most likely correct read with the assembly contig.

The SHARCGS assembler is another short read assembler which assembles reads between 25 and 40 base pairs in length [24]. Similar to the CAP3 assembler, it makes use of a quality score to rate the quality of reads. The assembly process then filters out poor quality reads before the assembly contigs are created and extended by the reads. Once the poor quality reads have been removed from the pool of reads, a pairwise comparison is performed to join the remaining reads to the assembly contig. The process of extending a contig also involves creating a verification region. This consists of a read sized portion of the contig suffix and the non-matching remaining portion of the read to be appended to the contig. This verification region is then used to generate all possible sub-strings of a predefined minimum length from within this region. If all reads containing these substrings as prefixes have a matching overlap with the verification region, then the read to be appended is merged with the assembly contig. This process is also repeated in the reverse compliment in order to extend the prefix of the assembly contig in a similar manner.

The assemblers discussed in this section follow the overlap-layout-consensus paradigm. They make use of the overlap step to perform a pairwise comparison between every read. The layout step then merges the reads together, and finally error correction is performed in the consensus step. If this approach had to be analysed using graph theoretic principles, the overlap step can be represented by creating an overlap graph as in Figure 1.2 (b). Every read corresponds to a vertex in this graph and two vertices are connected by an edge if they overlap with one another. Assembly is then performed by finding a Hamiltonian path through the graph where every vertex is visited exactly once [1]. The issue with this overlap approach to the DNA sequencing and assembly problem is that there is no polynomial time solution for the Hamiltonian path problem and regardless of the strategy used, the overlap approach has been shown to have $O(N^2)$ complexity [8, 18], where N is the number of reads taken during the shotgun sequencing process.

2.3.2 k-mer Assembly

Work done by Idury and Waterman [33] and Pevzner [19] has since extended the sequencing by hybridisation problem and has introduced more modern approaches to the DNA sequence assembly problem. They propose the construction of de Bruijn graphs to help in the sequence assembly process. The approach attempts to solve the assembly problem by dividing the reads into smaller $k < L$ pieces [1, 8, 34]. These pieces are then represented as edges of a de Bruijn graph where an edge between two $(k-1)$-mer vertices exists if they are adjacent to one another in a particular k-mer. The assembly problem then becomes a task of finding the Eulerian path through the graph, where this path represents the reconstructed DNA sequence [1, 8]. A number of different de Bruijn graph algorithms exist. The most recent are applied by the Velvet series of assemblers [23]. Additional algorithms such as the DEBRUIJN, SIMPLEBRIDGING and MULTIBRIDGING are also presented by Tse et al [34] while Pevzner et al [1] introduce the EULER algorithm. These k-mer assembly techniques are presented in the paragraphs to follow.

The Velvet assembler is one of the most successful de Bruijn graph assemblers. Unlike overlap-layout-consensus assemblers, Velvet performs the error correction as one of the first steps [23]. After the de Bruijn graph is created, simple and non-ambiguous connections are first resolved by merging vertices together. Two error correcting steps are then implemented to remove “tips” and “bubbles” from the graph caused by the sequencing process. The final step involves resolving repeats present within the graph such that correct Eulerian paths through the de Bruijn graph can be found. Because of the short read nature of Velvet, output contigs consist of between $2k$ to $3k$ base pairs in length for mammalian targets [23]. For this reason, other methods such as those implemented by overlap assemblers, need to be combined with Velvet in order to perform whole-genome-assembly [21, 23].

The EULER assembler proposed by Pevzner et al [1] is another de Bruijn graph approach which abandons the overlap-layout-consensus paradigm. Similar to Velvet, EULER attempts to simplify and resolve the de Bruijn graph before a Eulerian path can be found. It does this using “detachment” and “cut” techniques which make use of read information to create sub-paths within the graph and remove unwanted edges between vertices [1].

Another example of k-mer assembly is the ARACHNE assembler which is capable of whole genome assembly [35]. This is a hybrid assembler which sorts k-mers into map containers and then uses dynamic programming to find overlaps between reads. Because of the overlap assembly approach this assembler implements, it still suffers from time complexity issues. The ARACHNE assembler requires up to 8 days to sequence the genome of a mouse which demonstrates the need to improve upon the time complexity of such overlap assemblers [35].

The de Bruijn graph series of assemblers have introduced a more efficient means of performing the assembly phase of DNA sequencing and assembly. The complexity associated with the k-mer de Bruijn graph assemblers has been shown to be $O(N \log N)$ [34] which is a considerable improvement over the $O(N^2)$ complexity of the overlap assembly techniques.

2.4 Clustering Reads Using Machine Learning

An alternative approach to reducing the overall computational complexity of the DNA sequence assembly problem is to pre-allocate reads into similar groups before the assembly step is implemented. This clustering of reads can be performed with the use of machine learning.

Machine learning is a branch of artificial intelligence which aims at learning from given data. It has various applications such as pattern recognition, classification and clustering

[36]. A system is first trained using appropriate features from a given data set, and once trained, the system can be used to generalise other inputs. The term machine learning is a general term used to refer to a variety of different models and algorithms [36]. The accuracy of generalisation for a trained system depends greatly on the model and algorithms used in the machine learning process. Artificial neural networks and support vector machines are examples of some popular machine learning models.

The "divide and conquer" approach is being investigated in order to determine if there is any potential in reducing complexity and improving performance to the DNA assembly problem. An edit distance analysis could be used to sort reads into similar groups. However, the calculation for this edit distance requires Dynamic Programming techniques and can become expensive for a large read length L [25, 28]. An alternative approach proposed by Angeleri et al [11] makes use of machine learning to train neural networks to recognise, or track, specific reads and then group reads according to these tracked reads. This machine learning approach implements a recurrent 3-layer perceptron neural network which uses the backpropagation through time learning algorithm in order to cluster reads into similar groups [37]. This system makes use of five input nodes in the first layer (one for each possible base pair, plus a fifth in order to accommodate for the possibility of ambiguity caused by errors in the reads), 27 nodes in the hidden layer, and 4 nodes for the output layer (one for each possible base pair). In the literature, the CAP3 assembler was used to assemble each group [11], and the output from each group. However, a gap remains in the research, as potentially any assembly technique may be used instead of CAP3.

The use of a recurrent neural network, as opposed to a regular feed-forward neural network, introduces memory to the system [37]. This is justified by the need to address previous elements of a specific read in order to more accurately track it. There is a trade-off between accuracy and complexity regarding the number of nodes in the hidden layer; more nodes results in a more accurate but more complex system with slower computation, while less nodes results in a less accurate and less complex system with faster computing times [36]. Based on this principle, the number of nodes used in the work presented by Angeleri et al [11] was determined using trial and error.

2.5 Information Theoretic Analysis

The successful assembly of a target sequence greatly depends on the number and size of the reads (coverage depth) generated by the shotgun sequencing process. A number of works have been done in order to determine optimal values for these parameters. The work by Lander and Waterman established the mathematical foundation and bounds necessary for the overlap approach to the DNA assembly problem [20]. It showed how the number of islands, oceans, and average contigs per island can be calculated. This led to works by Tse et al [8] and Pervzner et al [34] which established critical lower bounds on the number of reads needed for successful reconstruction of a target sequence of a given size.

It was further discovered that the repetitive nature of the genetic information present within the genome of organisms also has a great impact on the size of reads generated by shotgun sequencing [8, 34]. It was shown that for successful reconstruction, repeats within the target sequence need to be fully covered by reads [8, 34]. These repeats further complicated the assembly process and need to be carefully considered when choosing appropriate assembly algorithms.

2.6 Literature Review Summary

Large advancements have been made in the field of DNA sequencing and assembly which has helped in reducing the time and cost associated with the process. Through analysis of the literature it has been shown that the feasibility of assembly depends largely on the complexity of the target and the size and coverage depths used in the sequencing process. However, given the greedy and de Bruijn graph assembly algorithms along with the clustering (read grouping) technique implemented using neural networks, there still remains a gap in research when combining these techniques together.

Chapter 3

Information Theoretic Background and Assembler Fundamentals

In this chapter, a more in depth analysis on important concepts is performed. These concepts are vital to the implementation and analysis of the assembly and grouping techniques discussed in this research. An information theoretic analysis is performed in order to determine under which conditions assembly is achievable. A k-mer approach to the DNA assembly problem is presented which promises to significantly reduce computational complexity in the assembly process. Additionally, a background into neural networks and the backpropagation training algorithm is also presented in order to support Chapter 5 which deals with the machine learning approach to the DNA assembly problem.

3.1 Entropy for Information Measure

Before analysis on the assembly techniques can begin, it is important to determine what information in the context of DNA assembly means. The early and well known work by Shannon [38] in the field of information theory defines information as a measure of freedom one has in selecting a message from a possible set. This relates to the concept of entropy to a large degree. The greater the freedom of choice, the greater the entropy in the system. Hence, entropy can be seen as a measure of Information by measuring the randomness within a system. In a binary case, Shannon defines information as the base two logarithm of the number of available choices that can be represented using binary bits [38, 39]. For example, a system with four possible messages; (00), (01), (10) and (11), the measure of information is $\log_2 4 = 2$. In the case of Shannon entropy, one needs to look at the occurrence probability for each message in addition to the number of available choices. Hence, for a stochastic system, the Shannon entropy is given as follows

$$H_{Shannon}(\mathbf{p}) = \sum_{i=1}^{m} -p_i \log_2 p_i \tag{3.1}$$

where $P = \{p_1, p_2, \cdots, p_m\}$ is the set of probabilities for each possible message. For the case of DNA there are only four possible message choices corresponding to each base pair A, T, C and G. Additionally, base pairs are not uniformly distributed across the genome of a species [3]. For example, the distribution of base pairs in humans is given as

$$A = 30.3\% \ T = 30.3\% \ G = 19.5\% \ C = 19.9\%$$

while the distribution for the common fly is as

$$A = 27.3\dot{3}\% \ T = 27.6\dot{6}\% \ G = 22.5\% \ C = 22.5\%$$

Using Equation (3.1) and the given distributions, the greatest achievable Shannon entropy when sequencing a human genome is $H_{Shannon}(\mathbf{p}) = 1.967$. For the fly genome it is given as $H_{Shannon}(\mathbf{p}) = 1.993$.

Another measure of information used in the work by Tse et al [8] is the Renyi entropy, a generalisation of the Shannon entropy [39]. The Renyi entropy is given as follows

$$H_\alpha(\mathbf{p}) = \frac{1}{1-\alpha} \log_2 \sum_{i=1}^{m} p_i^\alpha \tag{3.2}$$

where $\alpha > 0$ and $\alpha \neq 1$. It is shown that if $\alpha = 1$ then the Renyi entropy becomes the Shannon entropy which indicates that the Shannon entropy is a special case of the Renyi entropy, where $H_{Shannon}(\mathbf{p}) = H_1(\mathbf{p})$ [39]. The importance of this entropy measure becomes apparent when the coverage and read size lower bounds are established in the later sections.

3.2 Assembly Lower Bounds

An information theoretic approach to the DNA sequence assembly problem helps to establish some necessary bounds related to the assembly process. These bounds are useful in determining important properties such as the number and size of the reads needed to successfully assemble a given target sequence. This analysis is established using principles from DNA sequencing theory [40].

An important lower bound was first established in the work by Lander and Waterman [20]. This bound N_{cov}, defines the minimum number of reads needed in order to cover the entire target DNA sequence with a probability of at least $1 - \epsilon$, where ϵ is some error tolerance. This bound is significant as it ensures that no gaps, or oceans, are present in the coverage of a target sequence. The presence of gaps is undesirable as they lead to disjoint islands in the assembly process. The N_{cov} lower bound is given by the approximation

$$N_{cov}(\epsilon, G, L) \approx \frac{G}{L} \ln\left(\frac{G}{L\epsilon}\right) \tag{3.3}$$

where G is the target sequence length and L is the read length [8]. Equation (3.3) therefore acts as an absolute lower bound on the number of reads needed in the assembly process regardless of the assembly algorithm being used.

Another important bound is based on Ukkonen's condition which states that for a given target sequence S and set of reads, if there exists two interleaved repeats or a triple repeat whose copies are not fully covered by a read, then there exists another sequence $S^{'}$ which can be reconstructed [41]. The condition imposes a lower bound on the read length L [34].

$$L > l_{crit} := max\{l_{interleavied}, l_{triple}\} + 1 \tag{3.4}$$

This bounds addresses the issue of repeats which occur within DNA. It requires that the read length be greater than the largest repeat such that it can cover, or bridge, the repeats. This condition is further confirmed in the analysis by Pevzner et al [1], where it is shown that even for the de Bruijn graph assembly techniques, it is a requirement that reads fully bridge repeats. Therefore, for reconstruction of the target sequence to be successful, all interleaved and triple repeats need to be bridged.

In this research repeats are defined in a similar manner as in [34]. Let s_t^l denote a read of length l at position t within a DNA sequence S. A repeat is a read of length l appearing twice at some positions t_1 and t_2. Therefore $s_{t_1}^l = s_{t_2}^l$ and the following is true: $s(t_1 - 1) \neq s(t_2 - 1)$ and $s(t_1 + l) \neq s(t_2 + l)$. Similarly, a triple repeat is a read appearing three times at positions t_1, t_2 and t_3, where $s_{t_1}^l = s_{t_2}^l = s_{t_3}^l$ and where $s(t_1 - 1) \neq s(t_2 - 1) \neq s(t_3 - 1)$ and $s(t_1 + l) \neq s(t_2 + l) \neq s(t_3 + l)$. A pair of repeats, the first at t_1 and t_3 and the second at t_2 and t_4, is interleaved if $t_1 < t_2 < t_3 < t_4$ or $t_2 < t_1 < t_4 < t_3$.

It is shown in [34] that the existence of unbridged interleaved or triple repeats occurs with the following probability

$$\begin{aligned} P_l^{unbridged} &:= P[length\ l\ subseqence\ is\ unbridged] \\ &= e^{\frac{N}{G}(L-l-1)} \end{aligned} \tag{3.5}$$

Because an unbridged repeat will result in an erroneous assembly Equation 3.5 shows the probability of making an error. Hence, if $P_l^{Unbridged} \leq \epsilon$, then the lower bound for the number of reads can be given as

$$N_{bridge} \geq \frac{G}{(L - l_{repeat} - 1)\ln\left(\frac{1}{2\epsilon}\right)} \tag{3.6}$$

where N_{bridge} is the minimum number of reads needed to ensure that all repeats are bridged [34].

3.3 Information Theoretic Analysis of Greedy Algorithms

Satisfying the lower bound given in Equation (3.3) ensures that the entire target sequence is covered. This however does not guarantee the successful reconstruction of the target sequence. The true number of reads required for successful reconstruction depends on the assembly algorithm being used and is at least equal to N_{cov}, i.e.

$$N_{min} \geq N_{cov} \tag{3.7}$$

Tse et al [8] proposed that the minimum number of reads required for successful reconstruction using the greedy algorithm satisfies the bound

$$N_{min} \leq \frac{G}{L}\ln\left(GL^3\right) \tag{3.8}$$

It was shown that the assembly error rate ϵ tends to zero as the number of reads tends to N_{min} [8]. Hence, the minimum number of reads N_{min} needed for successful reconstruction of a target sequence using the greedy overlap algorithm is at most bound by Equation 3.8. It is important to note that the value of N_{min} differs when applying $k-$mer assembly techniques.

Further analysis done by Tse et al [8], identifies an additional bound on the read length L needed to successfully reconstruct the target sequence using the greedy algorithm. This bound is obtained using the ratio between the minimum number of reads needed for successful reconstruction N_{min} and the number of reads needed to cover the sequence N_{cov}. This ratio is asymptotically analysed as the read length L and sequence length G tend to infinity. This ratio is known as the minimum normalised coverage depth and is given by

$$c_{min}(\bar{L}) = \lim_{G\to\infty, L=\bar{L}\log_2 G} \frac{N_{min}(\epsilon, G, L)}{N_{cov}(\epsilon, G, L)} \tag{3.9}$$

where $\epsilon \in (0, \frac{1}{2})$ is some error tolerance and $\bar{L}$ is a normalised parameter given by

$$\bar{L} = \frac{L}{\log_2 G} \tag{3.10}$$

Using the Renyi entropy as a measure of information, it can be shown that

$$c_{min}(\bar{L}) = \begin{cases} \infty & \text{if } \bar{L} \leq \frac{2}{H_2(\mathbf{p})} \\ 1 & \text{if } \bar{L} > \frac{2}{H_2(\mathbf{p})} \end{cases} \tag{3.11}$$

where $H_2(P)$ is the Renyi entropy of order 2 defined as

$$H_2(\mathbf{p}) = -\log_2 \sum_i p_i^2 \tag{3.12}$$

p_i is the probability of each symbol (A, T, G, C) in the DNA sample sequence [8].

Equation (3.11) defines the threshold $\frac{2}{H_2(\mathbf{p})}$. It states that reconstruction is only possible if $\bar{L} > \frac{2}{H_2(\mathbf{p})}$, otherwise if $\bar{L} < \frac{2}{H_2(\mathbf{p})}$ then reconstruction is impossible. The derivation and proof for this theorem can be found in [8]. Subsequently, the greedy assembly algorithm is successful only if the threshold defined in equation (3.11) is overcome.

Tse et al [8] defines the greedy algorithm based on the contig grouping technique proposed by Lander and Waterman [20]. This algorithm, shown in Algorithm 1, requires the threshold in equation (3.11) to be met. Meeting this bound ensures no gaps, or oceans, are present in the assembled sequence. In other words, if $\bar{L}$, and therefore L, does not satisfy the lower bound $\frac{2}{H_2(\mathbf{p})}$, then there will exist gaps between assembled contigs and the target DNA sequence cannot be fully sequenced. This bound therefore ensures that the required coverage redundancy c is met since the number of islands depends on this coverage depth [20].

The greedy algorithm forms contigs by joining reads together in stages of l according to an overlap score starting from ϕ down to 0. It was shown that the majority of the errors occurred either at stage ϕ or at stage 0 [8]. Errors at stage ϕ are caused mostly by the presence of repeats in the target DNA sequence, while the errors at stage 0 are

Algorithm 1: The High Level Parallel Greedy Assembly Algorithm

```
1. Input the set of length $L$ reads;
2. Initialise all input reads as contigs;
3. Find and merge two contigs with the highest overlap score;
4. Repeat step 3 until no more contigs can be merged;
```

Algorithm 2: The High Level Sequential Greedy Algorithm

```
1. Input the set of length $L$ reads;
2. Initialise the first read in the set as the starting contig;
3. Find a contig and read pair with the largest overlap score and merge them into
   one contig;
4. Repeat step 3 until no more reads can be merged.
```

caused by poor coverage depth. The later case is dealt with by the coverage condition $c_{min}(\bar{L}) = 1$ that requires $\bar{L} > \frac{2}{H_2(\mathbf{p})}$. In order to deal with the repeats, the read length L must also meet the bound imposed by equation (3.4). The greedy algorithm fails if there exist any unbridged repeats.

By merging contigs with the highest overlap score first, the greedy algorithm effectively grows the contigs in parallel, until they all merge and only one contig remains representing the original target DNA sequence. Other contig grouping assemblers such as the SSAKE [22], VCAKE [32] and SHARCGS [24] assemblers, use a sequential greedy algorithm instead [8]. The sequential algorithm varies by growing one contig sequentially by appending reads which have the largest overlap score with the contig. The sequential greedy algorithm is shown in Algorithm 2.

The minimum normalised coverage depth given in (3.9) corresponds to the most optimal assembly possible where the minimum number of reads needed for successful reconstruction is the same as the minimum number of reads needed to ensure coverage of the entire target sequence. A normalised coverage depth given by

$$\bar{c} = \frac{c}{c_{cov}} \tag{3.13}$$

where c_{cov} is given by

$$c_{cov} = \frac{LN_{cov}}{G} \tag{3.14}$$

can hence be used to measure the performance of an assembler relative to the optimal lower bound imposed on the minimum number of reads N_{cov} needed to fully cover the entire target sequence G. This normalised coverage depth is useful for establishing which assemblers can achieve successful reconstruction of the target sequence using the lowest amount of reads.

3.4 k-mer Assembly

The k-mer based algorithms are an alternative to the overlap based algorithms which use the greedy algorithm. At first glance they seem an unintuitive approach to the DNA assembly problem because they further divide reads into smaller k-mer, or k-tuple, subsequences. These k-mer subsequences are then used to construct a de Bruijn graph in order to reconstruct the target DNA sequence. This approach allows the DNA sequence assembly problem to be solved using graph theory techniques [2].

Two graph theoretic approaches to the DNA assembly problem exist [1, 2]. The first creates an overlap graph where all k-mers correspond to a vertex. Two vertices are then connected by an edge if the suffix of one vertex is the same as the prefix of another. This is shown in Figure 1.2 (b). The reconstruction of the target DNA sequence is then done by finding a Hamiltonian path through the graph, where each vertex within the graph needs to be visited exactly once. The issue with this approach is that it does not scale well for a large number of reads as this technique has been shown to be NP-complete [1, 2]. For this reason, a second k-mer technique using de Bruijn graphs was developed. This second approach assigns the k-mer subsequences as edges in the graph instead of vertices as is shown in Figure 1.2 (c). These edges then connect $(k-1)$-mer vertices where the first is a prefix belonging to a k-mer sequence, while the second is the suffix within the same sequence. The assembly process is then achieved by finding the Eulerian path through the graph, where each edge has to be visited exactly once. Compared to the Hamiltonian path problem, the Eulerian path problem is easy to solve because there exist linear-time algorithms for solving this problem [1].

Based on the examples presented in [2], for a small circular genome $ATGGCGTGCA$ with reads $ATGGCGT$, $GGCGTGC$, $CGTGCAA$, $TGCAATG$ and $CAATGGC$ and choosing k=3, the graphs implementing a Hamiltonian and Eulerian path solution can be seen in Figures 3.1 and 3.2 respectively. The k-mers associated with both of these approaches are given as ATG, TGG, GGC, GCG, CGT, GTG, TGC, GCA, CAA and AAT. For the case of the Hamiltonian solution, these k-mers correspond to the vertices present within the graph. By connecting the vertices together with edges based on the overlap between two k-mer subsequences, the Hamiltonian path through the graph can be found. For example, due to the overlap between subsequences ATG and TGG, the edge $ATGG$ is created between the two subsequences. As shown in Figure 3.1, there exists more than one path through the graph. This is due to the small repeated sequences TG and GC present in the genome which causes the ambiguity in the assembly process. In order to deal with this, the edges, or $(k+1)$ subsequence, connecting two vertices need to be verified if they exists within the read set. In doing so the incorrect Hamiltonian path (shown with dotted lines in Figure 3.1) can be eliminated. The target genome is

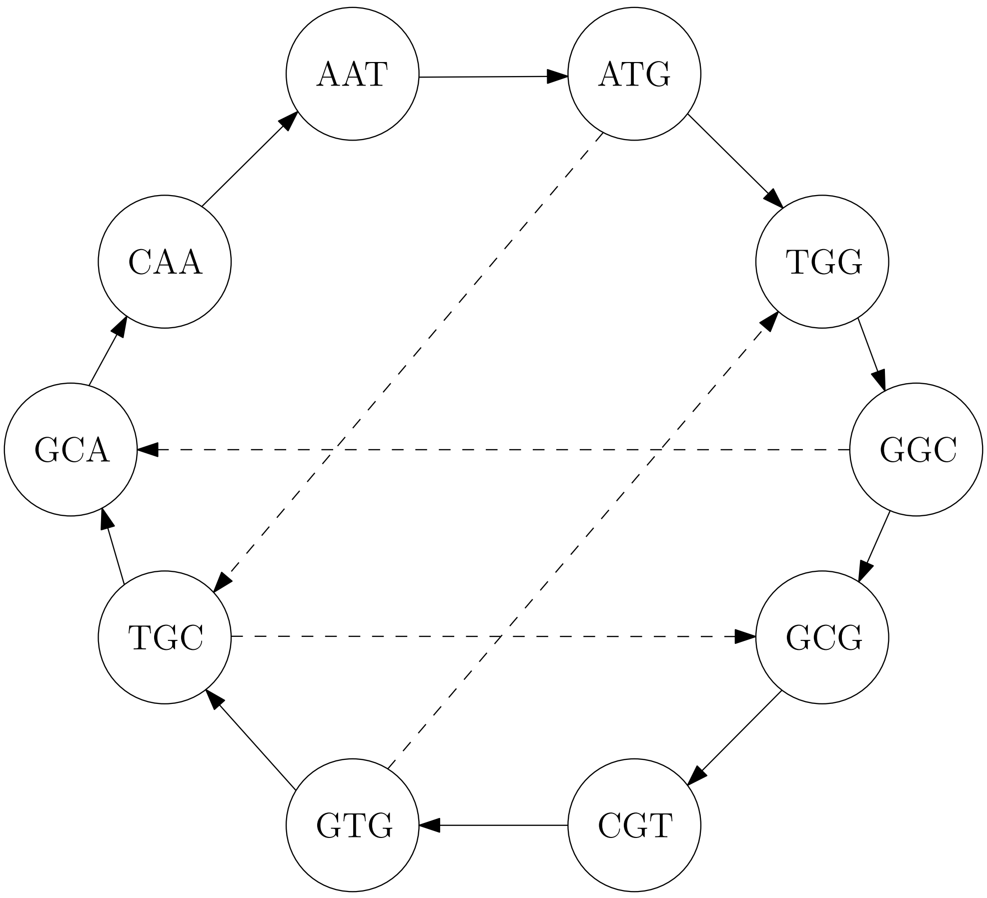


FIGURE 3.1: An overlap graph showing a Hamiltonian path approach to DNA sequence assembly. Adapted from [2]. In this case the k-mers represent vertices within the graph.

then assembled by taking the first letter from each vertex visited within the Hamiltonian path.

Alternatively, the Eulerian de Bruijn graph approach shown in Figure 3.2 represents the k-mers as edges within the graph instead of vertices. In doing so, the vertices of the graph become $(k-1)$-mer subsequences. Two vertices are then connected by an edge if the first exists as a $(k-1)$-mer prefix within the edge, while the second exists as a $(k-1)$-mer suffix within the edge. For example, the vertices AT and TG are connected by the edge ATG. Again, the target genome is then assembled by taking the first letter from each vertex visited from the Eulerian path. It has been shown by Euler that such a Eulerian path exists within a connected directed graph, such as in Figure 3.2, if and only if it is balanced [2]. In other words, a Eulerian path exists if and only if the number of edges entering each vertex within the graph equals the number of edges exiting the vertex. This condition was first discovered by Euler when trying to solve the “seven bridges of Königsberg” problem [2]. However, due to the presence of repeats, certain edges within the graph will be visited multiple times (a Chinese postman problem). This Chinese postman problem can be easily converted into a Eulerian path problem by

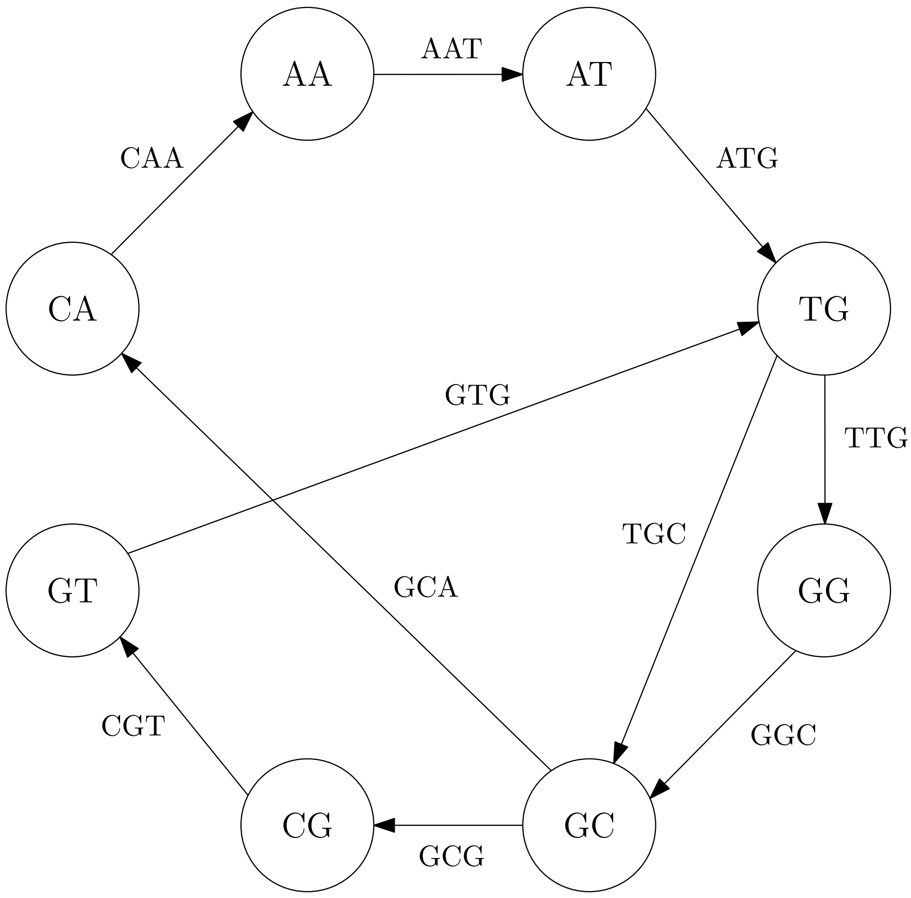


Figure 3.2: A de Bruijn graph showing the Eulerian path approach to DNA sequence assembly. Adapted from [2]. In this case the k-mers represent edges within the graph.

simply introducing a multiplicity l to the edges within the graph [1]. Where l are the number of times an edge is visited in the Chinese postman path. This Eulerian path problem approach to DNA sequence assembly has been implemented by the EULER algorithm discussed in the literature [1].

When breaking down the reads of length L into smaller length k-mers, some information about the target DNA is lost. This loss is minimal if a large enough value for k is chosen. This loss is also compensated for by the performance increase from using the de Bruijn graph approach. As is shown in Tse et al [8], there are some conditions imposed on the size of k. Firstly, k should be large enough to bridge all repeats within the target DNA [8, 34]. This will ensure that a Eulerian path connects k-mers that physically overlap. Similarly, as with the bound imposed on the read length L in the greedy algorithm, the bound on k is determined by

$$\frac{k}{\log_2 G} > \frac{2}{H_2(\mathbf{p})} \tag{3.15}$$

where $H_2(\mathbf{p})$ is the Renyi entropy defined by (3.12) [8]. The second condition requires all successive reads to overlap by at least k base pairs [8]. This implies that successive reads need to have a spacing less than $L-k$ base pairs. Using the Poisson approximation as in [20], the expected number of successive reads with spacing $L-k$ is shown to be $Ne^{-\lambda(L-k)}$. In order to ensure this value is small, the following is used

$$N > \frac{G \ln G}{L-K} \tag{3.16}$$

and hence by using Equations (3.10), (3.15) and (3.16), the following is obtained

$$\frac{N}{N_{cov}} > \frac{\bar{L}H_2(\mathbf{p})}{\bar{L}H_2(\mathbf{p})-2} \tag{3.17}$$

This is the minimum normalised coverage depth required for the successful implementation of the k-mer de Bruijn graph technique [8, 34].

The EULER assembly algorithm recovers information lost from breaking reads into k-mers by using read-paths within the graph. These read-paths correspond to the paths each read creates through the graph. The EULER algorithm, shown in Algorithm 3, attempts to find the Eulerian path through the graph that is consistent with all read-paths, which is analogous to the Eulerian superpath problem [1]. A problem when attempting to find the Eulerian superpath arises due to the presence of repeats within the de Bruijn graph as these repeats cause ambiguity. A path which visits the edges connecting the vertices v_1, v_2,...,v_n, is regarded as a repeat if *indegree*$(v_1) > 1$ and *outdegree*$(v_n) > 1$. Where *indegree* and *outdegree* are the number of edges entering and exiting a vertex respectively. Figure 3.3 shows an example of a repeat along with the ambiguity associated with it. As can be seen, it is unclear whether the superpath consists of subpaths R_1 and R_4 or R_3 and R_4. In this particular case, the read-path R_5 helps to resolve this ambiguity. Pevzner highlighted that when repeats are not fully covered by read-paths, multiple Eulerian paths through a de Bruijn graph exist [42].

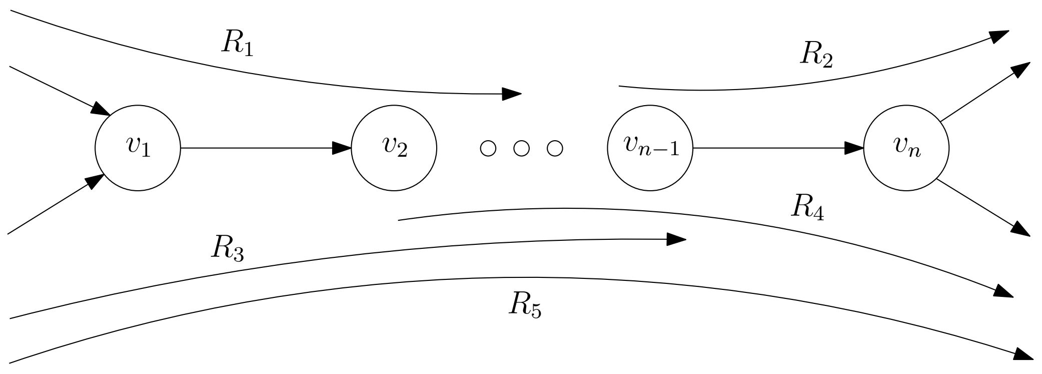


FIGURE 3.3: An example of a repeat path within the de Bruijn graph and a set of overlapping read-paths which help remove the ambiguity associated with the repeat. Adapted from Pevzner et al [1].

Algorithm 3: The High Level EULER path finding algorithm within a de Bruijn graph.

```
1. Pick a starting vertex at random;
2. Pick an outgoing edge consistent with the read-path information for which to
   exit the vertex;
3. Traverse the selected edge to move to the next vertex;
4. Remove the traversed edge from the graph;
5. Repeat 2, 3 and 4 until the final vertex is reached and the Eulerian path is found.
```

3.5 Supervised Neural Networks and the Backpropagation Algorithm

Neural network systems are a popular and powerful machine learning technique which learns from data in order to perform a specific task [36]. A supervised neural network system is trained according to a set of training data $(\mathbf{X_1}, \mathbf{T_1}), (\mathbf{X_2}, \mathbf{T_2}), \cdots, (\mathbf{X_n}, \mathbf{T_n})$, where for a given input $\mathbf{X_n} = \{x_1, x_2, \cdots, x_d\}$, the system is told what output, or target $\mathbf{T_n} = \{t_1, t_2, \cdots, t_k\}$, to expect. How well the system performs is based on how well the system output $\mathbf{Y_n} = \{y_1, y_2, \cdots, y_k\}$ approximates $\mathbf{T_n}$. One can think of a neural network as a system consisting of multiple layers of more primitive machine learning units known as perceptrons. Perceptrons are used to perform simple linear classification, however when combined together into multiple layers, they may be used to perform more complex non-linear classifications [36]. This concept is what makes the neural network, also known as a multilayer perceptron system, so powerful.

On its own, each unit within the network performs a weighted sum of its inputs in order to produce an output. In a regular feed-forward neural network, inputs to a particular unit are always obtained from units belonging to the previous layer within the network. Hence, the outputs from the neural network system are obtained by propagating the system inputs according to a simple product summation at each unit. This sum, also referred to as an activation, is defined as

$$output_m = \sum_i w_{im} * input_i + bias \tag{3.18}$$

where w_{im} is the weight associated with an input to a particular perceptron m within a layer. The machine learning process, or training, of a supervised neural network system is achieved through the modification of such system weights. This is done by completing forward and backward propagation, or passes, of information within the system. The first, or forward pass, produces the outputs $\mathbf{Y_n}$ for a given system input $\mathbf{X_n}$. The second,

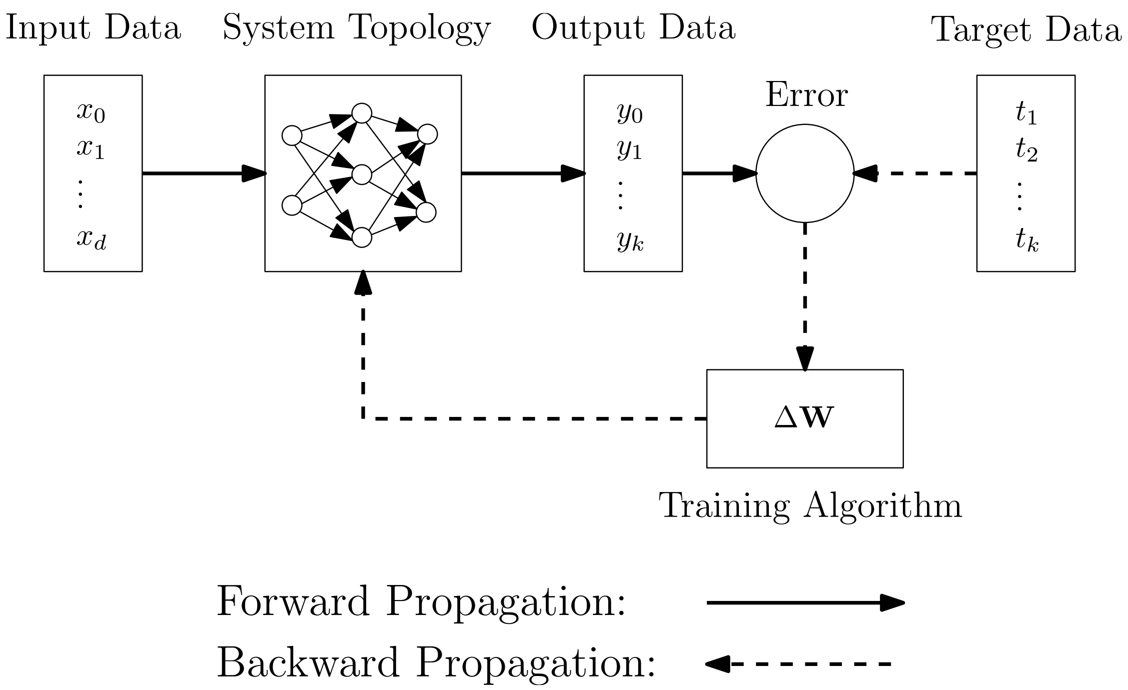


FIGURE 3.4: The supervised machine learning training process applied to a neural network.

or backward pass, modifies the weights within the system based on the error obtained by comparing the system output $\mathbf{Y_n}$ with the target $\mathbf{T_n}$. These steps are then repeated N times with different inputs from the training set $\mathbf{X_n}$. The goal is that after training has been performed, the system is be able to generalise for inputs which were not necessarily included in the training set. The neural network training process is shown in Figure 3.4 and also defined in Algorithm 4.

An example of a three layer feed-forward neural network is shown in Figure 3.5. This topology consists of $(I + 1)$ input layer units, $(J + 1)$ hidden layer units, and K output layer units. Equation (3.18) is applied at each neuron to propagate forward all inputs in

Algorithm 4: Feed-forward neural network training algorithm.

```
1. Define the network topology;
2. Initialise all system weights;
3. Perform forward pass;
4. Perform backward pass;
5. Update weights;
6. Repeat steps 3, 4 and 5 $n$ times in order to train the system.
```

order to obtain the system outputs. However, for the hidden and output layer units, an additional non-linear transformation known as an activation function is performed on the activation from each unit. This function limits the outputs according to a specific bound. A commonly used activation function is the sigmoid function given as

$$\sigma(a) = \frac{2}{1+e^{-\beta a}} - 1 \tag{3.19}$$

which in this case limits the outputs, or activation, between 1 and -1. With equations (3.18) and (3.19) in mind, the unit activations from the hidden and output layers are calculated according to

$$z_j = \sigma\left(\left(\sum_{i=1}^{I} w_{ij}x_i\right) + w_{0j}\right) \tag{3.20}$$

$$y_k = \sigma\left(\left(\sum_{j=1}^{J} w_{jk}z_j\right) + w_{0k}\right) \tag{3.21}$$

where z_j is a hidden layer activation, y_k is an output layer activation and $\sigma(*)$ is the sigmoid function. The w_{0j} and w_{0k} terms in equation (3.20) and (3.21) represent a bias term in the summation. If $x_0 = 1$ and $z_0 = 1$ is set, then the overall neural network system forward propagating equation, as seen at the output, can be rewritten as

$$y_k(\mathbf{X}, \mathbf{W}) = \sigma\left(\sum_{j=0}^{J} w_{jk}\sigma\left(\sum_{i=0}^{I} w_{ij}x_i\right)\right) \tag{3.22}$$

Where $\mathbf{X}$ is the input vector and $\mathbf{W}$ is the weight vector containing all weights from each layer. Based on the modification leading to equation (3.22), equation (3.18) can be rewritten as follows

$$a_m = \sum_i w_{im}z_i \tag{3.23}$$

where a_m represents the activation output at a unit m within any layer of the neural network. In this case z_i may either represent an activation from another unit in a previous layer, or an input to the neural network from the input set $\mathbf{X}$. Hence, using a sigmoid function such as the one defined in (3.19), the following is obtained

$$z_m = \sigma(a_m) \tag{3.24}$$

again, a_m may represent the activation output from a unit within any of the layers.

The backward pass of the neural network involves changing the weight vector in order to train the system. A popular approach in which to modify the weight vector is with the gradient descent optimisation method together with some error calculation [36]. The gradient descent approach consists of small changes in $\mathbf{W}$ until a minima, or optimal

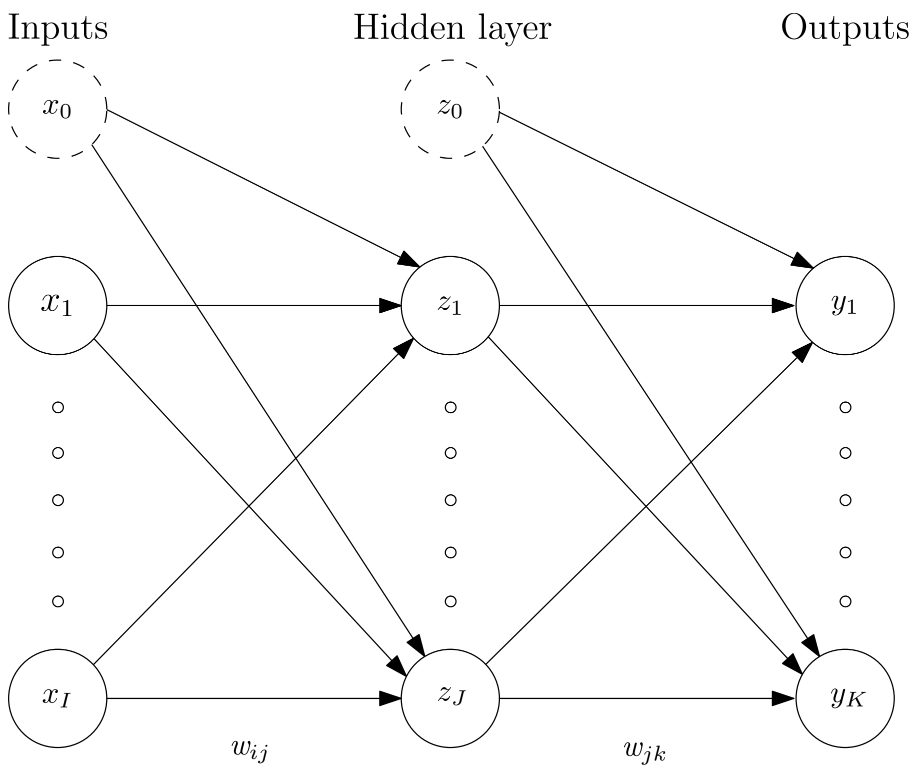


FIGURE 3.5: A three layer feed-forward neural network with I input neurons, J hidden neurons and K output neurons.

value, is found. The gradient descent formula is given as

$$\mathbf{W}^{(\theta+1)} = \mathbf{W}^{(\theta)} - \eta \nabla E(\mathbf{W}^{(\theta)}) \tag{3.25}$$

where η is some learning rate and θ is an iteration, or epoch number, used to distinguish between one forward and backward propagation instance from another. A popular error function is the squared error given as

$$E(\mathbf{W}^{(\theta)}) = \sum_{k=1}^{K} \left(y_k(\mathbf{X}^{(\theta)}, \mathbf{W}^{(\theta)}) - t_k^{(\theta)} \right)^2 \tag{3.26}$$

where $\mathbf{X}^{(\theta)}$, $\mathbf{W}^{(\theta)}$ and $t^{(\theta)}$ are values specific to each epoch. When using gradient descent optimisation, it is important to observe how this error changes with respect to the system weights. The derivative of the error with respect to w is given as

$$\nabla E := \frac{\partial E}{\partial w_{ij}} = 2\left(y_j - t_j\right) x_i \tag{3.27}$$

An efficient way in evaluating the gradient of an error function for a regular feed-forward neural network is with the use of the backpropagation algorithm. Backpropagation makes use of the chain rule which allows us to redefine equation (3.27) as

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial a_j}\frac{\partial a_j}{\partial w_{ij}} \tag{3.28}$$

By using useful notation along with equations (3.23) and (3.24), the following can be written

$$\frac{\partial E}{\partial a_j} = \delta_j \tag{3.29}$$

$$\frac{\partial a_j}{\partial w_{ij}} = z_i \tag{3.30}$$

Substituting these into (3.28) the following is obtained

$$\frac{\partial E}{\partial w_{ij}} = \delta_j z_i \tag{3.31}$$

Equation (3.31) indicates that in order to obtain the required derivative, the δ on the output end of the weight is multiplied with the activation z on the input end of the weight. Each δ within the network is required to be calculated in order to obtain the change in weights. This is achieved with the use of the following backpropagation formula [36]:

$$\delta_j = \sigma^{'}(a_j)\sum_k w_{jk}\delta_k \tag{3.32}$$

Hence in order to calculate δ in a particular layer, all the δ values in the preceding layer need to be known. For this reason backpropagation starts at the output, where δ_k can be calculated from the system output and the target, and works backwards to calculate the hidden layer δ values throughout the network. This concept is illustrated in Figure 3.6. As shown in [11] and [36], δ_k at the output layer units can be calculated using

$$\delta_k = \sigma^{'}(a_j)2(y_k - t_k) \tag{3.33}$$

Once all necessary δ values have been calculated using equations (3.33) and (3.32), the weights within the network can then be modified. In the case of the three layer feed-forward neural network system, the weights connecting the hidden and output layers are calculated using

$$w_{jk}^{(\theta+1)} = w_{jk}^{(\theta)} - \eta\delta_k z_j \tag{3.34}$$

while the weights connecting the input and the hidden layers are calculated using

$$w_{ij}^{(\theta+1)} = w_{ij}^{(\theta)} - \eta\delta_j x_i \tag{3.35}$$

This modification of the system weights is repeated N times until a weight vector $\mathbf{W}$ producing minimal error at the output layer is found.

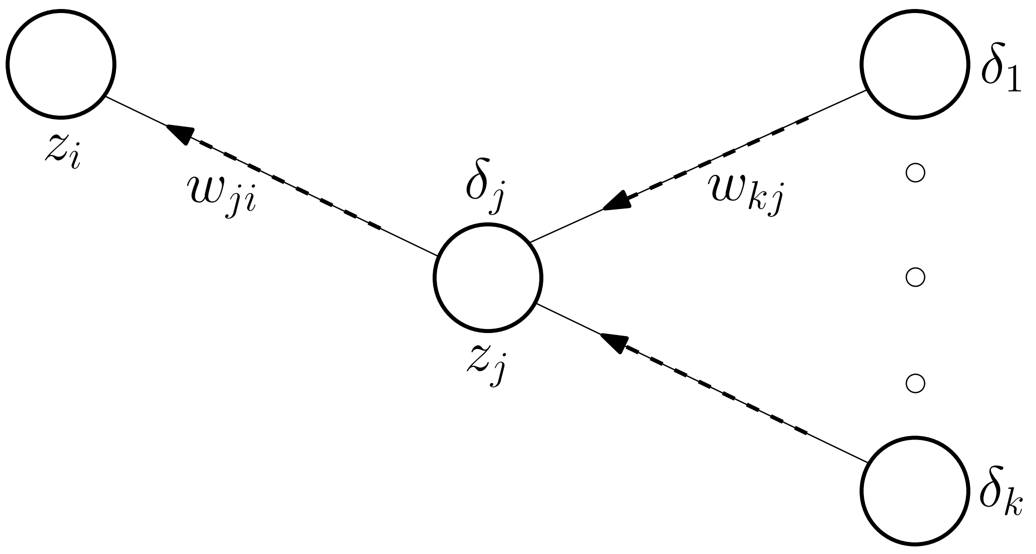


FIGURE 3.6: Illustration of the backward propagation flow and the δ values associated with each unit within the neural network topology.

In the case of a recurrent neural network, the hidden layer units are connected with other units within the same layer as shown in Figure 3.7. These connections, unlike the regular feed-forward connections, are applied with an epoch delay. This allows for information from previous epochs to affect the current output of the system. This introduction of memory into the system will hence change the mechanics within the system. In order to accommodate for this change, the forward and backward propagations need to be modified. In a regular feed-forward network, the system is trained by applying the forward passes and then backward passes N times. However, the training of a recurrent neural network requires that all N forward passes occur before the N backward passes are implemented. It is therefore necessary to modify the standard backpropagation algorithm in order to accommodate for memory. This new algorithm, known as the backpropagation through time algorithm [37], is shown in Algorithm 5. In order to implement this algorithm some modifications also need to be made to existing equations. Firstly, in the forward pass, the unit summation within the hidden layer given in (3.20) changes to

$$a_j^{(\theta)} = \sum_{i=0}^{I} w_{ij} x_i^{(\theta)} + \sum_{i=0}^{J} w_{ij} z_i^{(\theta-1)} \tag{3.36}$$

where θ in this case indicates the current input from the input set $\mathbf{X}$. $z^{(0)} = 0$ is set for $\theta = 1$. Secondly, when processing the backward passes, equation (3.32) needs to change in order to accommodate for memory as follows

$$\delta_j^{(\theta)} = \sigma'(a_j^{(\theta)}) \left(\sum_{i=1}^{K} w_{ij} \delta_i^{(\theta)} + \sum_{i=0}^{J} w_{ij} \delta_i^{(\theta+1)} \right) \tag{3.37}$$

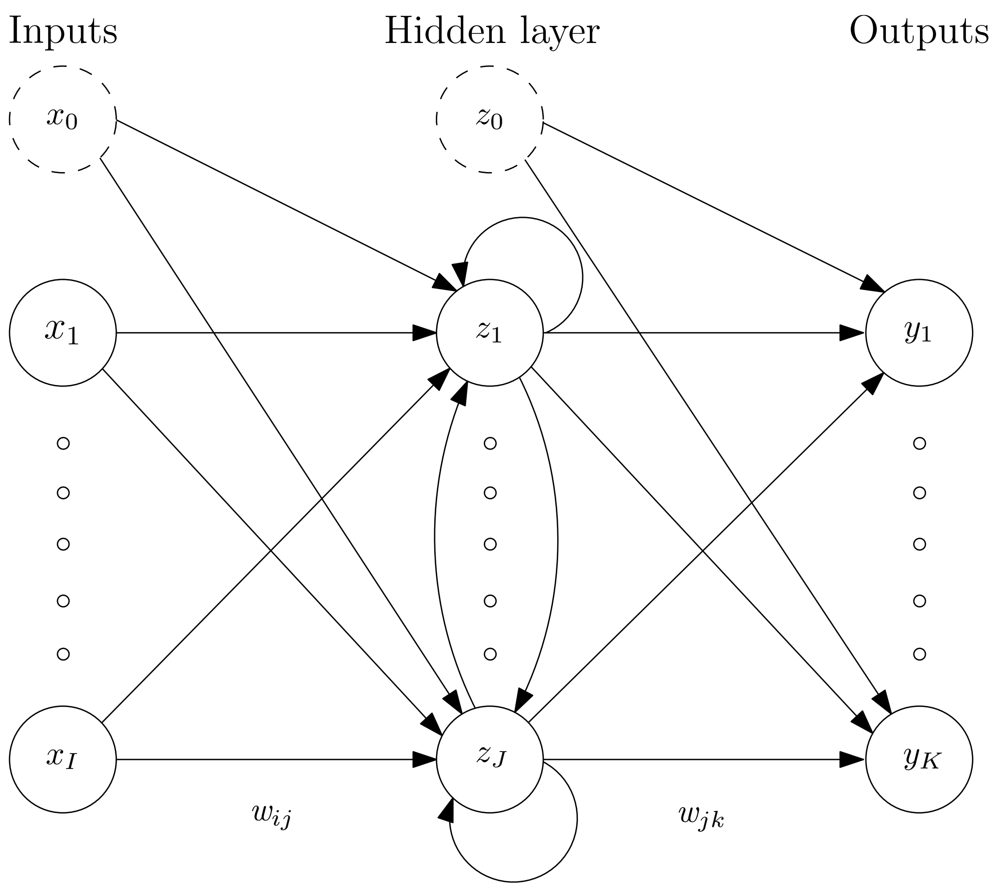


FIGURE 3.7: Three layer recurrent neural network with I input neurons, J hidden neurons and K output neurons. The connections between hidden nodes implement a time delay of 1.

Additionally, the weights connecting the units within the hidden layer also need to be modified according to

$$w_{ij}^{(\theta+1)} = w_{ij}^{(\theta)} - \eta \delta_j z_i^{(\theta-1)} \tag{3.38}$$

where in this case both i and j are indices of J (the number of hidden units).

The drawback to this recurrent approach is the requirement of extra memory in order to store intermediate information. The N forward passes need to be processed and the outputs stored to memory in order for the N backward passes to have enough

Algorithm 5: Recurrent neural network training algorithm.

```
1. define the network topology;
2. initialise all system weights;
3. perform the forward passes $N$ times;
4. perform the backward passes and update the weights $N$ times.
```

information. When very large and complex network topologies are used, this need for storing memory does not scale well. However, for smaller topologies, the ability to better predict future values based off previous patterns is beneficial for the tracking of DNA sequences.

3.6 Summary

The information presented in this chapter is used as a foundation for the rest of this dissertation. An information theoretic analysis was performed in order to determine if assembly is achievable and under which conditions. A k-mer approach to the DNA assembly problem was presented which promises to significantly reduce computational complexity in the assembly process. Additionally, a background into neural networks and the backpropagation training algorithm was presented in order to support Chapter 5 which deals with a machine learning approach to the DNA assembly problem.

Chapter 4

The Greedy and de Bruijn Assembly Schemes

In this chapter, the methodology of the greedy and de Bruijn assemblers is presented and discussed. These two assemblers are based on existing assemblers presented in the literature review in Chapter 2. The greedy assembler makes use of a greedy overlap assembly algorithm inspired by the CAP series of assemblers [9, 10], while the de Bruijn assembler, inspired by the EULER assembler [1], makes use of de Bruijn graphs and a simple path finding algorithm to perform DNA sequence assembly. These two assemblers have been implemented in this research in order to highlight and compare the differences in their computational complexity and assembly accuracy. Additionally, they were selected in order to investigate their performance when combined with a machine learning grouping approach (which is discussed in Chapter 5).

4.1 The Greedy Assembler

The greedy assembler implements a greedy and naive strategy to the DNA sequencing problem. The greedy algorithm is a brute force approach at assembly by performing a pairwise comparison between all possible read sequences until a matching pair is found. It searches for areas of similarity between reads by comparing the prefix and suffix of reads with one another. When two reads containing an area of overlap is found, the two reads are merged together to form a larger contig. This overlap merging procedure is depicted in Figure 4.1. Unlike the modern overlap assemblers discussed in Chapter 2, the greedy assembler does not implement any error correcting or consensus steps. By not implementing these extra steps in the assembly algorithm, the focus remains solely on the overlap assembly step and how it is affected by the read grouping strategy discussed

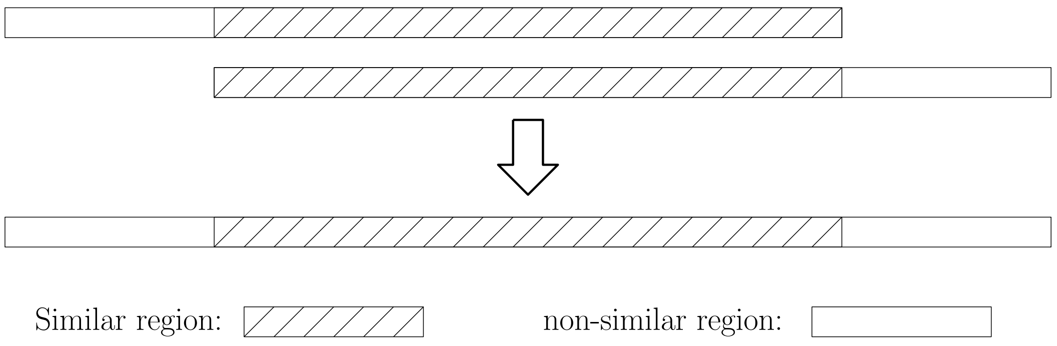


FIGURE 4.1: The contig overlap merging process shows how two similar matching reads are merged together into one contig.

later in Chapter 5. For this reason, the greedy assembler establishes a performance baseline with which to compare the rest of the assembly strategies implemented in this research.

The greedy algorithm is implemented in a sequential manner, growing a single read, or contig. When this is no longer possible, the contig is stored and a new contig is selected for assembly. This process is repeated until no reads remain. For each possible contig-read comparison, the overlap between the two is varied until either a match is found or until a minimum threshold is reached. The overlap fraction ϕ varies according to

$$\phi_{min} < \phi < \phi_{max} \tag{4.1}$$

where the upper bound, or maximum overlap, of ϕ is limited by the size of the read as follows

$$\phi_{max} = L - 1 \tag{4.2}$$

The lower bound ϕ_{min}, or minimum overlap, is an important factor affecting the performance of the greedy assembly algorithm and needs to be carefully chosen. This factor corresponds to the stringency of the assembly process. If this threshold is set too high, the similarity criteria becomes too stringent, resulting in potentially similar contig-read pairs not being merged together. On the other hand, if the threshold is set too low, there will be an increased chance for erroneous merges at very low overlap instances. The probability for erroneous mergers between a contig-read pair calculated in [8] is given by

$$P_{erroneous\ merger} = 2^{-\phi H_2(p)} \tag{4.3}$$

This shows that as the overlap ϕ decreases, the probability for erroneously merging an overlapping contig-read pair increases. It is therefore important to set ϕ_{min} at a large enough value in order to prevent comparing contig-read pairs at smaller overlaps. In the work by Lander and Waterman, it was shown that selecting ϕ_{min} to be 20% the size of the maximum overlap ϕ_{max} produced good results [20]. For this reason, the minimum overlap used for comparing contig-read pairs in the greedy algorithm was set to $\phi_{min} = 0.2\phi_{max}$. This ensures that contig-read pairs are not erroneously merged when the probability of error as shown in (4.3) is large.

Establishing the similarity between reads and the assembly contig at each overlap value is performed using a correlation function. This function implements a sliding window comparison between two sequences. At each overlap position, an overlap score is recorded for each base pair found to match between the two sequences. A correlation factor, given as

$$correlation\ factor = \frac{overlap\ score}{max\_size\{sequence1, sequence2\}} \tag{4.4}$$

is used to quantify the similarity between two sequences. In addition to being used as a measure for similarity between the overlapping regions of contig-read pairs, the correlation function is also used to determine if a particular sub-sequence exists within a larger sequence. This process is displayed in Figure 4.2. The correlation function is used to filter out duplicate reads in the greedy assembly algorithm as well as to compare the final assembled contigs to the known target sequence. In the case of the greedy assembler, the function is always used to compare sequences of the same size

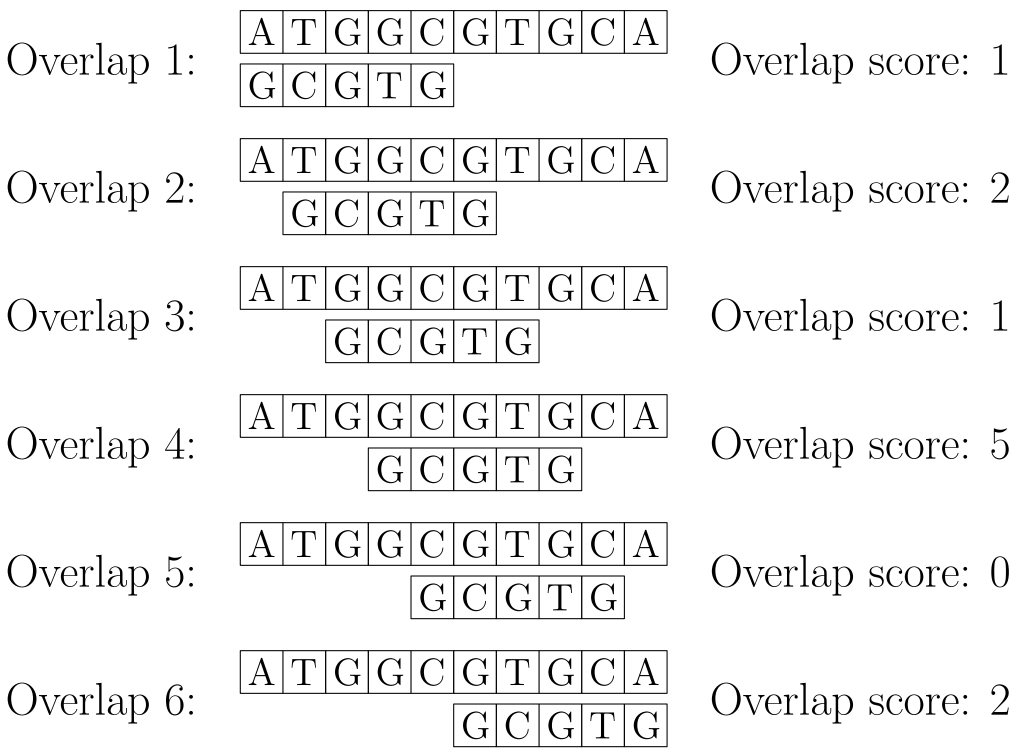


FIGURE 4.2: Example of the correlation process comparing two different sized sequences together. The smaller sequence is compared in a sliding window fashion to the larger sequence and the overlap score is recorded.

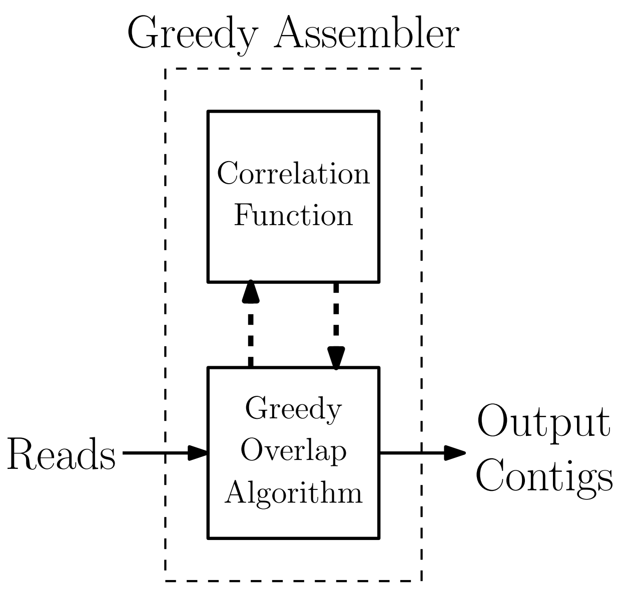


FIGURE 4.3: Block diagram of the greedy assembler showing the assembler components.

(determined by the current overlap value) which produces only a single possible overlap score. By merging only contig-read pairs which generate correlation factors over 95%, the greedy assembler incorporates an error tolerance in the assembly process. Having a high error tolerance ensures that overlapping sequences will not be erroneously merged, while still tolerating the occasional single base pair error caused by erroneous reads generated by the sequencing process. The block diagram in Figure 4.3 shows the components used by the greedy assembler. The reads provided as an input to the greedy assembler are processed by the greedy overlap matching algorithm presented in Algorithm 6. The greedy algorithm then makes use of the correlation function to find matching contig-read pairs in order to produce the final assembled output contigs.

Analysis of the greedy algorithm shows that it is of $O(N^2)$ complexity. For each assembly contig, the algorithm has to search through N reads in order to find a contig-read pair with the highest matching overlap. When no matches are found, a new assembly contig is selected from the remaining N reads. This validates the analysis done by Tse et al [8]. This $O(N^2)$ complexity is a major drawback to the greedy assembly algorithm and is responsible for the introduction of other assembly algorithms which attempt to reduce this complexity.

A further pitfall to this algorithm is its poor performance when dealing with repeats within the target sequence. The algorithm is greedy because it merges the largest overlapping contig-read pair it finds. This may not always be the correct solution. An example showing this problem can be seen by studying Figure 4.4. In this example, two reads are found to overlap with the current assembly contig which covers region A and the first repeat region. The correct assembly scenario would be to merge the read

Algorithm 6: The greedy overlap algorithm implemented by the greedy assembler.

```
Select first read from read pool, assign it as a contig and remove it from the pool;
while read pool is not empty do
    Set the recorded largest overlap to zero;
    for read in read pool do
        Determine the maximum and minimum possible overlap bounds relative to the
        current contig and read sizes;
        for overlap from maximum bound to minimum bound do
            Determine prefix and suffix overlap scores using correlation function;
            if correlation factor is over 95% and overlap is larger than current recorded
            largest overlap then
                Record the matching overlap as largest overlap;
    Merge contig and read with the largest recorded matching overlap;
    if no match found then
        Store current contig;
        Select first read from read pool, assign it as a contig and remove it from the
        pool;
```

covering region B with the assembly contig. However, because the read covering region C is found to have a higher overlap with the assembly contig, it is erroneously merged instead.

4.2 The de Bruijn Assembler

The de Bruijn assembler makes use of graph theoretic principles which approach the DNA sequence assembly problem differently in order to reduce complexity. It is based on the work by Idury and Waterman [33] and Pevzner [19]. The assembler takes an unintuitive approach to solving the problem by first breaking down the reads generated in the sequencing process into smaller pieces. These smaller k-mers are used as building blocks to construct the de Bruijn graph, discussed in Chapter 3, which maps the relationships between reads. The assembler then reconstructs the target sequence by attempting to find a Eulerian path through the graph. Finally, the assembler implements a greedy overlap algorithm, the same as the one used in the greedy assembler, in order to piece together the final set of contigs generated by the de Bruijn path finding algorithm. The de Bruijn assembler block diagram presented in Figure 4.5 shows the components within the de Bruijn Assembler.

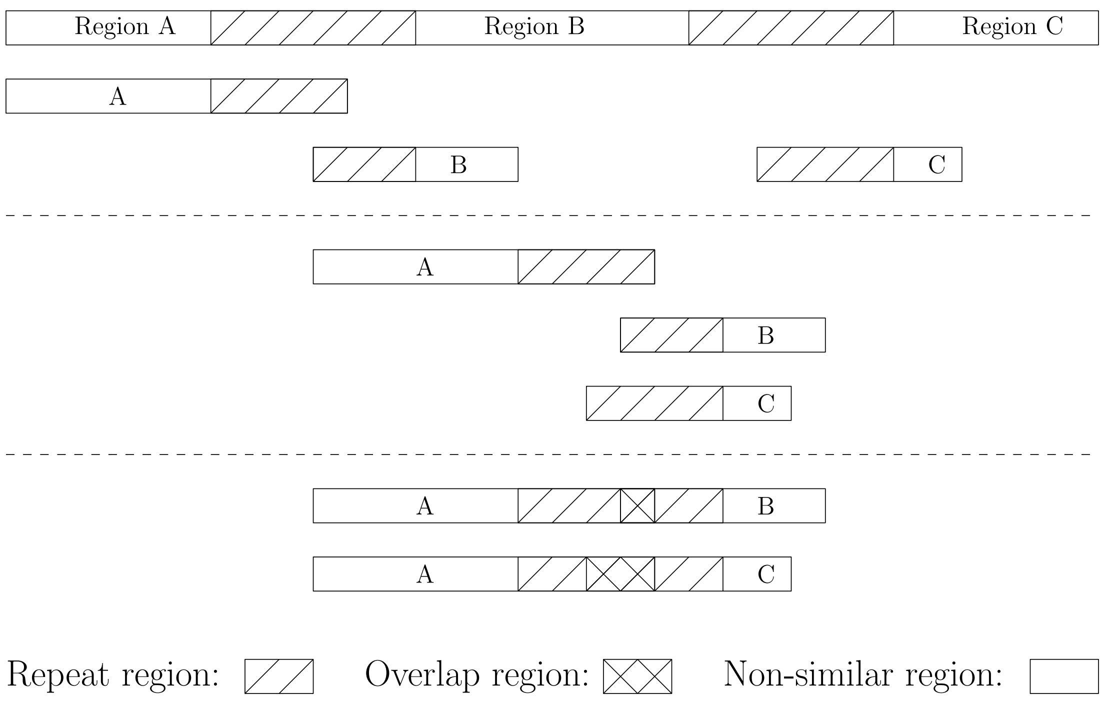


FIGURE 4.4: Erroneous merging of contig-read pair due to the presence of repeat regions within the target sequence. In this example, the contig containing region A should be merged with the read containing region B. However, due to the repeat regions, erroneous merging may occur by merging the read containing region C with the contig instead.

Algorithm 7 presents the graph construction process. This includes generating all the edges and vertices extracted from the reads as well as finding all possible starting vertices within the graph. Each k-mer subsequence corresponds to an edge within the graph, while each (k-1)-mer corresponds to a vertex. Each read therefore corresponds to a small subsection, also known as a read-path, of the de Bruijn graph. Information from these read-paths is then used when creating the de Bruijn graph. The number of times each

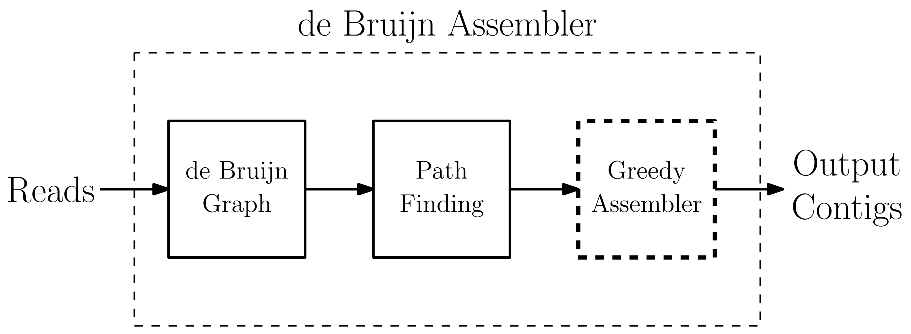


FIGURE 4.5: Block diagram of the de Bruijn assembler showing the assembler components. The de Bruijn assembler implements an instance of the greedy assembler in order to piece together multiple overlapping output contigs obtained from the path finding algorithm, if they exist.

k-mer edge subsequence is found within the reads, and from which read it is found, is recorded. This information is then used to help determine the correct Eulerian path through the de Bruijn graph. Studying Algorithm 7 shows that creating the de Bruijn graph from a set of reads is of order $O(N)$ complexity. Generating a (k-1)-mer subsequence is of order $O(L-(k-1))$ while finding the starting vertices involves searching through all vertices, which is bound by $O((L-k)N)$ [34]. The total complexity of the graph construction process is therefore of order $O((L-k)N)$.

The path finding algorithm responsible for finding the Eulerian path through the constructed de Bruijn graph can bee seen in Algorithm 8. Finding a Eulerian path within a directed graph requires that all vertices are balanced. In other words, the number of input edges to a vertex must equal the number of output edges [2]. This is true for all vertices except for the starting and ending vertices. Finding the starting vertex for the Eulerian path is important for finding the entire Eulerian path. A path is created by starting at each vertex in the graph which has an *indegree* smaller than its *outdegree*. The path continues traversing the graph visiting each edge only once and ends when it reaches a vertex which has no more traversable edges. Multiple disjoint paths within the graph may be found if there are gaps in the coverage of the target sequence. Erroneous reads may also create extra starting and ending vertices. These erroneous vertices negatively affect the path finding algorithm by causing it to select incorrect starting locations or terminate prematurely. For this reason, multiple sub-paths within the graph may be found for each possible starting vertex. The complexity associated with the path finding algorithm is at most of order $O((L-k)N)$ since every traversed edge belonging to every

Algorithm 7: The de Bruijn graph construction algorithm.

```
for each read in read pool do
    for each (k-1)-mer within read do
        if vertex with (k-1)-mer sequence does not exist then
            Create vertex with ($k$-1)-mer subsequence;
            Add leading $k$-mer subsequence as an input edge to the vertex;
            Add lagging $k$-mer subsequence as an output edge to vertex;
        else
            Add leading $k$-mer subsequence as an input edge to the vertex;
            Add lagging $k$-mer subsequence as an output edge to vertex;

for each vertex in de Bruijn graph do
    if number of output edges is greater than the number of input edges then
        Mark vertex as a starting vertex;
```

Algorithm 8: de Bruijn graph path finding algorithm.

```
for each starting vertex do
    Reset all traversed edges;
    while current vertex has traversable edge do
        Append the first base pair of current vertex to the current assembly path;
        Use read-path information to select traversable edge;
        Move to next vertex by traversing selected edge while marking it as traversed;
    Append remaining base pairs within the current vertex to current assembly path;
    Store current assembly path as an output contig;
```

vertex has to be reset when there are multiple starting vertices.

The size of the k-mers relative to the read size L is important when it comes to finding the correct path through the graph when there are repeat regions present in the target sequence. As shown in Figure 1.2 (c) in Chapter 1, these repeat regions create loops within the de Bruijn graph which need to be resolved. The read-paths within the graph provide information on how to resolve these loops. They provide the direction which a path must take when there are multiple output edges at a given vertex along the current assembly path. Selecting a smaller value for k will increase the relative size of these sub-paths compared to the size of each edge, ensuring more information is available to the path finding algorithm. For example, selecting a small k value ensures more k-mers can be found within each read. This increases the number of edges covered by each read and therefore more information is available regarding which path to take within the graph. This can be seen in Figure 4.6. In case (a), the read-paths are too small to cover the entire repeat, and the correct path through the repeat can not be resolved. In case (b), the read-paths do cover the entire repeat, and the correct exit edge from the repeat section is selected based on which read-path the edge entering the repeat region was from.

Erroneous reads produced in the sequencing process are also responsible for creating multiple paths through a de Bruijn graph. Figures 4.7 and 4.8 show how two phenomena, bubbles and tips, are caused from a single substitution error present within a read. Bubbles cause two possible paths through a graph which presents the path finding algorithm with the challenge of deciding the correct one. Tips on the other hand, cause dead ends within a graph to occur. In such cases, the graph is no longer balanced, which causes the path finding algorithm to prematurely terminate and create incomplete assemblies. Bubbles occur from errors present in the center of reads, while tips occur from errors present on the edges of reads. Additionally, tips are more likely to occur when larger k-mers are used relative to the read size. In order to deal with these issues, the

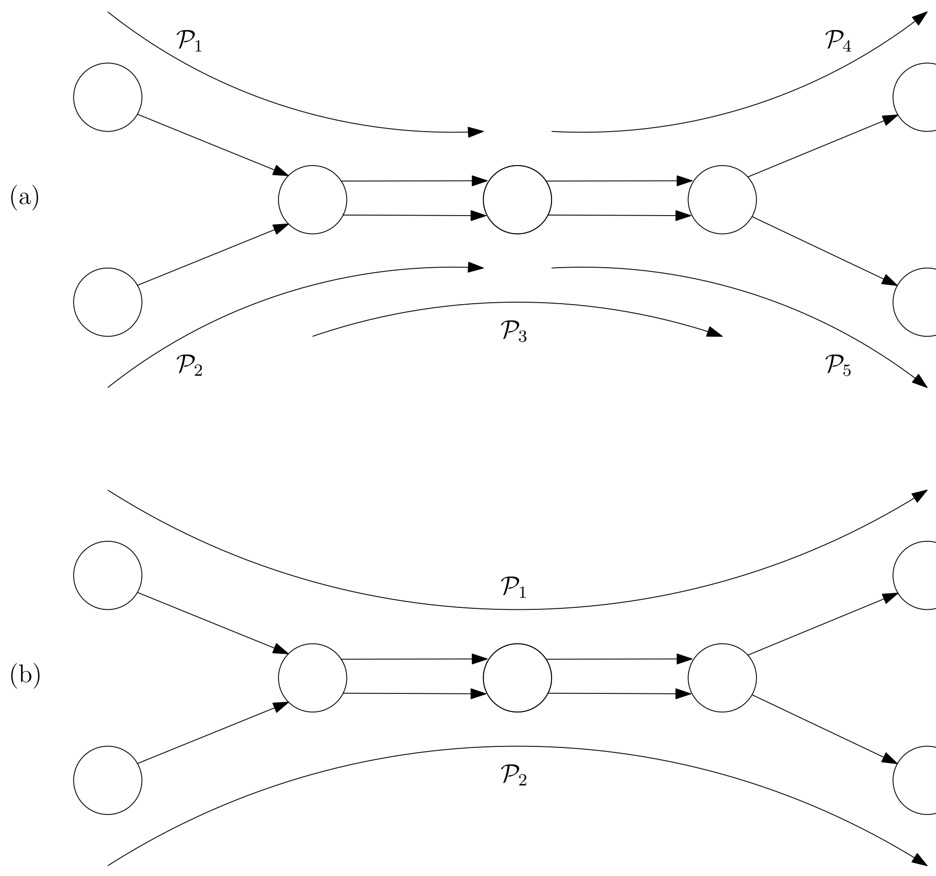


FIGURE 4.6: Read path information used to resolve repeats within the de Bruijn graph.

path finding algorithm makes use of an occurrence rate for each edge when traversing the graph. This information, which is obtained when creating the graph, helps to detect erroneous edges. It was shown in literature [23], that the expected number of occurrences for each edge can be calculated using

$$E(occurance\ rate\ of\ edge) = C\frac{L-k+1}{L} \tag{4.5}$$

Erroneous edges will have an occurrence rate far below the expected value obtained from (4.5) and will therefore be ignored by the path finding algorithm when the next traversable edge is being chosen.

Preliminary experiments were performed in order to determine the optimal value for the size of k. They showed that at read sizes ranging from between $L = 50$ and $L = 500$, the most successful de Bruijn assemblers used k-mers of size $k = 0.4L$. Figure 4.9 shows the coverage performance of an assembler using reads of size $L = 500$ and a coverage depth of $c = 20$. The figure shows that assembly is unsuccessful for higher values of k. The reason for this is due to the read-paths being relatively smaller when compared to the k-mers when large k values are used. This implies that at higher k values, read-paths

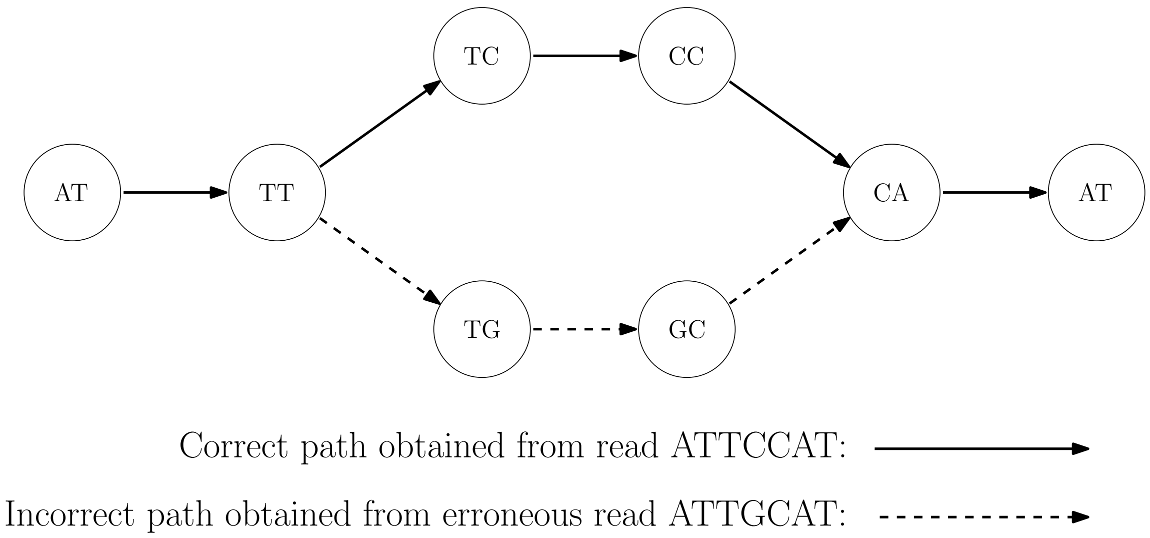

TC
CC
AT
TT
CA
AT
TG
GC
Correct path obtained from read ATTCCAT:
Incorrect path obtained from erroneous read ATTGCAT:

FIGURE 4.7: An example of a bubble within a $k = 3$ de Bruijn graph (k-mers correspond to edges in a de Bruijn graph). The solid line represents the correct path through the graph, while the dashed line represents an erroneous path through the graph.

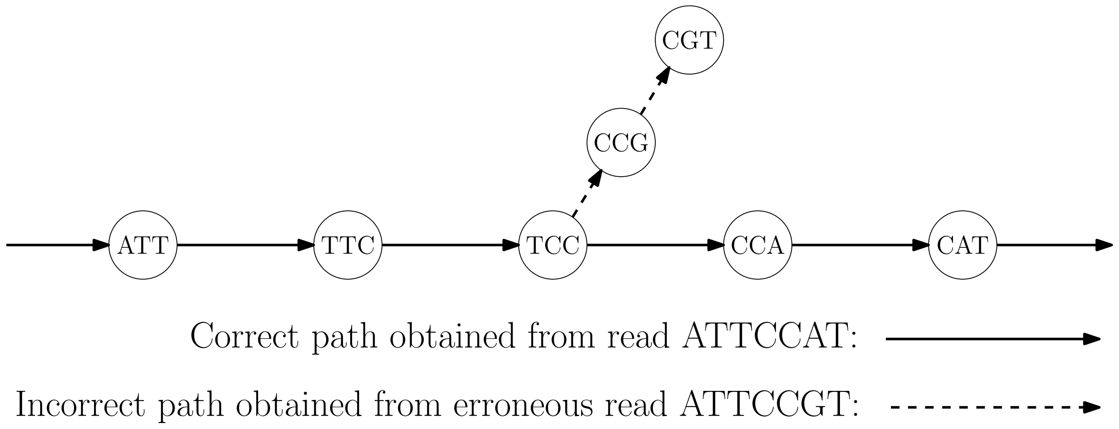

CGT
CCG
ATT
TTC
TCC
CCA
CAT
Correct path obtained from read ATTCCAT:
Incorrect path obtained from erroneous read ATTCCGT:

FIGURE 4.8: An example of a tip within a $k = 4$ de Bruijn graph (k-mers correspond to edges in a de Bruijn graph). The solid line represents the correct path through the graph, while the dashed line represents an erroneous path through the graph.

cover fewer edges and therefore provide less information to the Eulerian path finding algorithm. Conversely, for small values of k, the overlap between adjacent vertices becomes too small to bridge repeats present in the target sequence. This then creates loops within the graph which the Eulerian path finding algorithm can not resolve.

In cases when multiple output contigs are generated at the end of the path finding algorithm, the greedy overlap algorithm is implemented in order to combine them. At the very least, when a single large output contig can not be generated, the path finding algorithm significantly reduces the input to the greedy overlap algorithm. This is because the number of output contigs from the path finding algorithm is significantly less than the number of reads generated by the read sequencing process. In such cases the de

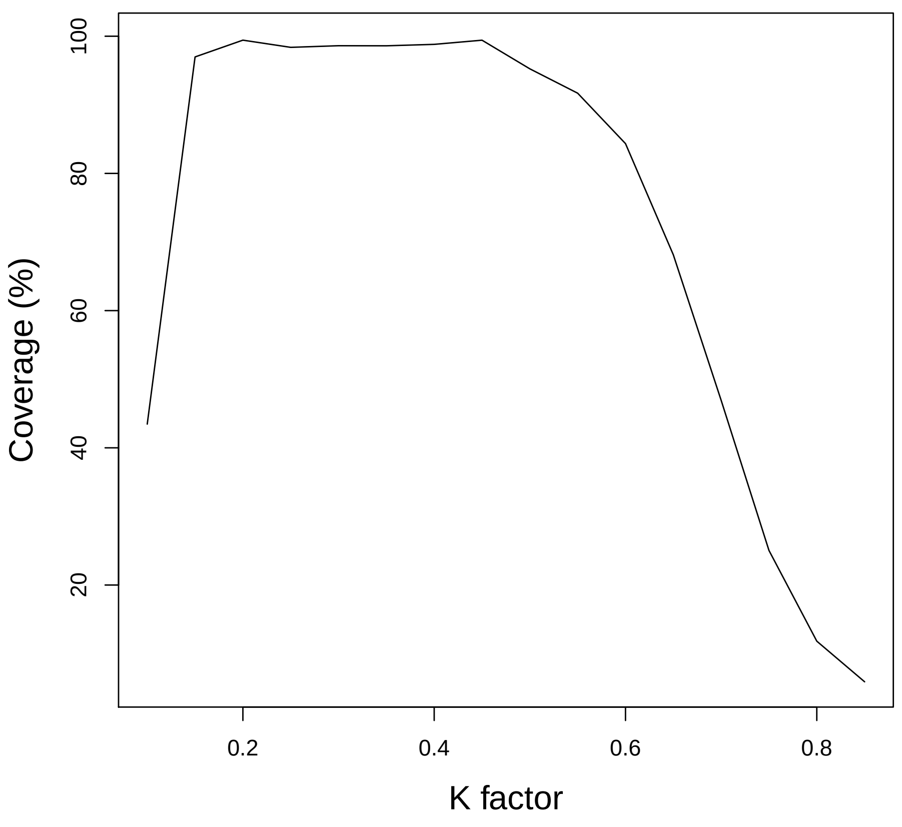


FIGURE 4.9: Assembly performance of a de Bruijn assembler using a read size of $L = 500$ and a coverage depth of $c = 20$ at varying k values.

Bruijn graph approach acts as a filter to the greedy overlap algorithm which reduces the size of the dominating factor affecting its complexity.

4.3 Summary

The greedy and de Bruijn assemblers have been implemented in order to highlight the differences in their computational complexity and assembly accuracy. These assembly strategies were selected in order to investigate the effect of the machine learning grouping approach (discussed in the next chapter) has on the different assembly algorithms used in each assembler.

Chapter 5

The Neural Network Assembly Scheme

This chapter provides a rationale for applying a divide and conquer approach to address the complexity problem associated with comparing large amounts of reads in the DNA sequencing and assembly problem. Secondly, a neural network approach to grouping reads is presented which introduces a new, and more accurate, similarity metric μ for grouping reads when compared to more traditional methods. Lastly, the implementation of the neural network scheme together with the greedy and de Bruijn assemblers is discussed.

5.1 The Complexity and Edit Distance Problem

Modern day applications of DNA sequencing typically target large areas of a genome. It is often necessary to sequence an entire genome, as in whole genome sequencing, where all chromosomes need to be sequenced [35]. The average size of a typical human chromosome is about 1.5×10^8 base pairs [3]. Therefore, depending of the sequencing technology being used, the number of reads (N) needed to accurately assemble a target sequence is between 10^9 to 10^{10}. It was shown in Chapter 3 that N is the dominating factor in the complexity of any assembly algorithm. Reducing the size of N for an assembly algorithm will reduce the time taken to complete the assembly. By introducing a “divide and conquer” approach it is possible to break the assembly problem into multiple and much smaller groups. In doing so, the dominating factor in complexity (N) for the assembly of each group is reduced. For the case of the greedy and de Bruijn algorithms, which have $O(N^2)$ and $O(N \log N)$ complexities respectively [8, 34], this is shown mathematically by the fact that for large N:

$$N^2 >> m \times \left(\frac{N}{m}\right)^2 \quad \textbf{for large } m \tag{5.1}$$

$$N \log(N) >> m \times \left(\frac{N}{m}\right) log\left(\frac{N}{m}\right) \quad \textbf{for large } m \tag{5.2}$$

This approach however introduces a new problem of sorting, or classifying, reads into similar groups. One solution to this classification problem is to use a similarity measure when comparing reads with one another. If reads are considered similar then they are placed in the same group. This same similarity measure is used by common overlap assembly algorithms such as the greedy algorithm. These similarity measures use a metric known as an edit distance ($D(s_1, s_2)$) to measure the difference, and hence the similarity, between two sequences [25]. This metric originates from the work of Levenshtein when comparing binary codes [26]. One can interpret this edit distance as a series of insertion, deletion or substitution transformations γ acting on a sequence s_1 of DNA in order to transform it into another sequence s_2 [11].

$$s_2 = \gamma_i(\ldots \gamma_2(\gamma_1(s_1))) \tag{5.3}$$

The value i represents the number of steps needed to change one sequence into another. Hence, the edit distance between two sequences is then given by

$$D(s_1, s_2) = i \tag{5.4}$$

An issue with this type of similarity measure is that it does not account for the structure of the sequences being compared. The grouping process will place reads in a group if the edit distance is below a certain threshold. However, two reads might be erroneously considered similar if their edit distance is below the threshold and the differences between them are interleaved throughout the reads. This is a problem because overlap assembly algorithms look for overlaps, or similarity, between the ends of particular reads. It is therefore desirable to have reads placed in similar groups only if contiguous regions within the reads are found to be below the edit distance threshold. This principle is depicted in Figure 5.1. Both (a) and (b) show cases which are below the edit distance, however only case (a) has a contiguous region below the edit distance threshold. A classification technique is therefore needed which can measure the similarity in a contiguous fashion by recognizing patterns in the sequences being compared.

The field of machine learning provides pattern recognition techniques capable of performing classification [36]. One such technique makes use of neural networks to achieve this [11]. Using a neural network, a new measure for similarity is provided which will help

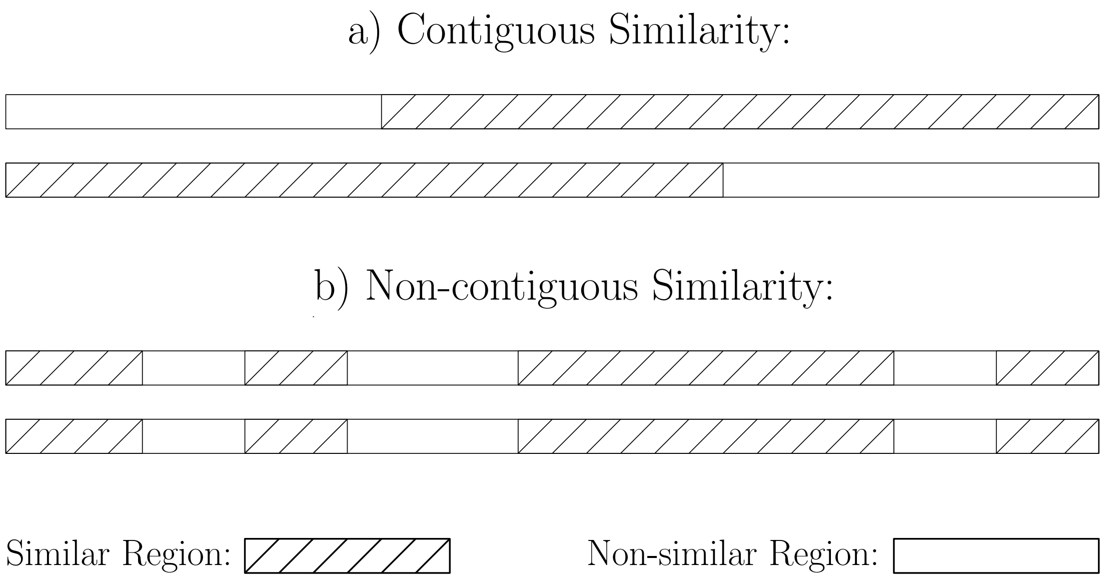


Figure 5.1: (a) Two contiguously similar contigs which should be merged. (b) Two non-contiguously similar contigs which should not be merged.

deal with the misclassification problem shown in Figure 5.1. With the use of the back-propagation through time algorithm a neural network is able to recognise the pattern of base pairs within a read. This classification technique provides a more appropriate measure of similarity in the context of overlap assembly algorithms as reads will be placed in the similar groups only if they are found to be contiguously similar.

5.2 Neural Network Structure and Read Tracking

In order to successfully perform classification using a neural network, a network is first trained to track a particular read. This read, referred to as a seed, is fed into the network one base pair at a time. For each base pair input, the network attempts to predict the next base pair in the seed sequence. When the neural network reaches the last base pair within the seed, it predicts the first base pair of that sequence instead. This is achieved using a three layer recurrent neural network. The network consists of four input, four output and twenty hidden neurons connected in a feed-forward orientation. Additionally, all hidden neurons are interconnected with a time delay of one. This general structure for this type of network is shown in Figure 3.7 in Chapter 3. Each neuron from the input and output layers corresponds to one of the possible base pairs (A, T, G, C). A particular base pair is represented by setting the state of its corresponding neuron to approximately equal 1, and the other neurons to approximately equal -1. The inputs and outputs of the neural network are hence mapped according to

$$\begin{aligned}
\mathbf{X_A}, \mathbf{Y_A} &:= \begin{bmatrix} x_1, y_1 \\ x_2, y_2 \\ x_3, y_3 \\ x_4, y_4 \end{bmatrix} \approx \begin{bmatrix} 1 \\ -1 \\ -1 \\ -1 \end{bmatrix} \qquad \mathbf{X_T}, \mathbf{Y_T} := \begin{bmatrix} x_1, y_1 \\ x_2, y_2 \\ x_3, y_3 \\ x_4, y_4 \end{bmatrix} \approx \begin{bmatrix} -1 \\ 1 \\ -1 \\ -1 \end{bmatrix} \\
\mathbf{X_G}, \mathbf{Y_G} &:= \begin{bmatrix} x_1, y_1 \\ x_2, y_2 \\ x_3, y_3 \\ x_4, y_4 \end{bmatrix} \approx \begin{bmatrix} -1 \\ -1 \\ 1 \\ -1 \end{bmatrix} \qquad \mathbf{X_C}, \mathbf{Y_C} := \begin{bmatrix} x_1, y_1 \\ x_2, y_2 \\ x_3, y_3 \\ x_4, y_4 \end{bmatrix} \approx \begin{bmatrix} -1 \\ -1 \\ -1 \\ 1 \end{bmatrix}
\end{aligned} \tag{5.5}$$

Where X_A and Y_A, X_T and Y_T, X_G and Y_G, and X_C and Y_C correspond to inputs and outputs representing the A, T, G and C base pairs respectively. Figure 5.2 shows output examples after the read tracking process has been completed. Case (a) shows an example of successful read tracking where proceeding base pairs are correctly predicted. Case (b) shows an example of erroneous read tracking where base pairs have been incorrectly predicted. Finally, case (c) also shows an example of erroneous read tracking where the predicted base pairs are unknown. This occurs then the output y_i from each output neuron does not correspond to any of the mapped states as defined in (5.5).

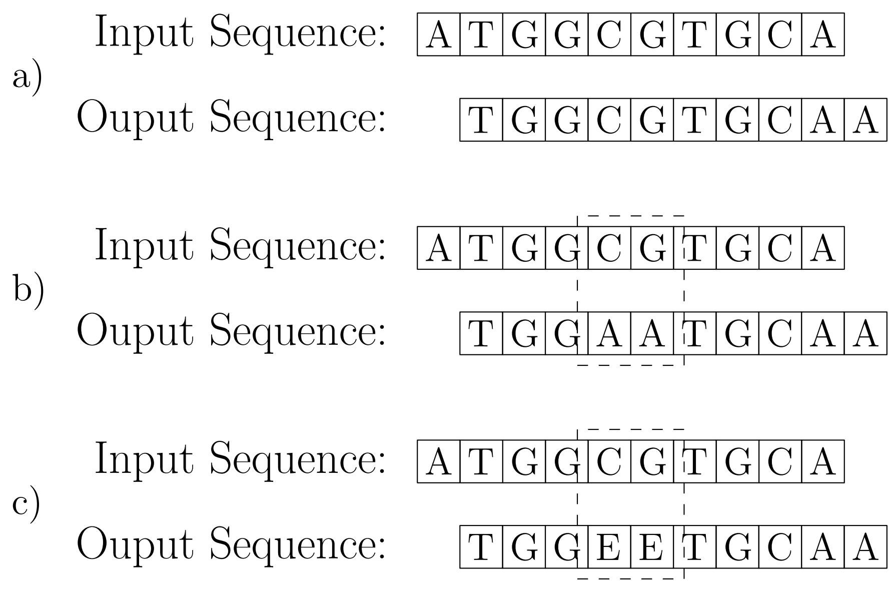

a)
Input Sequence: A T G G C G T G C A
Ouput Sequence: T G G C G T G C A A
b)
Input Sequence: A T G G C G T G C A
Ouput Sequence: T G G A A T G C A A
c)
Input Sequence: A T G G C G T G C A
Ouput Sequence: T G G E E T G C A A

FIGURE 5.2: Neural network output after read tracking is completed. (a) shows correct tracking. (b) shows erroneous read tracking where base pairs are incorrectly predicted. (c) shows erroneous read tracking where neuron outputs do not correspond to any particular base pair.

Training is achieved by performing a series of forward and backward propagations responsible for modifying the weights associated with each connection between neurons within the network. For a given input, forward propagation obtains the outputs, while the backward propagation obtains the change in the weight vector **W**. The backpropagation through time algorithm modifies the standard backpropagation equations given in Chapter 3 by introducing memory into the system. The forward and backward propagation equations are hence given in Equations (5.6) to (5.9) [37].

$$a_j^{(\theta)} = \sum_{i=0}^{I} w_{ij} x_i^{(\theta)} + \sum_{i=0}^{J} w_{ij} z_i^{(\theta-1)} \tag{5.6}$$

$$\sigma(a) = \frac{2}{1+e^{-\beta a}} - 1 \tag{5.7}$$

$$\delta_j^{(\theta)} = \sigma^{'}(a_j^{(\theta)}) \left(\sum_{i=1}^{K} w_{ij} \delta_i^{(\theta)} + \sum_{i=0}^{J} w_{ij} \delta_i^{(\theta+1)} \right) \tag{5.8}$$

$$w_{ij}^{(\theta+1)} = w_{ij}^{(\theta)} - \eta \delta_j z_i^{(\theta-1)} \tag{5.9}$$

Due to the recurrent nature of the network, the standard implementation of the forward and backward propagations is changed. Standard implementation of the backpropagation algorithm involves performing forward and backward propagations in pairs for each input within a given input vector. In the case of backpropagation through time, all forward propagation is first performed on each input within the input vector and the outputs stored in memory. All backward propagations then follow in series for each output stored in memory. Performing backpropagation in this manner allows for previous inputs to the network to influence the propagation of present inputs. This modification to the algorithm is ideal for pattern recognition and hence the tracking of reads for this particular application [36, 37].

The sigmoid function shown in Equation (5.7) is used in order to cap outputs from each neuron to 1 or -1. It is used in order to avoid using a binary hard decision as this would reduce the resolution of the network outputs. The sigmoid function achieves this by acting as a linear function for small inputs and as a hard decision function for larger inputs. The β parameter is responsible for how steep the gradient is for the linear portion of the function. The effect of changing the β value can be seen in Figure 5.3. Setting $\beta = 6$ is desirable because it provides a suitably sized linear region while still capping larger values. A learning rate of $\eta = 0.0001$ was chosen in order to minimise the changes made to **W** and therefore increasing its resolution. A small η value therefore improves the training precision, however it increases the number of forward and backward iterations, or epochs, needed for **W** to converge. The number of hidden neurons also affects the ability of the network weights to converge to the desired value. More hidden neurons

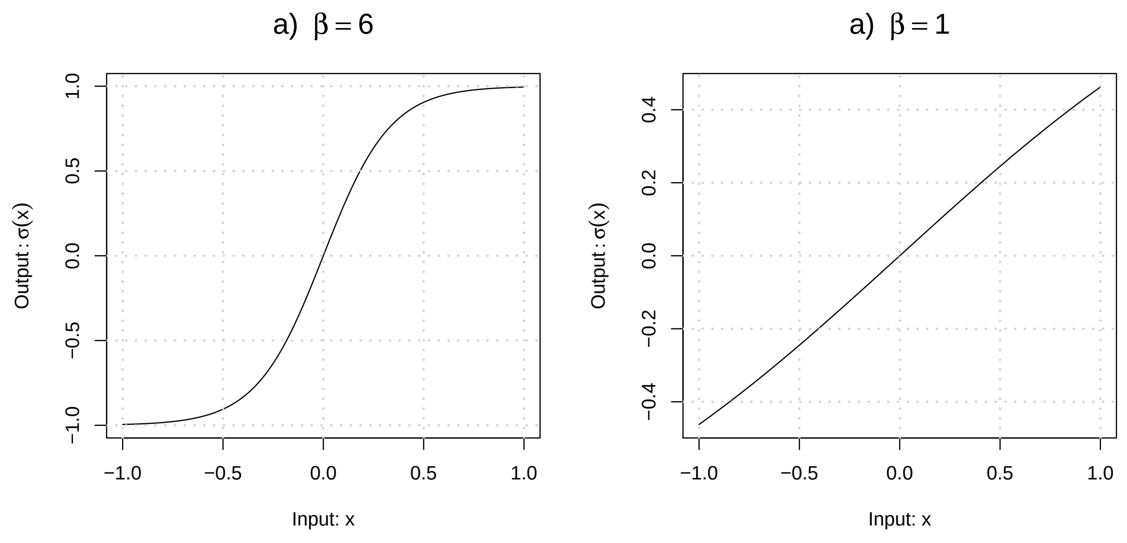


FIGURE 5.3: Effect that β has on the sigmoid function which limits the neuron outputs between -1 and 1.

allow the network to track more complex sequences. Increasing the length of the reads therefore requires an increase in the number of hidden neurons needed to successfully track a read. However, having too many hidden neurons will introduce the problem of over-fitting if the input sequence does not require such higher complexity. Twenty hidden neurons was chosen as an adequate number for tracking reads of length fifty base pairs. The bias terms x_0 and z_0 in Equation (5.6) were set to zero since the network outputs are bound between 1 and -1. The starting weight values of $\mathbf{W}$ were initialised to values ranging between 0.05 and -0.05.

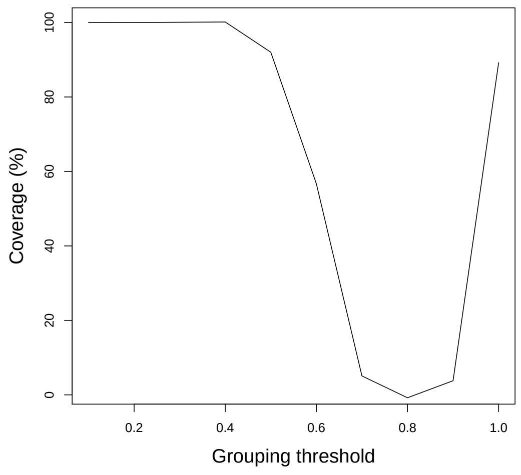
The training process is performed until the changes in $\mathbf{W}$ between consecutive epochs are considered small enough. This happens when the error function used to compare the network output with the desired output reaches a minimum value below specified convergence threshold (a very small number). In such a situation the output from the network at every time interval corresponds to the input read shifted cyclically to the left by one base pair. If the output sequence does not represent the shifted input, either not enough time was provided for the network to converge to the correct $\mathbf{W}$ or a local minimum in the error function was discovered and the network converged to an incorrect set of weights in $\mathbf{W}$. If a network does not converge within a specified number of epochs, or it converges incorrectly, then training is reset by initializing a new set of weights in $\mathbf{W}$.

5.3 Read Grouping

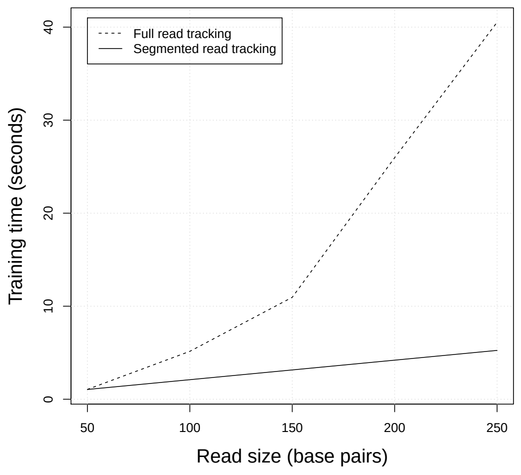
In machine learning, and in this case neural networks, the training process is performed on a training data set. How well classification performs on inputs from a given training set is referred to as in-sample performance. Once trained, a neural network is be able to generalise inputs other inputs which are not within the training data set. How well the network performs on non-training data is known as out-of-sample performance [36]. In this case, training is performed on a training set consisting of a single element, the seed for a number of epochs until the in-sample performance converges to an acceptable level. Testing the network with other reads (non-training data) is then done in order to group reads [36]. The membership of a read to a group is determined by whether its out-of-sample performance is above a certain threshold defined as τ [11].

Preliminary experiments have shown that the size and number of groups is dependent on τ. If the value of τ is too high, then the similarity between seed and read needs to be great for a read to be placed in the same group as the seed. A larger amount of groups containing fewer reads will be created in this case. On the other hand, if τ is too low then fewer but much larger groups are created. It is important to find the right balance as both cases are undesirable. The grouping threshold used by the neural network grouping scheme was set to $\tau = 0.4$ which is consistent with the value used by Angeleri et al. [11]. Figure 5.4a shows the effects that the grouping threshold has on the assembly process. It shows that at lower values, the grouping criteria is very strict which leads to many groups consisting of few reads. In such a case, the second assembly round of the neural network assembler resembles the situation where grouping is performed. On the opposite side of the scale, the grouping criteria is so loose that all reads are placed in the same group, or a very small set of groups. In this case the first assembly round resembles the pre-grouping situation (placing all reads in one group does not divide reads). The trough found in the middle of the scale is present due to reads being erroneously placed into groups. The grouping threshold τ was chosen at the knee of the curve in Figure 5.4a as this represents the highest simulation coverage using the largest amount of groups [11].

The neural network grouping assembly scheme is introduced to partition reads into groups before assembly takes place. Once grouped, either the greedy or de Bruijn assemblers are implemented in order to piece together the groups of reads. However, the performance of these assemblers depends on the ability of the neural network to correctly and efficiently place the read sequences into groups. Preliminary experiments have shown the neural network training process to be quadratic in time with respect to the length of the read being tracked. This can be seen in Figure 5.4b. In order to overcome the quadratic nature of the training process a segmented approach to the



(A) The effect which the neural network grouping threshold has on the assembly performance.



(B) Complexity of the read tracking process based on read length. This is the time it takes to train a neural network to track a seed.

FIGURE 5.4: Preliminary experimental results relating to (A) the grouping threshold τ and (B) the neural network training complexity.

training process is proposed, larger sequences can be segmented (broken down into smaller chunks) where each segment is used to train a unique neural network. This segmented approach employs multiple neural networks to track a read sequence where each neural network will be responsible for tracking a subsequence, or segment, of the entire read sequence. Figure 5.4b shows a dramatic decrease in training time for larger sequences when implementing this segmented read tracking approach. This approach deviates from that proposed by Angeleri et al. [11] in that it splits the training process into multiple smaller training processes. This new approach is further described in the next section (Section 5.4)

Using machine learning and neural networks, a new similarity measure metric μ between reads, as described by Angeleri et al. [11], is introduced. This metric is arrived at by first selecting a seed A on which a neural network NN_a is trained. This network then acts as a discriminating function used by other fragments to determine their similarity to A using the following definitions. First, for two sequences of the same length, $d(A, B)$ is defined as the ratio between the number of symbol differences between the sequences and their length. Second, if A_a is defined as the output of network NN_a tracking input A and B_a is the output of network NN_a tracking input B, then read B belongs to the same group as A if $d(B_a, B)$ is below the threshold τ. This is depicted in Figure 5.5. In order to clean up notation, the similarity measure μ is then defined as

$$\mu(B, A) := d(B_a, B) \tag{5.10}$$

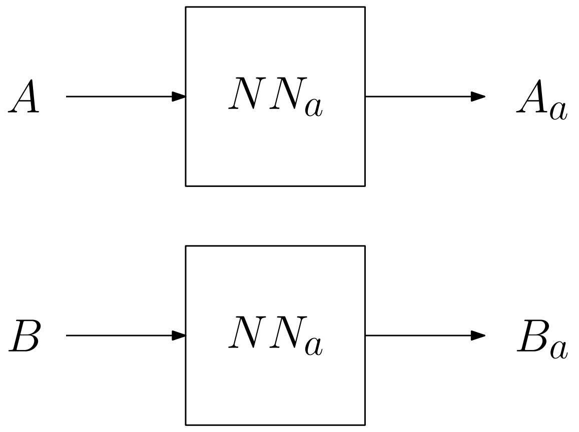


FIGURE 5.5: Block diagram showing the input and output sequences to a neural network. NN_a is a neural network tracking read A. A_a is the output sequence from the neural network NN_a given an input sequence A. B_a is the output sequence from the neural network NN_a given an input sequence B.

which measures the degree of membership of sequence B to the cluster tracking seed A. Therefore a read B belongs to the same group as A if

$$\mu(B, A) < \tau \tag{5.11}$$

The network NN_a attempts to predict the sequence within read B and if this is achieved well enough it is said that B belongs to the same group as A. The similarity measure $\mu(B, A)$ is preferable over similarity measure $D(A, B)$ described in Section 5.1 due to its ability to group reads that are contiguously similar with one another [11]. Additionally, differences in read sizes are well handled by this grouping approach. This is due to the read grouping process being based on a comparison between sequences of the same length, since $\mu(B, A)$ in (5.10) is a measure of difference between sequences B and B_a, which are the same size. If the neural network NN_a is trained on a seed a of different length compared to read B, then the difference between B and B_a will be greater, and the threshold criteria of (5.11) will be more difficult to satisfy if B is to belong to the same group as A.

5.4 Neural Network Assembler

The ability of the neural network to classify and place reads into different groups is important as it divides the assembly problem into multiple smaller scale problems [11]. This is desirable as it reduces assembly complexity as shown in Section 5.1. After the assembly problem has been reduced into smaller groups, the greedy or de Bruijn

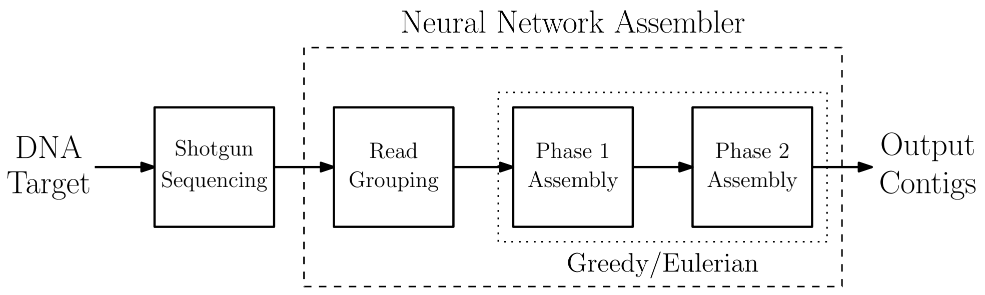


FIGURE 5.6: Block diagram of the neural network assembler. Phase 1 and 2 assembly blocks implement either the greedy or de Bruijn assemblers.

assembly algorithms are then applied in order to assemble the target sequence. This scenario is depicted in Figure 5.6. The crux of the neural network assembler therefore lies in the classification process of read sequences. This process consists of training one or multiple neural networks to track a single seed sequence selected at random from the read set. This neural network, or collection of multiple sub-neural networks (if more that one neural network is used to track a particular seed) will be used to classify other reads into the group tracked by the seed. This process is repeated until no more seeds and reads remain.

For example, if reads of size 200 base pairs are segmented into chunks of size 50 base pairs, four neural networks will be used to track each seed (one for each segment). In this case the neural network NN_a tracking seed A, as shown in Figure 5.5, consists of multiple sub-neural networks NN_{1a}, NN_{2a}, NN_{3a} and NN_{4a}. When another read B is tested against NN_a, it too is split into four segments B_1, B_2, B_3 and B_4 such that each segment will be tested in a sliding window fashion with each sub-neural network in NN_a. Each result is recoded and if the average sum of any of the results meets the requirement of (5.11), then read B will be placed in the same group as seed A. The algorithm for this segmented machine learning assembly scheme is presented in Algorithm 9.

The success of the machine learning assembly scheme depends on the accuracy and efficiency of the read grouping process [11]. If read grouping, steps 2, 3 and 4 of Algorithm 9, is too time consuming or if grouping is too inaccurate then assembly becomes infeasible. The complexity of the grouping step is also of $O(N^2)$ complexity, however the training of networks can be parallelised in order to help speed up the grouping process. Furthermore the individual group assembly can also be parallelised to reduce the overall assembly time. The machine learning assembly scheme therefore allows for faster assembly based on the amount of processing cores available.

Additionally, the first round of assembly in the neural network assembler does not produce contigs of the same size. This is because the number of reads in a group and the accuracy of assembly within the group varies due to a number of factors. These include

Algorithm 9: Neural Network Assembly Scheme Algorithm

```
while more than 2% of original reads remain in read set do
    Select a seed $S$ at random from the read set;
    Split seed into $n$ segments;
    for $i < n$ do
        Train a neural network set $NN_{S_i}$ to track the seed segment $S_i$
    for each read in the read set do
        Select a read $R$;
        Split $R$ into $n$ segments;
        for each alignment $j = n + (n - 1)$ of $R$ in a sliding window with $NN_S$ do
            Test segments with corresponding neural network to generate $\mu_k$, where $k$ is
            the number of neural networks being tested $(k = 1, 2, ..., j)$;
            Generate $\mu_{ave} = \frac{\sum \mu_k}{k}$;
            if $\mu_{ave} < \tau$, place $R$ into the same group as $S$;
Apply assembly phase 1 one every group to create a new pool of contigs;
Apply assembly phase 2 on the pool of contigs to obtain the final assembly output.
```

gaps in the target sequence coverage and the error rates associated with the shotgun sequencing process. Because the inputs to the greedy assembly may now vary in size, the maximum overlap associated with the greedy assembly algorithm needs to be adapted as follows

$$\phi_{max} = min(x - 1, y - 1) \tag{5.12}$$

where x and y are the lengths of the contig and read being compared respectively.

5.5 Summary

In this section the idea of breaking the assembly problem into multiple but smaller subproblems was discussed. We showed that in doing so the overall assembly complexity is reduced. A new similarity metric μ was introduced which was shown to more accurately place reads into the correct groups compared to the edit distance metric D. A neural network grouping approach which uses this new metric was discussed and a new assembly scheme was presented.

Chapter 6

Results and Analysis

The DNA sequencing and assembly problem introduced in the earlier chapters was simulated using the aforementioned algorithms and techniques. The Greedy and de Bruijn assembly algorithms were implemented together with the neural network grouping technique in various combinations in order to establish several assembly strategies, or assemblers. The performance of these strategies is compared to determine if the 'divide and conquer' approach of the neural network technique helps to reduce the overall complexity and improve the accuracy of the assembly algorithms. This chapter discusses the research paradigm and the approach taken with regard to data generation and the implementation of the assembly algorithms. All simulation components were implemented in the C++ environment.

The simulation results are presented and analysed in this chapter. The performance of all three assembly strategies are compared with one another and justified. A complexity analysis for each strategy is presented to support the results. The optimal operating conditions for each strategy are also identified along with their strengths and weaknesses.

6.1 Research Paradigm and Setting

This research aims to answer the research question by means of computer simulations implementing real-world assembly algorithms. A pre-assembled (target) DNA sequence is used to test each simulation strategy. These simulations will provide empirical data measuring both the time complexity of algorithms and the accuracy of the assembly outputs. This analysis is aided by information theoretic principles in order to optimise the assembly process by determining the strategies which produce the highest reconstruction accuracy using the lowest amount of reads. Additionally, the trade-off between accuracy and time complexity for each assembly strategy is investigated. Hence, this research implements both quantitative and qualitative means of analysis in addressing the research problem.

Due to the DNA assembly problem being a notoriously difficult problem to solve, some assumptions have been made in order to simplify the research problem. Firstly, a target DNA sequence with length of 50456 base pairs ($G \approx 50000$) was selected in order to reduce simulation times. The target length is therefore small enough such that the simulations do not become a high performance computing problem while still being a large enough sample for analysis. It was found that some practical assembly scenarios are of this order of magnitude [10]. Secondly, the double stranded nature of DNA has been ignored and it was assumed that all input data was generated from only a single DNA strand. While we know that this is not the case for practical real-world situations, this decision was made in order to reduce complexity of the problem. The issue of noise (errors) in the input generation, which is a common occurrence [8], was also taken into consideration and tests were performed on both error and error-free cases.

Prior to implementing any simulation, the target sequence must first be determined. A 50456 base pair portion from the Fruit Fly genome was selected as the target sequence. The gene was obtained from the FlyBase gene database [13] and exported to a text file. The file is then parsed in order to remove all meta-data and extract the 50456 base pair sequence. This specific target sequence was chosen due to its size and repeat characteristics as it contains repeats which are two hundred base pairs in length.

6.2 Simulation Methodology

The simulations were implemented in three phases, namely sequencing, assembly and information formatting. These steps are responsible for generating input data to the assemblers, implementing the assembly strategies, and formatting the output information from the assemblers such that it may be easily interpreted. The accuracy of each assembler is determined by comparing its output to that of the known target sequence. Obviously, this luxury is not afforded in real-world assembly applications.

6.2.1 Shotgun Sequencing

The shotgun sequencing process was implemented by a stand-alone program. The process generated sub-sequences (reads) of a predetermined size at random locations across the target sequence and with a uniform distribution. These reads were treated as string variables by the simulation program and consist of characters representing the A, T, G and C base pairs.

The number of reads taken is based on the size of the reads and the coverage depth according to

$$N = \frac{G\ c}{L} \tag{6.1}$$

In practice, the coverage depth varies based on the sequencing technology being used and may range from five to sixty times coverage [1, 23]. Based on the results from this research, it was found that coverage depths larger than 20 were unnecessary for the size of the selected target sequence. The simulations were performed using four coverage depths

$$c = \{5, 10, 15, 20\}$$

Different read sizes were also taken in the range consistent with popular sequencing technologies. Sanger sequencers create reads of size anywhere between $100 - 1000$ [8] while more modern sequencers such as Illumina generate reads of 250 base pairs average length [7]. In order to be consistent with existing sequencers and to investigate the effect that varying read sizes has on the assembly process, the simulations used six difference read sizes

$$L = \{50, 100, 200, 300, 400, 500\}$$

The assumption that all reads generated by the sequencer are of equal length was made in order to simplify the simulation process. In this way, the research can identify differences in complexity and performance between assemblers under ideal read conditions. In reality, as discussed in Section 2.2, sequencers generate reads of varying length. This may effect read grouping, and hence, the performance of the neural network assembly scheme proposed in Chapter 5, however, as discussed in Section 5.3, the proposed scheme will still be able to handle varying read sizes.

The Velvet assemblers use reads of size $25 - 35$ base pairs [23]. The simulations were not performed with such small read sizes since the de Bruijn assembler implemented in this research did not make use of any of the improvements made to the de Bruijn graph approach as was the case in the Velvet assembler.

The shotgun sequencing process was performed using four separate error rates in order to evaluate the performance of the assembly strategies at varying read qualities. These error rates are as follows

$$error\ rate = \{0, 0.0001, 0.001, 0.01\}$$

The read sets generated by the shotgun sequencing process obeyed the minimum bounds described in Chapter 3 needed for complete coverage of the target sequence. However, due to the nature in which reads are generated by the shotgun sequencing program, the head and tail ends of the target sequence are very rarely covered. This is because for a given target sequence size G, the probability of covering the head or tail of the sequence is given by

$$P(covering\ head) = P(covering\ tail) = \frac{N}{G} \tag{6.2}$$

In most simulation scenarios, the size of G is considerably larger than the size of N. For this reason, 100% assembly is unlikely to occur at any of the coverage depths and read sizes used in the simulations.

The choice to perform simulations at varying read sizes, coverage depths and error rates provides a total of 96 different simulation scenarios. The assemblers were applied to each of these scenarios in order to establish trends and give insight into the parameters which most affect the various assembly strategies. This will also provide information as to how each assembly strategy performs under such varying conditions. For each given size and coverage depth, the reads generated are stored in intermediate text files to be parsed by the assemblers at a later stage.

6.2.2 Assembly Strategies

Assembly of the target sequence takes place after sequencing is completed. The size and number of the reads, and hence the coverage depth, is determined from the read file selected as an input to the assembler. Once the reads have been obtained, four assemblers are implemented in parallel. The assemblers are as follows:

- The greedy assembler implements the greedy assembly algorithm;
- The de Bruijn assembler implements the Eulerian path finding assembly algorithm;
- The greedy neural network assembler implementing read grouping followed by the greedy assembly algorithm;
- The de Bruijn neural network assembler implements read grouping followed by the Eulerian path finding assembly algorithm.

The first two assemblers implement the popular de Bruijn and greedy assembly algorithms. The de Bruijn assembler implements the modern approach to the DNA assembly problem and is expected to have a better time complexity performance compared to the greedy assembler [1, 8]. The final two assemblers introduce the neural network grouping

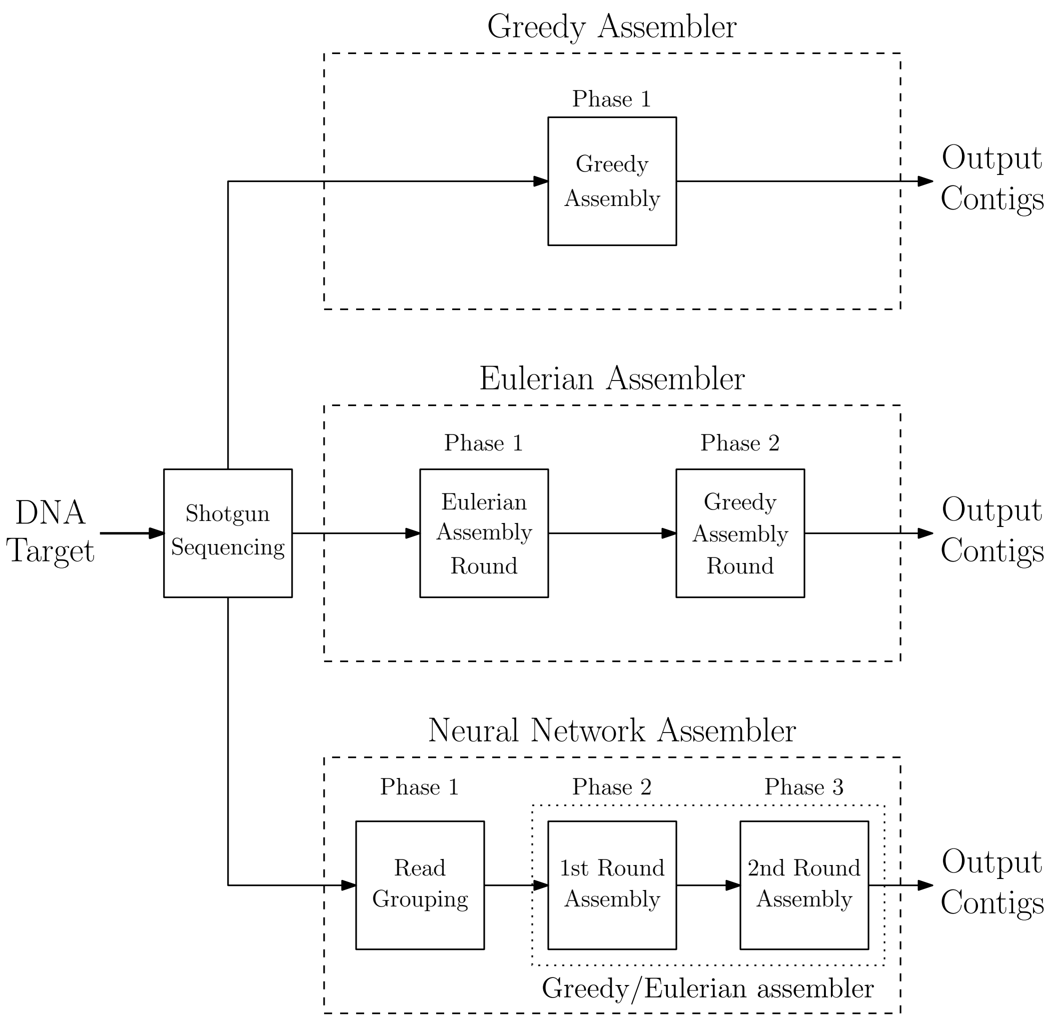


FIGURE 6.1: Assembler block diagrams.

as an extra step prior to the implementation of the assembly algorithms. The aim of this extra step is to reduce the time complexity of assembly and potentially improve accuracy. The neural network assemblers are implemented in three phases. Firstly, phase one groups similar reads together. Secondly, phase two assembles the reads within each group in order to form group contigs. Finally, phase three assembles the group contigs together to generate the final output contigs. Figure 6.1 shows a block diagram overview of each of these strategy scenarios.

Both the greedy and de Bruijn assembly algorithms generate a number of contigs as outputs depending on the degree of success of the assembly process. If assembly is successful, a single contig representing the target sequence is generated. However, if the target sequence is not fully reconstructed, multiple contigs which cover various areas within the target sequence are generated. For the case of the de Bruijn assembler, it is common for the de Bruijn algorithm to generate many output contigs as only sub-paths within the de Bruijn graph can be found [23]. In these situations, it is necessary to implement a final round of greedy assembly in order to piece these contigs together. Similarly, for the case of the de Bruijn neural network strategy, the greedy algorithm is implemented on the outputs from the de Bruijn algorithm in the third phase of the assembler.

Multiple simulations were performed with each assembler under the 96 different conditions in order to obtain an average and determine the success rate of each assembler. Due to time complexity being an issue, and with the limited computing power available, only 20 assemblies for each input scenario were simulated using the de Bruijn, greedy neural network and de Bruijn neural network assemblers. For the case of the standalone greedy assembler, only 10 simulations were performed at each input scenario due to its $O(N^2)$ time complexity. These decisions were made in order to obtain results in a reasonable amount of time. While the low number of simulations might be of concern, they were enough to show the trend in assembly performance across the four assemblers being simulated. A higher number of simulations would be preferred, and recommended, if larger amounts of computational power is obtainable.

6.2.3 Information formatting

The assemblers generate output files containing the assembly statistics which measure the coverage information, the number of contigs created, the average contig size, the percentage error, and the time taken to complete the assembly. Once the assemblers have finished generating the output files, a third program is used to parse these files and provide input to a plotting program. The R software environment was used to generate the assembler performance plots and sequence dot plots which are presented later in this chapter.

6.3 Evaluating Assembly Performance

Keeping in mind that the target sequence is known, the performance of each strategy can be measured against the known target sequence. The output contigs generated by the assemblers are correlated against the target sequence using a sliding window approach in order to determine how much of the target sequence they cover. Hence, a contig's coverage is defined as the maximum number of base pairs which correlate against the target sequence. The total coverage of an assembler is therefore determined by summing together the coverage of each output contig. However, an issue arises when using total coverage to evaluate the performance of assemblers which generate a different number of output contigs. Some assemblers produce a single output contig while others produce

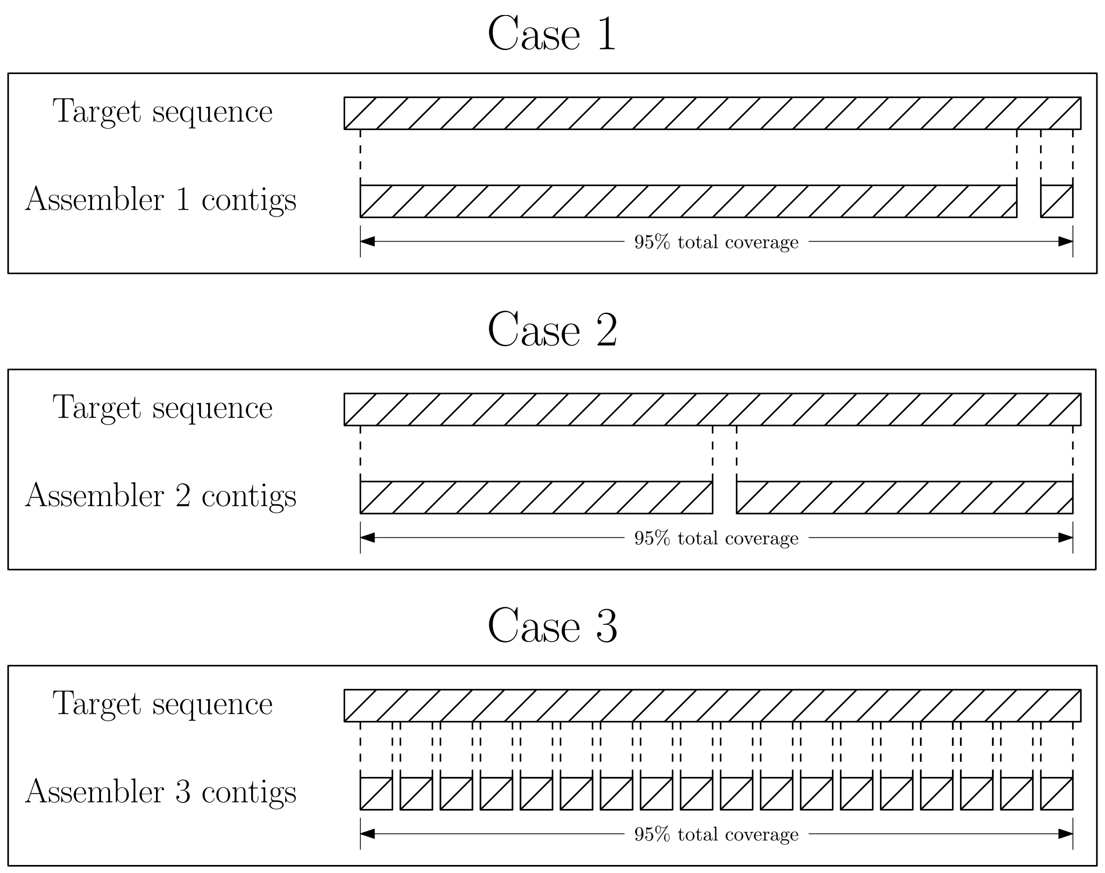


FIGURE 6.2: Three cases of varying assembly outputs. All three achieve a total coverage of 95% of the target sequence, however the size and number of contigs vary.

multiple ones. For example, Figure 6.2 shows three cases where assemblers produce the same total coverage with varying amount of contigs at varying sizes. One can argue that the outputs from cases one and two are superior when compared to the outputs from case three. Similarly the output from case one is superior to that of case two. This is because larger contiguous output contigs provide more meaningful information compared to the many disjoint output contigs. While case three might produce the same total coverage compared to case one and two, the output contigs do not provide the same level of information about the target sequence. In reality, one does not know which portion of the target sequence each output contig represents. Therefore a new metric, other than the total coverage, which takes into account the varying number of output contigs and their size is needed to appropriately compare assemblers.

A new performance metric is proposed which produces an adjusted coverage score based on the assembler's output statistics. Instead of simply adding together the coverage from each contig, the performance metric is composed of two components. The first component represents the largest contig produced by the assembler while the second consists of a scaled squared average of the coverage from all contigs, including the largest.

This performance metric is defined as

$$M_{adjusted\ coverage} = M_{largest\ contig} + \alpha M_{squared\ average} \tag{6.3}$$

where α is the scaling factor acting on the squared average component. The components $M_{largest\ contig}$ and $M_{squared\ average}$ along with the scaling factor α are obtained using

$$M_{largest\ contig} = \max\{x_1, x_2, \ldots, x_n\} \tag{6.4}$$

$$M_{squared\ average} = \sqrt{\frac{x_1^2 + x_2^2 + \cdots + x_n^2}{n}} \tag{6.5}$$

$$\alpha = \frac{(\sum_{i=1}^{n} x_i) - M_{largest\ contig}}{100} \tag{6.6}$$

where x and n represent the assembly contig coverage and number of output contigs respectively.

The scaling factor α is a normalised sum of the total coverage obtained from all but the largest contig. The squared average component is used in order to produce an average which has a bias towards larger contigs produced by the assembler. This combination of components ensures that the performance metric generates an adjusted coverage which lies between the largest contig and the total coverage. This approach to measuring the performance metric ensures that assemblers which produce larger contigs with the same overall coverage are scored higher than assemblers which produce multiple contig outputs with a smaller individual coverage.

In order to avoid counting unassembled reads, or contigs consisting of only a few assembled reads, towards the total coverage and adjusted coverage performance metric, only contigs larger than 1.5 times the size of the original reads were considered as output contigs. This is necessary since all reads would otherwise contribute to the total coverage and all assemblers would achieve 100% coverage.

6.4 Simulation Results

All simulations were performed using the input parameters presented in Section 6.2.1 of this chapter. Based on the cardinality of the input conditions, the four assemblers were simulated under 96 different conditions. These parameters ensure that the minimum bounds required to fully cover the target sequence at a specified read size are met. These minimum bounds are shown in Table 6.1 and are defined by Equation (3.3) in Section 3.2. While these bounds are necessary, they do not guarantee successful reconstruction and therefore the values used for N, and hence c, in the simulations are

Table 6.1: The minimum number of reads and coverage depths needed at varying read sizes in order for reads to fully cover the target sequence of 50456 base pairs with a probability of 99% ($\epsilon = 0.01$).

Read size:	N_{cov}	c_{cov}
$L = 50$	11513	11.51
$L = 100$	5410	10.82
$L = 200$	2532	10.12
$L = 300$	1620	9.72
$L = 400$	1179	9.43
$L = 500$	921	9.21

much higher than the lower bounds shown in Table 6.1. The relationship between these two parameters is dependent on the target size G and read size L, and is given as $c_{cov} = \frac{LN_{cov}}{G}$. Additionally, The size of the target sequence also imposes a lower bound on the read size. This was discussed in Section 3.3 and is given as

$$\bar{L} > \frac{2}{H_2(\mathbf{p})} \tag{6.7}$$

For the fruit fly genome and a target sequence of 50456 base pairs in size, this bound requires that $L \leq 15.7$, which is satisfied by all six read sizes used in the simulations.

6.4.1 Coverage Statistics

The plots found in Appendix A contain the simulation results showing the coverage statistics at the varying coverage depths, read sizes and error rates for each assembler. These figures, which will be discussed in more detail, show; 1) the total coverage of all contigs, 2) the total coverage of only the correct contigs (correct coverage), 3) the number of contigs and 4) the size of the largest contig obtained from each assembly simulation. This information is used to determine the coverage performance along with the adjusted coverage performance metric defined in Section 6.3 for each assembler.

The total coverage consists of the summed total of all contig coverages generated by the assembler. Some of these contigs, which contribute to the total coverage, are erroneously merged. These contigs are identified by correlating each contig with the target sequence; if more than 5% of the contig does not match the target, it is given an erroneous status. The correct coverage is then calculated by ignoring contigs with an erroneous status. These erroneous contigs occur due to the presence of repeats within the target sequence as discussed in Section 4.1. The difference between the total coverage and correct coverage is used to establish the assembly error of each assembler.

The results in Appendix A show that the total coverage lies above 70% for the greedy and greedy neural network assemblers regardless of the coverage depth, read size, and error rate used. This is because in most cases, the assemblers produce many small output contigs which range between $1 - 5\%$ of the target sequence in size. As was discussed in Section 6.3, this does not reflect the true performance of an assembler. When larger read sizes are used, these assemblers produce a similar total coverage with fewer but larger contigs (which is desirable). Additionally, increasing the error rates did not negatively affect the correct coverage and largest contig sizes for the greedy assemblers. It does however increase the number of contigs generated when using smaller read sizes and in some cases causes the total coverage to exceed 100%. As discussed in Chapter 4.1, this occurs when the fractional overlap between contigs is smaller than the minimum overlap threshold ϕ_{min}. In such cases, contigs that should otherwise be merged together are left un-merged, which then contribute to the total coverage.

Unlike the greedy and greedy neural network assemblers, the de Bruijn and de Bruijn neural network assemblers produce a smaller total coverage at higher error rates. Additionally, they also produce more numerous and smaller contigs. This is due to the differences in which each assembler handles the contig creation process. Unlike the greedy assembler, where contig-read pairs are merged together, the de Bruijn assembler finds a path through the de Bruijn graph. When errors from the sequencing process causes bubbles and tips to occur with a graph, finding the correct path can become an issue. In cases when read-path information can not resolve a bubble or tip, the path finding algorithm generates incorrect paths or terminates prematurely resulting in smaller output contigs.

Repeats present in the target sequence are also resolved using read-path information. As discussed in Chapter 4.2, repeats are resolvable only if the size of the k-mers are large enough to cover the repeat regions. In the case of the de Bruijn assemblers, discrepancies between the total and correct coverage still exist due to the final round of the de Bruijn algorithm implementing an instance of the greedy overlap algorithm (as shown in Figure 6.1).

6.4.2 Adjusted Coverage Performance

The adjusted coverage performance metric was introduced as a more appropriate comparison between the coverage performance of each assembler (as discussed in Section 6.3). Figures 6.3, 6.4, 6.5 and 6.6 show the coverage performance of each assembler at 5, 10, 15 and 20 times coverage depths respectively. They show the adjusted coverage using the proposed performance metric at each read size, however they only show the results from the 0% and 1% error rate simulations. The rest of the adjusted coverage results, with error rates of 0.01% and 0.1%, can be found in Appendix B. The plots show the adjusted coverage performance from only the most successful simulations performed by each assembler under the given input conditions. Additionally, the number of successful simulations (simulations with over 95% adjusted coverages) are also displayed to give an indication of the success rate of each assembler.

Several observations can be made from Figures 6.3, 6.4, 6.5 and 6.6 along with those in Appendix B. The first, and most intuitive, is that the coverage performance of each assembler increases when the read size is increased. Increasing the size of reads increases the number of unique possible reads by a factor of four for each base pair added (since there are only four possible nucleotides; A, T, G and C) [22]. Having a larger set of unique reads significantly decreases the occurrence of redundant reads which then decrease the probability of erroneous merging. Having larger reads also accommodates for larger overlaps by providing more information to the assembly algorithms. The second, and also intuitive, observation is that increasing the coverage depth also increases the coverage performance of all assemblers. Naturally, having more pieces to reconstruct the target sequence increases the likelihood of covering the entire sequence and hence having the information available for reconstruction. Hence, decreasing the coverage depth reduced the assembler's performance. Studying Figure 6.3, it can be seen that no successful simulations occurred due to the coverage depth being too low. Thirdly, introducing sequencing errors into the reads decreases the performance of all assemblers at all coverage depths and read sizes. However, the two de Bruijn based assemblers are significantly more affected when compared to the two greedy based assemblers. Regardless of the coverage depth, the de Bruijn assemblers produced no successful reconstructions at error rates of 1%. Additionally, not one assembler achieved 100% coverage. As explained in Section 6.2.1 of this chapter, this is due to the low probability of covering the head and tail ends of the target sequence.

6.4.3 Results Evaluation Based on Adjusted Coverage

The most optimal performance of each assembler can be calculated using the normalised coverage depth given in (3.14) from Chapter 3. Because the success rates (achieving above 95% adjusted coverage) of each assembler varies, only assemblers with 50% and above success rates were used in determining optimality. This was calculated using the lowest coverage and read size where an assembler achieved successful reconstruction and the c_{cov} value corresponding to the matching read size in Table 6.1. Using this approach, Table 6.2 shows the normalised coverage depth for each assembler without sequencing errors. Due to the limited number of simulations performed for each assembler, these values may not be 100% accurate. They do however give an indication that the greedy type assemblers perform closer to the optimal bound ($\bar{c}_{min} = 1$) when compared to the de Bruijn type assemblers. This is consistent with the analysis done by Tse et al [8].

The greedy assembler along with the greedy neural network assembler were able to achieve successful reconstruction at 10, 15 and 20 times coverage depths. Introducing sequencing errors in the shotgun sequencing process did not affect their performance. This is due to the error tolerance incorporated into the greedy assembly algorithm. Due to the grouping process of the neural network assembler, increasing the coverage depth seems to have a negative impact on the assembler's success rate. Since more reads are generated at higher coverage depths, there is a higher chance for erroneously classifying reads into incorrect groups. The benefit to using the greedy neural network assembler comes from the significant improvement in the computational complexity. This can be seen in Figure 6.7 where the neural network grouping scheme significantly improved simulation time of the greedy assembler.

The de Bruijn assembler also achieved successful assemblies at 10, 15 and 20 times coverage depths. However they require higher coverage depths in order to achieve higher success rates when compared to the greedy assemblers. It is desirable to keep the value of k to a minimum as this will reduce the coverage depth needed for successful assembly [34]. However, it is also important to keep the size of k large enough such that repeats present in the target sequence are still resolvable by the Eulerian path finding algorithm. As was discussed in Chapter 4, the size of the k-mers was chosen as $k = 0.4L$. Because the de Bruijn assembler creates these smaller k-mers, it performs significantly worse at lower read sizes when compared to the greedy assemblers. As can be seen in Figure 6.7, the benefit from using the de Bruijn assembler comes from the significant improvement in computational time needed for assembly. Unfortunately, this significant improvement in computational complexity is lost when combining the neural network grouping scheme with the de Bruijn assembler. While the de Bruijn neural network assembler performs on par in terms of coverage with the standard de Bruijn assembler, it is slower due to the added overhead associated with the neural network training and read grouping process. For this reason implementing the de Bruijn neural network assembler is undesirable.

TABLE 6.2: Assembler normalised coverage depths without the presence of sequencing errors.

Assemblers	Coverage	Read size	$\bar{c}$
Greedy assembler	10	300	1.03
de Bruijn assembler	15	200	1.48
Greedy neural network assembler	10	300	1.06
de Bruijn neural network assembler	15	200	1.48

6.5 Results Discussion

Studying the simulation results reveals a number of important findings. Firstly, the greedy and greedy neural network assemblers had a better coverage performance at smaller read sizes when compared to the de Bruijn assembler. This is expected since information is lost when breaking down reads into the smaller k-mers [34]. Secondly, the coverage performance of the de Bruijn assemblers were significantly reduced when introducing higher error rates in the shotgun sequencing process. This is due to the lack of a consensus or validation step within the de Bruijn assembly algorithm as in assemblers such as Velvet [23]. These steps would help eliminate the effect that bubbles and tips, caused by sequencing errors, have on the construction of the de Bruijn graph [23]. Thirdly, implementing the neural network grouping scheme together with the greedy algorithm does not negatively impact the coverage performance of the greedy assembler. Instead, the neural network grouping significantly improved the computation time, and in some cases even the coverage performance, of the greedy assembly algorithm. This is consistent with the results obtained by Angeleri et al [11]. However, unlike the greedy assembler, implementation of the neural network together with the de Bruijn assembler is infeasible. As discussed in Section 5.4, a neural network is trained to track each seed selected to represent a group. Additionally, once each neural network is trained, the remaining reads are tested against it in order to determine their candidacy to the group. This overhead significantly slows down the de Bruijn assembler since the training and grouping process was shown to be of a higher complexity compared to the de Bruijn assembly algorithm. Lastly, at lower error rates, the de Bruijn assemblers achieved more consistent successful assemblies (coverage over 95%) when compared to the greedy assembler operating under the same conditions.

With the above findings in mind, due to the high computational complexity associated with the stand-alone greedy assembler, it is recommended that the neural network grouping is always implemented together with the greedy assembler. However, even with the reduction in computational time provided by the read grouping process, the de Bruijn assembler still significantly outperforms the greedy neural network assembler in terms of computational complexity. It is therefore recommended that the de Bruijn assembler be used when sequencing large targets, if good quality reads can be obtained. In situations where good quality reads can not be obtained, or when sequencing smaller targets, the greedy neural network assembler is a suitable assembler as it produces a more accurate coverage.

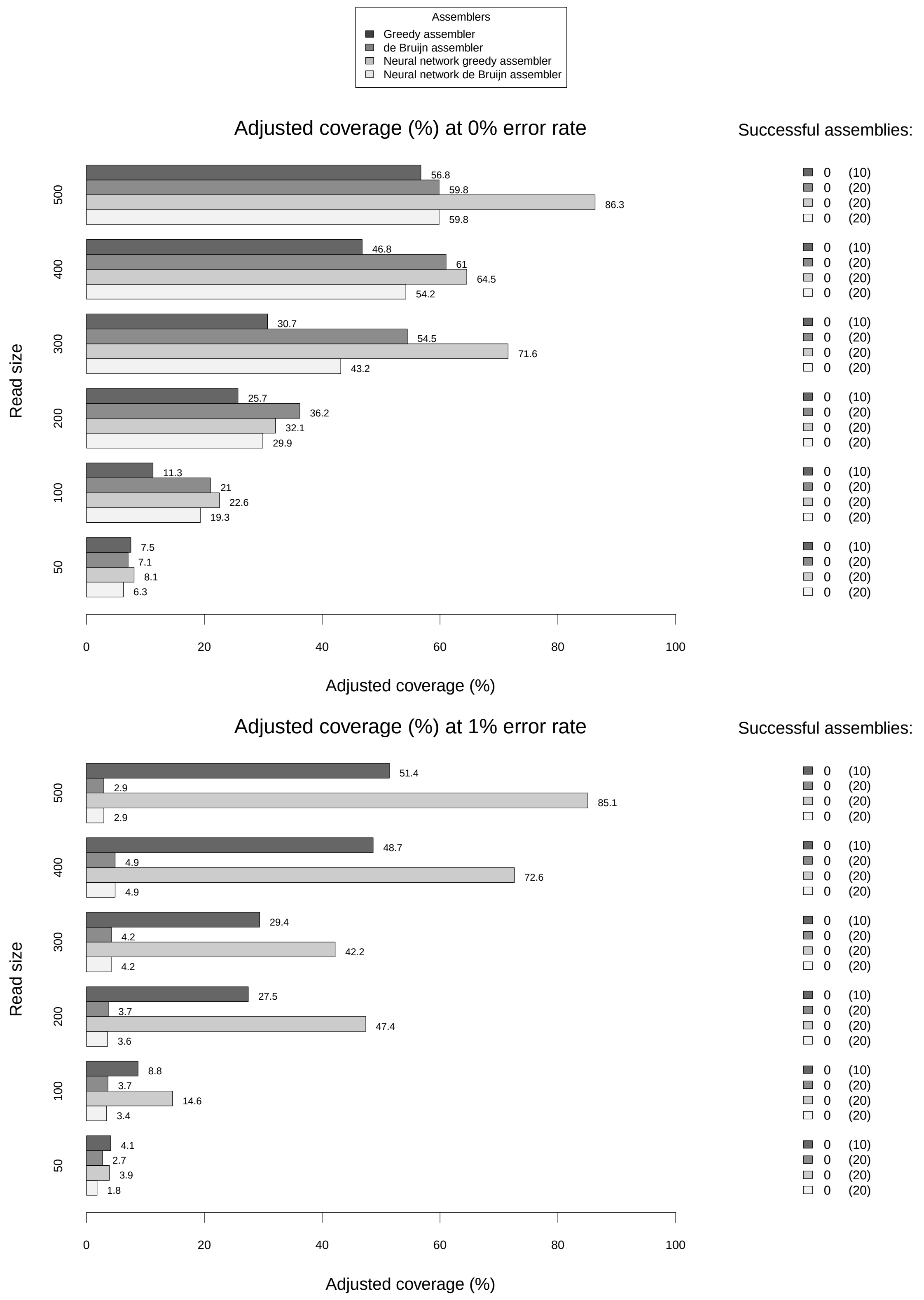


FIGURE 6.3: Adjusted coverages of best performing simulations at 5 times coverage depth. The plot shows performance at both 0% and 1% sequencing error rates along with the success rate of each assembler.

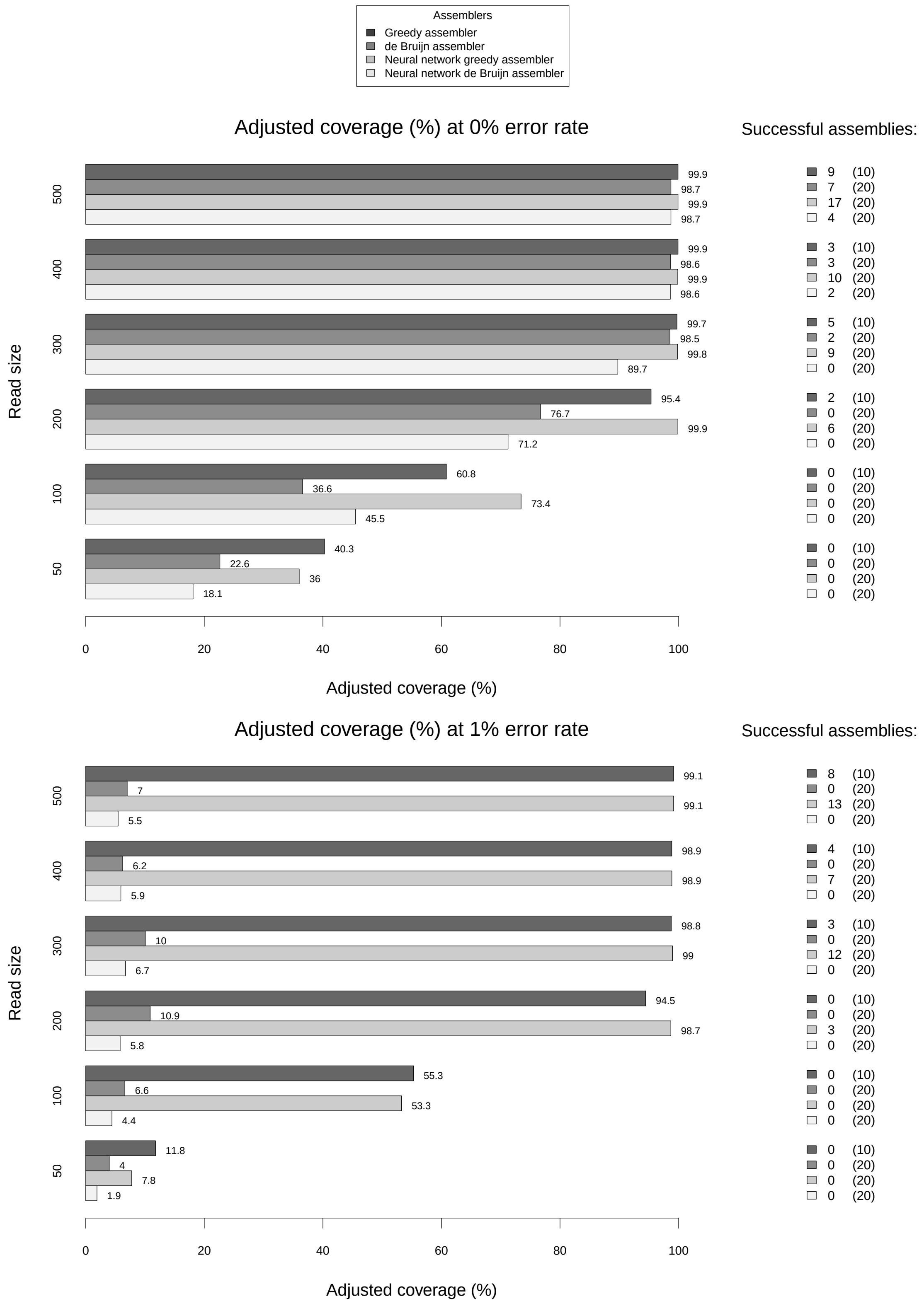


FIGURE 6.4: Adjusted coverages of best performing simulations at 10 times coverage depth. The plot shows performance at both 0% and 1% sequencing error rates along with the success rate of each assembler.

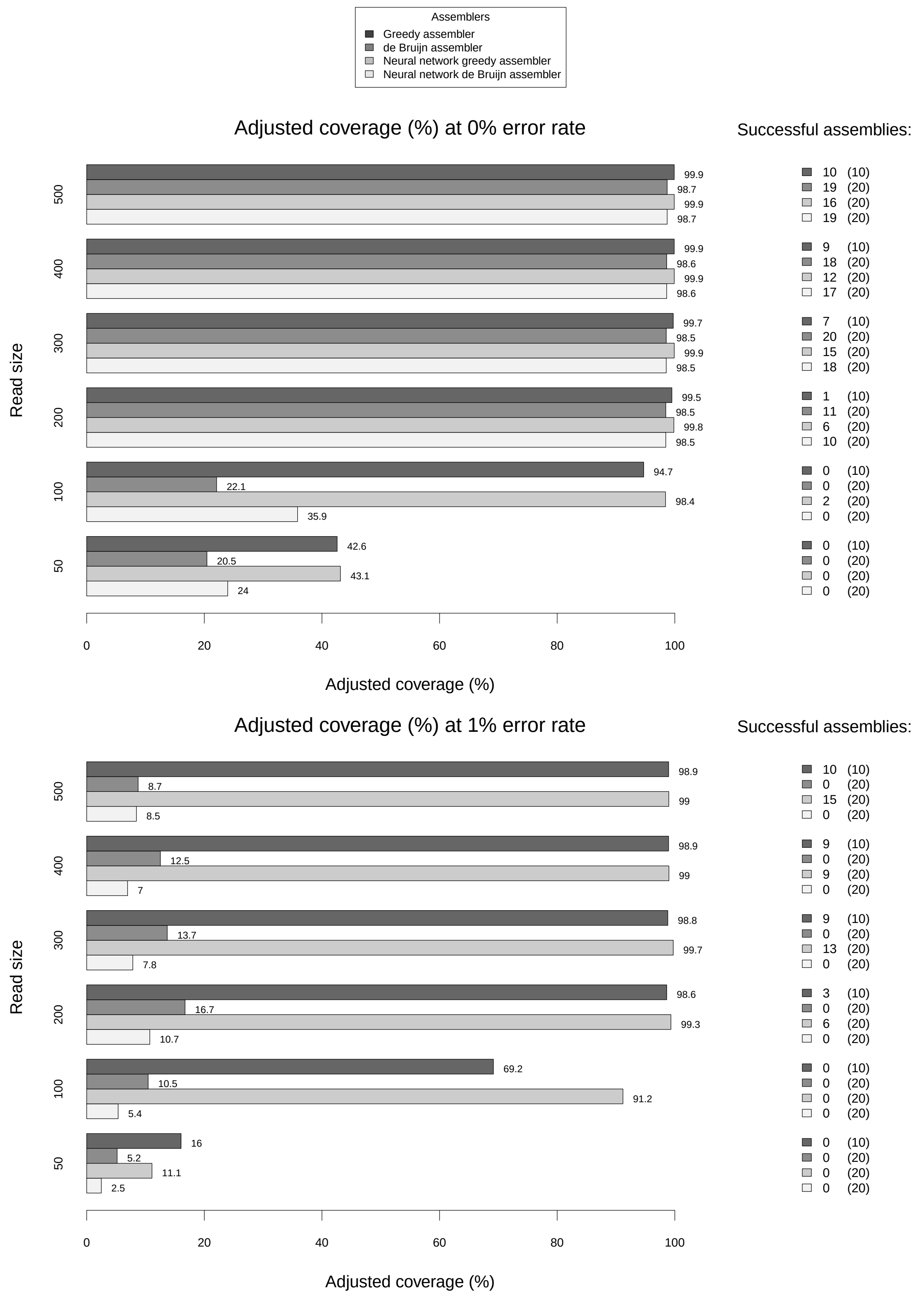


FIGURE 6.5: Adjusted coverages of best performing simulations at 15 times coverage depth. The plot shows performance at both 0% and 1% sequencing error rates along with the success rate of each assembler.

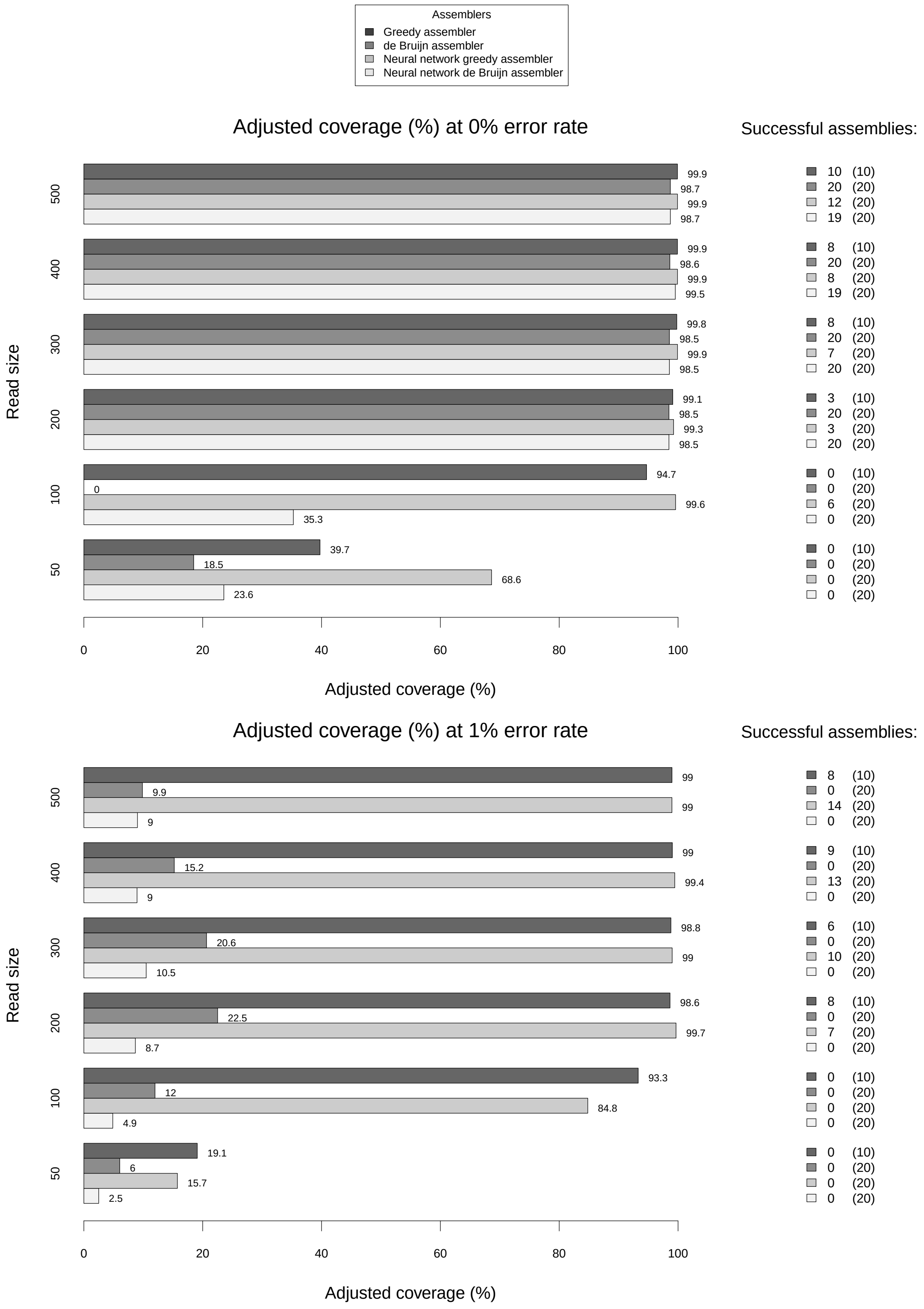


FIGURE 6.6: Adjusted coverages of best performing simulations at 20 times coverage depth. The plot shows performance at both 0% and 1% sequencing error rates along with the success rate of each assembler.

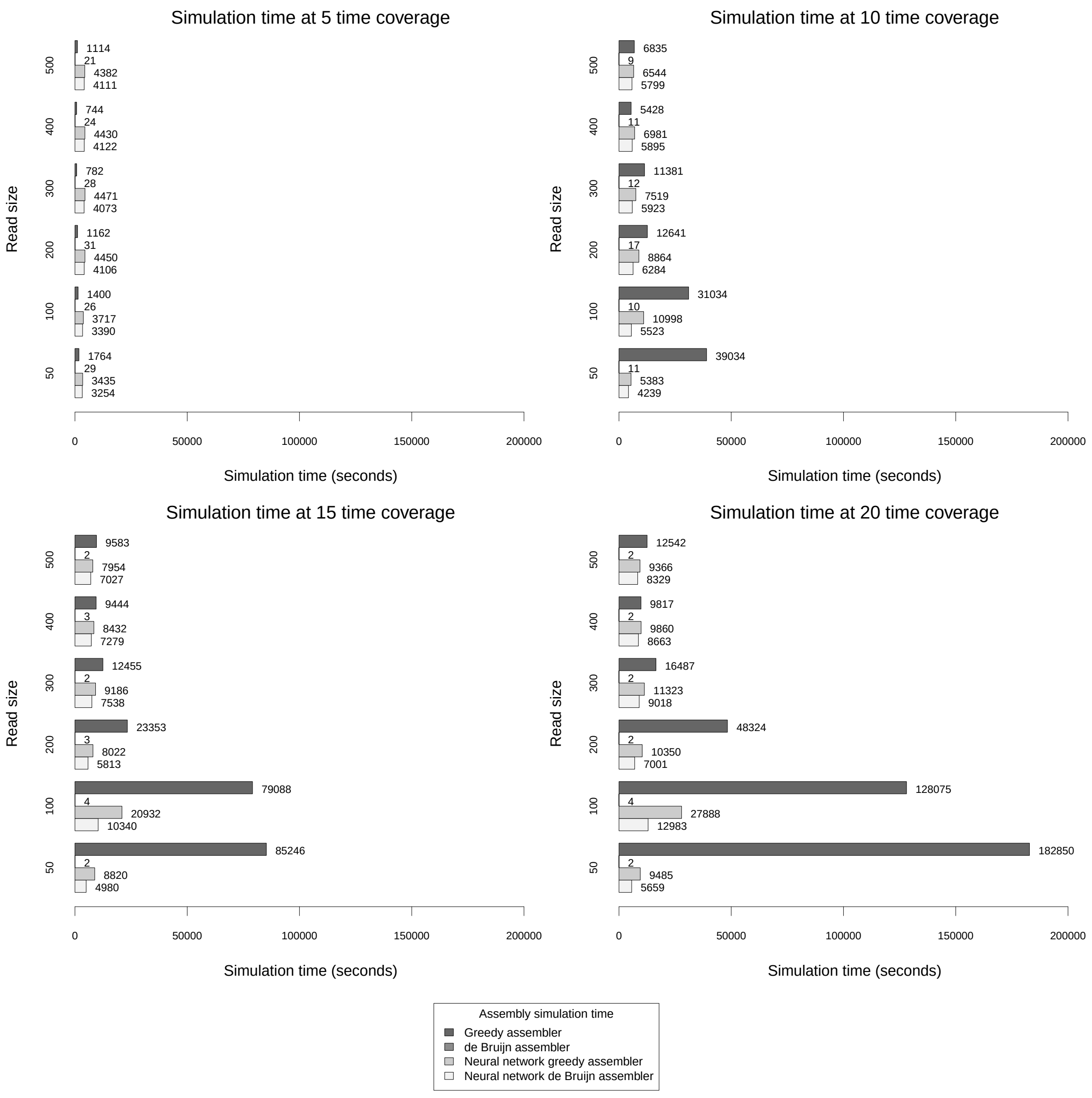


FIGURE 6.7: Simulation times of each assembler at 5, 10, 15 and 20 times coverage depths.

Chapter 7

Conclusion

The DNA sequencing and assembly problem was simulated under varying read size, coverage depth and error rate conditions on a 50456 base pair sequence obtained from the Fruit Fly (Drosophila Melanogaster) genome. A shotgun sequencing process was simulated to generate the reads used for assembly. Thereafter, four assembly strategies were simulated. The four assembly strategies were the greedy assembler, the de Bruijn assembler, the greedy neural network assembler and the de Bruijn neural network assembler. The simulations were performed in order to determine which of these assembly strategies achieved the highest coverage accuracy and with what computational complexity. Additionally, the research investigates the potential benefits of implementing a "divide and conquer" approach together with the greedy and de Bruijn assembly algorithms.

There are some limitations associated with this research. Firstly, the target sequence used was relatively small when compared to real-world assembly scenarios [6–8]. Secondly, the double stranded nature of DNA was ignored in order to simplify the research problem. Lastly, no validation or error correcting features, as found in the Velvet assembler [23], were used. However, a major strength of this research was the introduction of information theoretic principles which assist in establishing the correct parameters for successful assembly. A new adjusted coverage metric was also proposed which more accurately measures the assembly accuracy of each assembly strategy.

It was shown that under ideal conditions (low error, large reads and high coverage depths), the de Bruijn assembler outperforms the greedy assembler in terms of computational complexity and performs on par with the greedy assembler in terms of coverage performance. However, the introduction of high error rates to the shotgun sequencing process significantly reduced the coverage performance of the de Bruijn assembler. The greedy assembler on the other hand was more resilient to the introduction of this error and should therefore be used in the presence of higher sequencing error rates. note: when compared to the other one and recommended

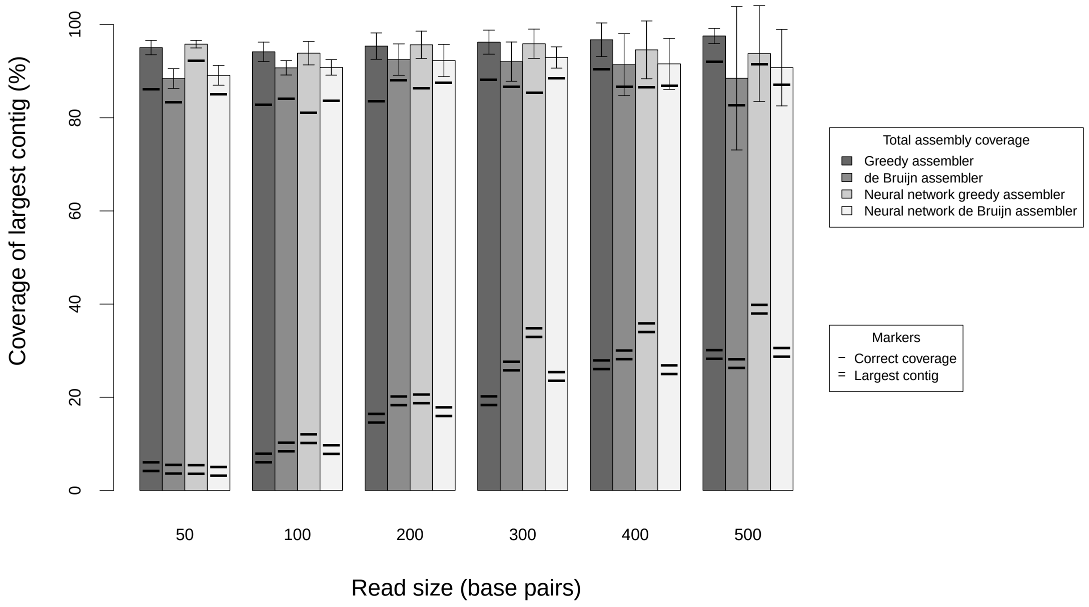
Additionally, it was shown that the computational performance of the greedy assembler can be significantly improved when implementing it together with the "divide and conquer" machine learning approach. Moreover, this increase in computational performance resulted in only a slight drop in assembly accuracy. Therefore, in situations when there are high error rates associated with the shotgun sequencing process, applying the neural network grouping together with the greedy assembler is always recommended as the slight drop in accuracy is negligible when compared to the increase gained in the complexity performance.

Implementing the neural network grouping scheme together with the de Bruijn assembler was deemed infeasible due to the training and grouping process introducing a higher complexity into the algorithm. For this reason, there remains opportunity for future research by investigating ways of reducing the complexity associated with the training and grouping process. More specifically, research can be done on unsupervised clustering machine learning algorithms in order to determine if a complexity of $O(N)$ is achievable. If the neural network training and read grouping process can be achieved in an $O(N \log N)$ complexity or lower, then the overhead introduced will not significantly compromise the computational complexity of the de Bruijn assembly algorithm. Future work can also exploit the parallelisable nature of the neural network grouping scheme. In this manner, the neural network and grouping process can benefit from the abundance of computing power available in today's world.

Appendix A

Coverage statistics

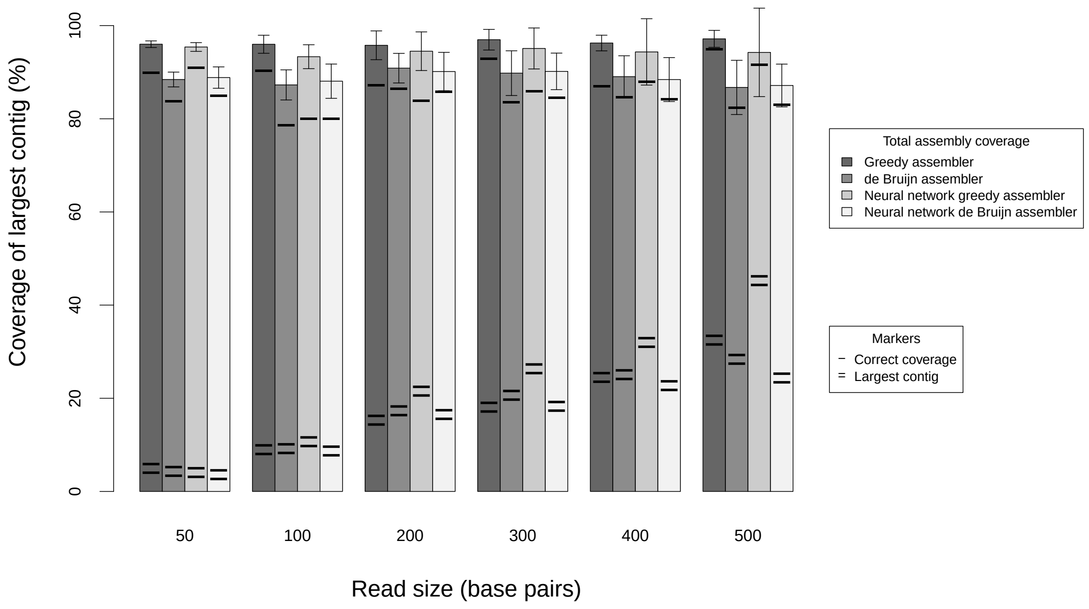
The figures contained in this appendix display all supporting results presented and discussed in Chapter 6.



Total number of contigs generated:

■ 95.5	■ 45.7	■ 22.1	■ 15.2	■ 10.1	■ 8.9
▨ 101.05	▨ 38.3	▨ 18.2	▨ 10.85	▨ 8.25	▨ 7.25
░ 115.7	░ 33.3	░ 15.4	░ 9.45	░ 7.2	░ 6.3
□ 121.6	□ 40.7	□ 19.4	□ 12.1	□ 9.15	□ 7.75

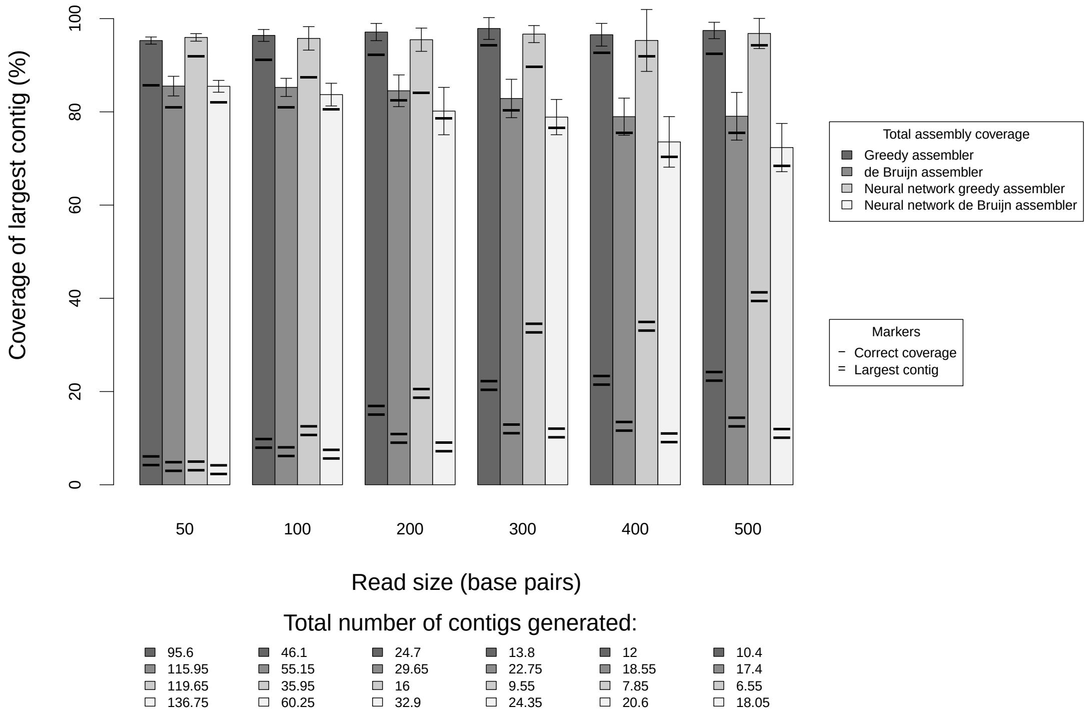
Figure A.1: Total contig coverage, correct contig coverage and largest contigs at varying read sizes, 5 times coverage depth and 0% sequencing error rate



Total number of contigs generated:

■ 100.8	■ 45.1	■ 22.1	■ 15.1	■ 11.7	■ 9.1
▨ 105.65	▨ 40.3	▨ 17.5	▨ 12.8	▨ 9.7	▨ 8.9
░ 117.4	░ 33	░ 14.65	░ 9.6	░ 7.9	░ 6.4
□ 124.1	□ 42.55	□ 19	□ 14.35	□ 11.05	□ 9.7

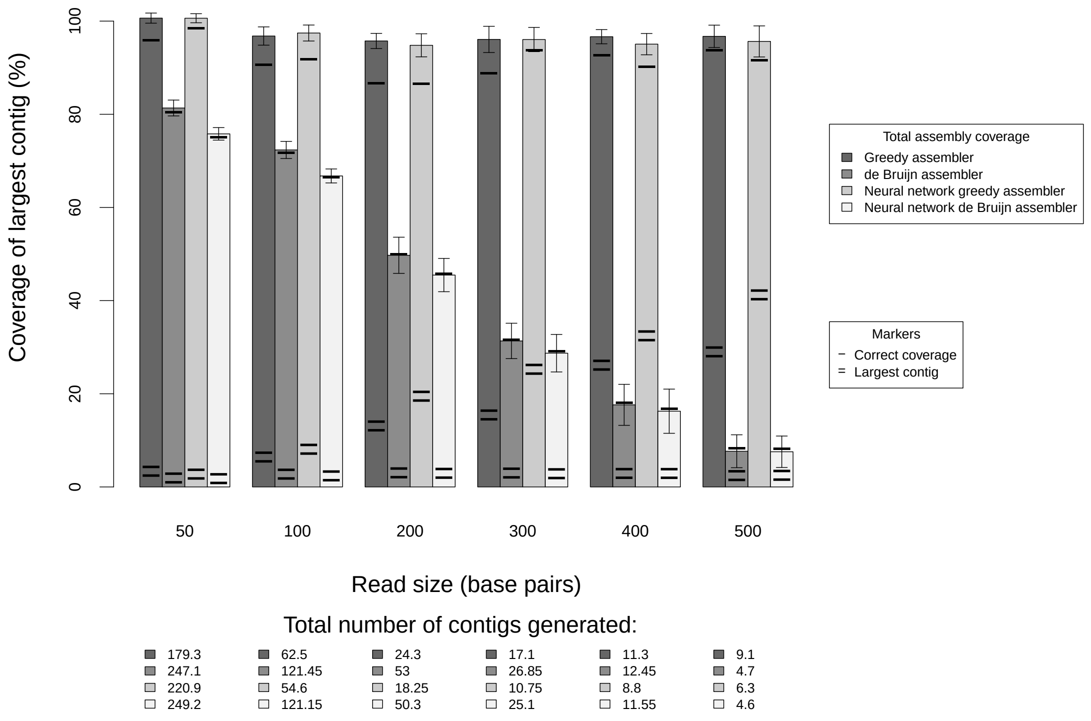
Figure A.2: Total contig coverage, correct contig coverage and largest contigs at varying read sizes, 5 times coverage depth and 0.01% sequencing error rate



Total number of contigs generated:

	50	100	200	300	400	500
■	95.6	46.1	24.7	13.8	12	10.4
▨	115.95	55.15	29.65	22.75	18.55	17.4
▧	119.65	35.95	16	9.55	7.85	6.55
□	136.75	60.25	32.9	24.35	20.6	18.05

FIGURE A.3: Total contig coverage, correct contig coverage and largest contigs at varying read sizes, 5 times coverage depth and 0.1% sequencing error rate



Total number of contigs generated:

	50	100	200	300	400	500
■	179.3	62.5	24.3	17.1	11.3	9.1
▨	247.1	121.45	53	26.85	12.45	4.7
▧	220.9	54.6	18.25	10.75	8.8	6.3
□	249.2	121.15	50.3	25.1	11.55	4.6

FIGURE A.4: Total contig coverage, correct contig coverage and largest contigs at varying read sizes, 5 times coverage depth and 1% sequencing error rate

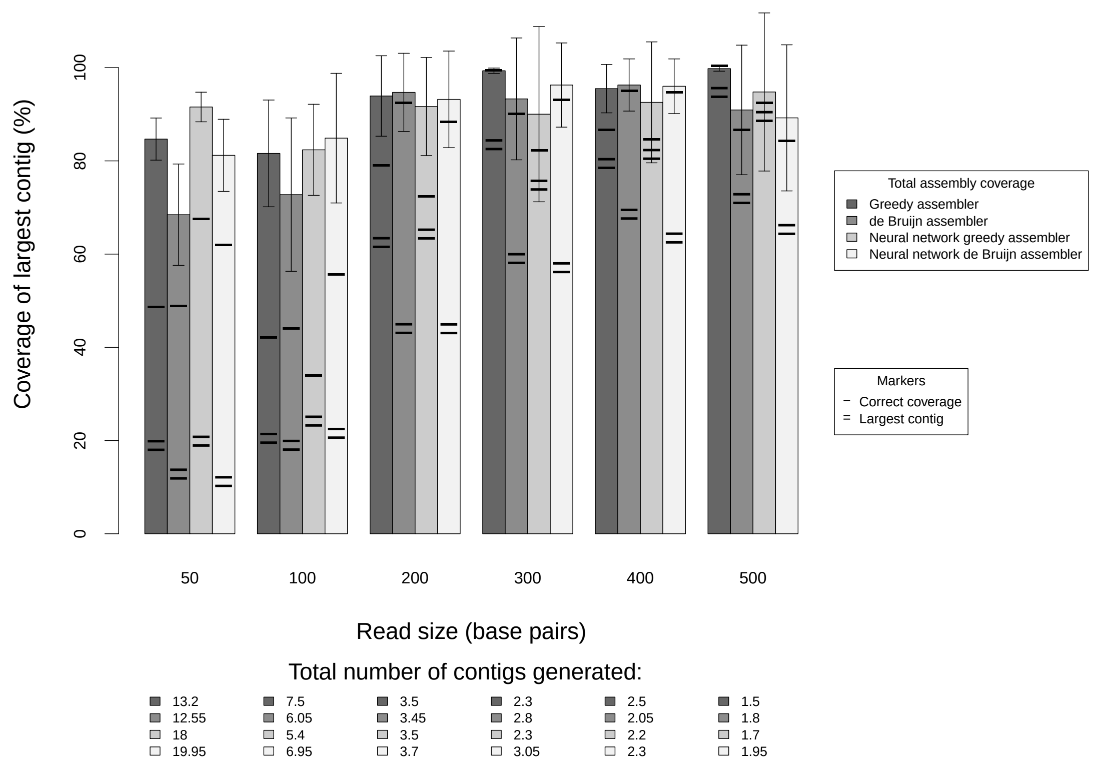


FIGURE A.5: Total contig coverage, correct contig coverage and largest contigs at varying read sizes, 10 times coverage depth and 0% sequencing error rate

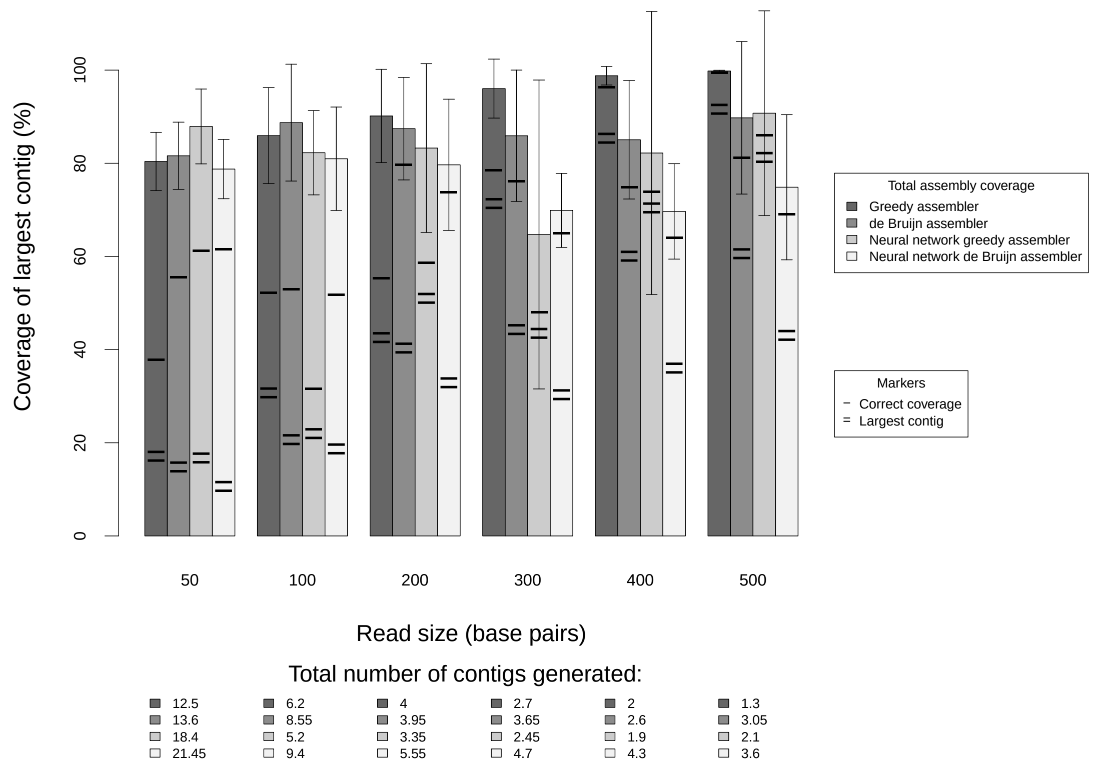
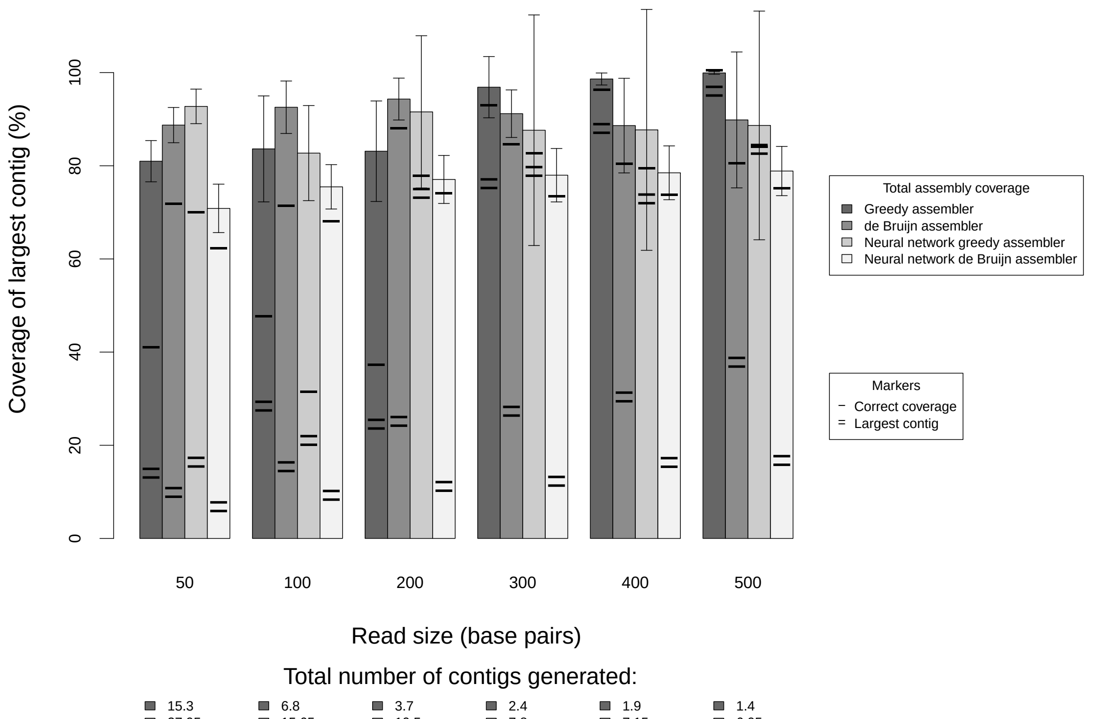


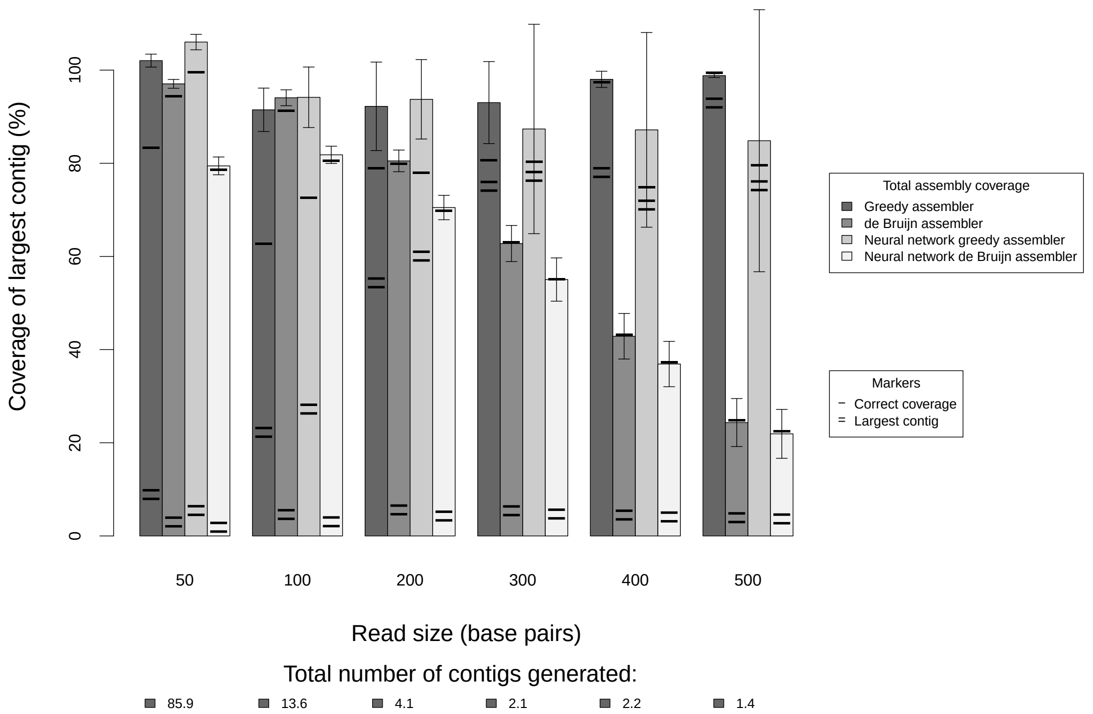
FIGURE A.6: Total contig coverage, correct contig coverage and largest contigs at varying read sizes, 10 times coverage depth and 0.01% sequencing error rate



Total number of contigs generated:

■ 15.3	■ 6.8	■ 3.7	■ 2.4	■ 1.9	■ 1.4
▩ 27.95	▩ 15.65	▩ 10.5	▩ 7.8	▩ 7.15	▩ 6.05
▨ 22	▨ 5.65	▨ 2.55	▨ 2.15	▨ 1.85	▨ 1.8
□ 43.4	□ 29.6	□ 23.15	□ 18.4	□ 14.1	□ 14.1

Figure A.7: Total contig coverage, correct contig coverage and largest contigs at varying read sizes, 10 times coverage depth and 0.1% sequencing error rate



Total number of contigs generated:

■ 85.9	■ 13.6	■ 4.1	■ 2.1	■ 2.2	■ 1.4
▩ 177.5	▩ 93.75	▩ 53.3	▩ 37.05	▩ 25.05	▩ 12.9
▨ 122.65	▨ 14.55	▨ 4.1	▨ 1.95	▨ 2.2	▨ 1.5
□ 224.55	□ 123.2	□ 58.4	□ 36.85	□ 22.85	□ 11.8

Figure A.8: Total contig coverage, correct contig coverage and largest contigs at varying read sizes, 10 times coverage depth and 1% sequencing error rate

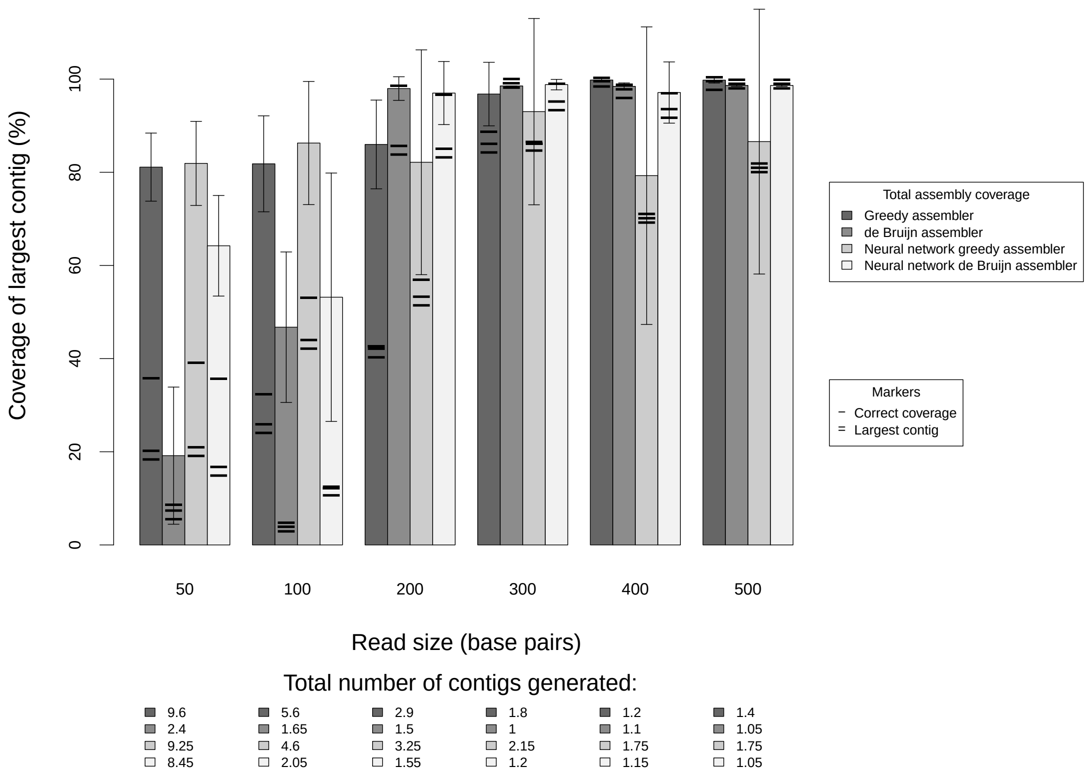


FIGURE A.9: Total contig coverage, correct contig coverage and largest contigs at varying read sizes, 15 times coverage depth and 0% sequencing error rate

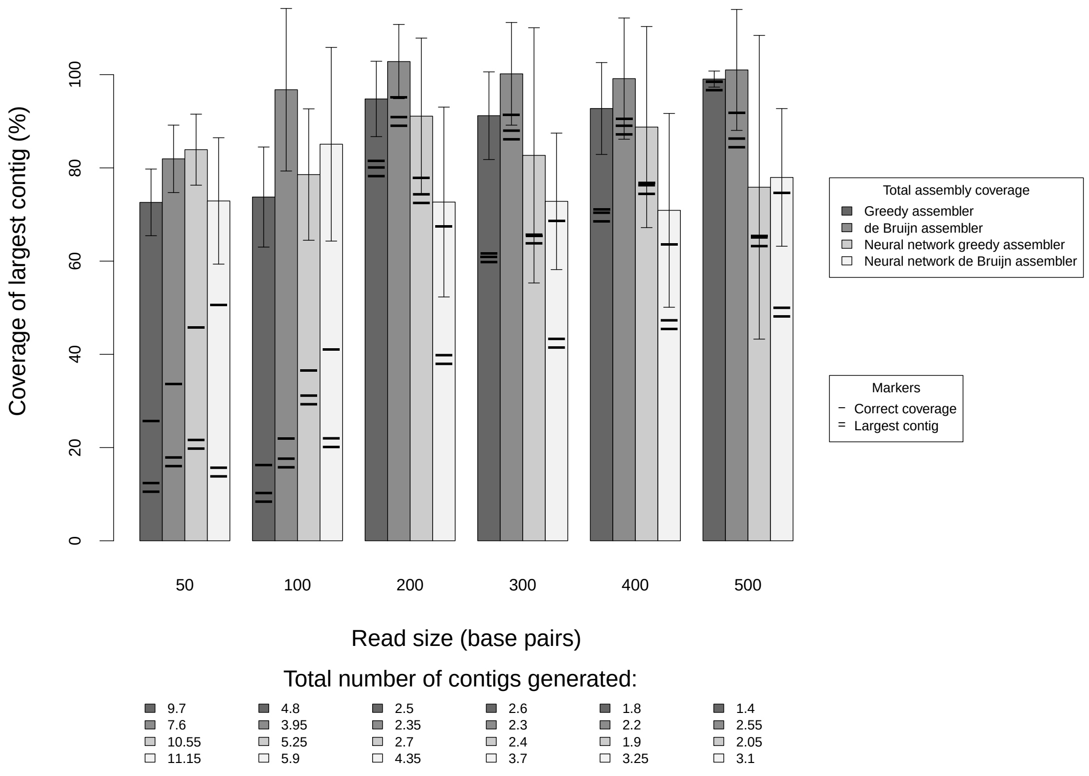
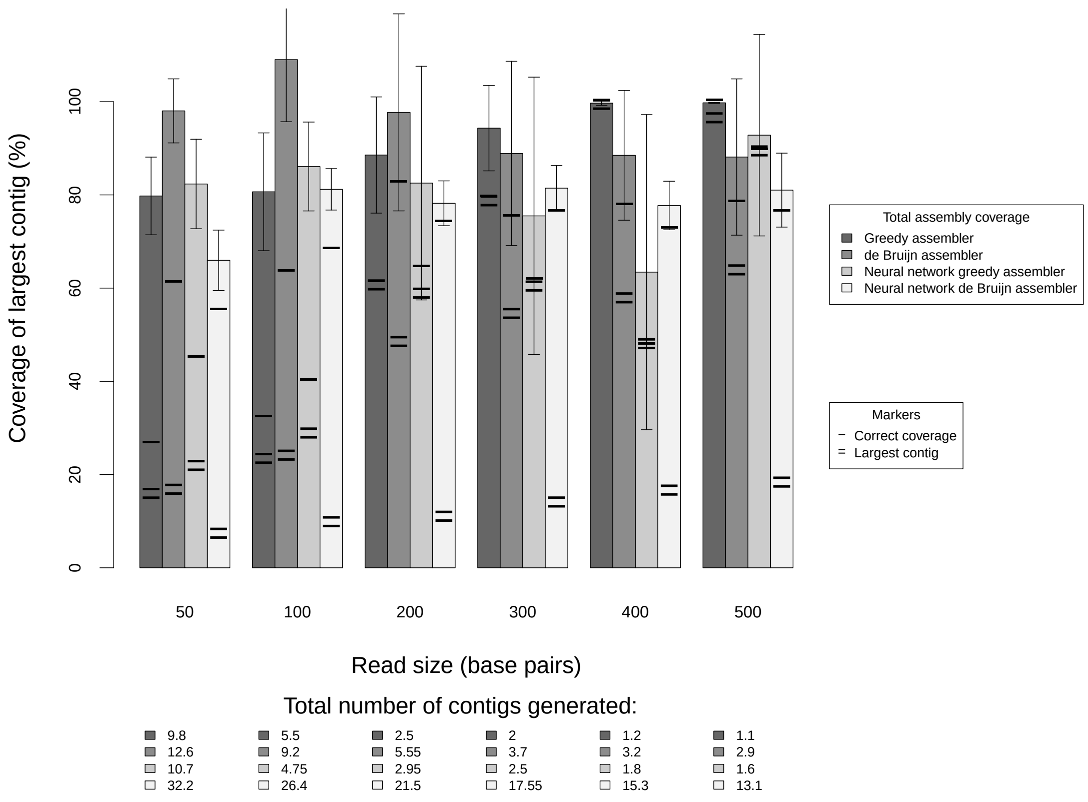


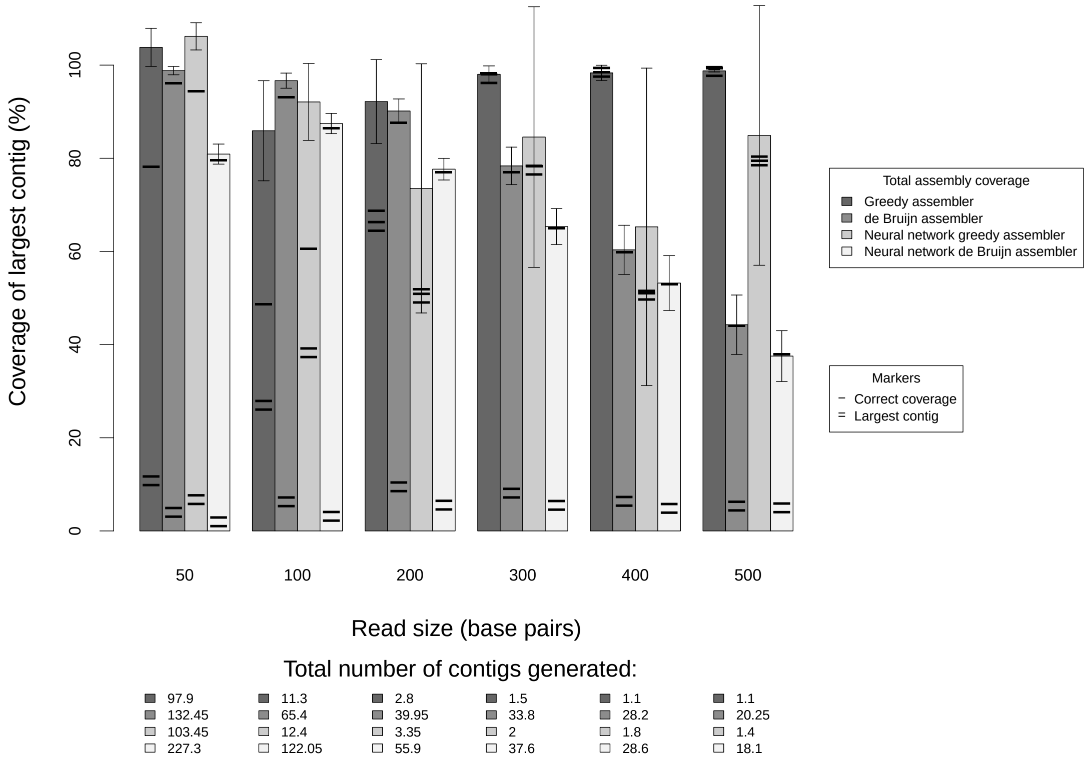
FIGURE A.10: Total contig coverage, correct contig coverage and largest contigs at varying read sizes, 15 times coverage depth and 0.01% sequencing error rate



Total number of contigs generated:

	50	100	200	300	400	500
■	9.8	5.5	2.5	2	1.2	1.1
▨	12.6	9.2	5.55	3.7	3.2	2.9
▧	10.7	4.75	2.95	2.5	1.8	1.6
□	32.2	26.4	21.5	17.55	15.3	13.1

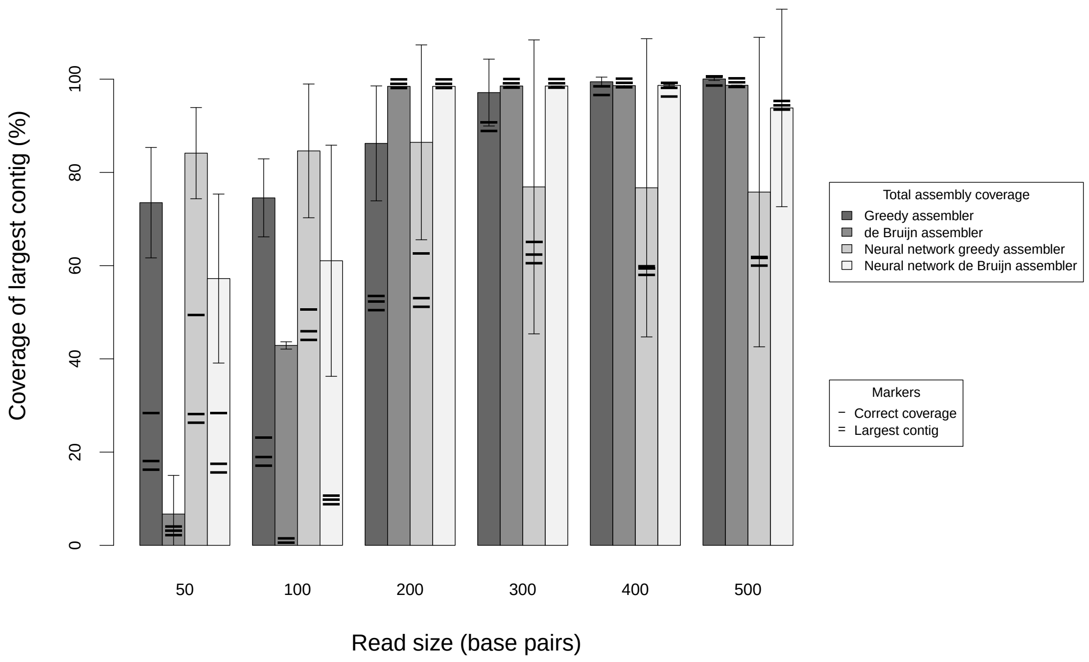
FIGURE A.11: Total contig coverage, correct contig coverage and largest contigs at varying read sizes, 15 times coverage depth and 0.1% sequencing error rate



Total number of contigs generated:

	50	100	200	300	400	500
■	97.9	11.3	2.8	1.5	1.1	1.1
▨	132.45	65.4	39.95	33.8	28.2	20.25
▧	103.45	12.4	3.35	2	1.8	1.4
□	227.3	122.05	55.9	37.6	28.6	18.1

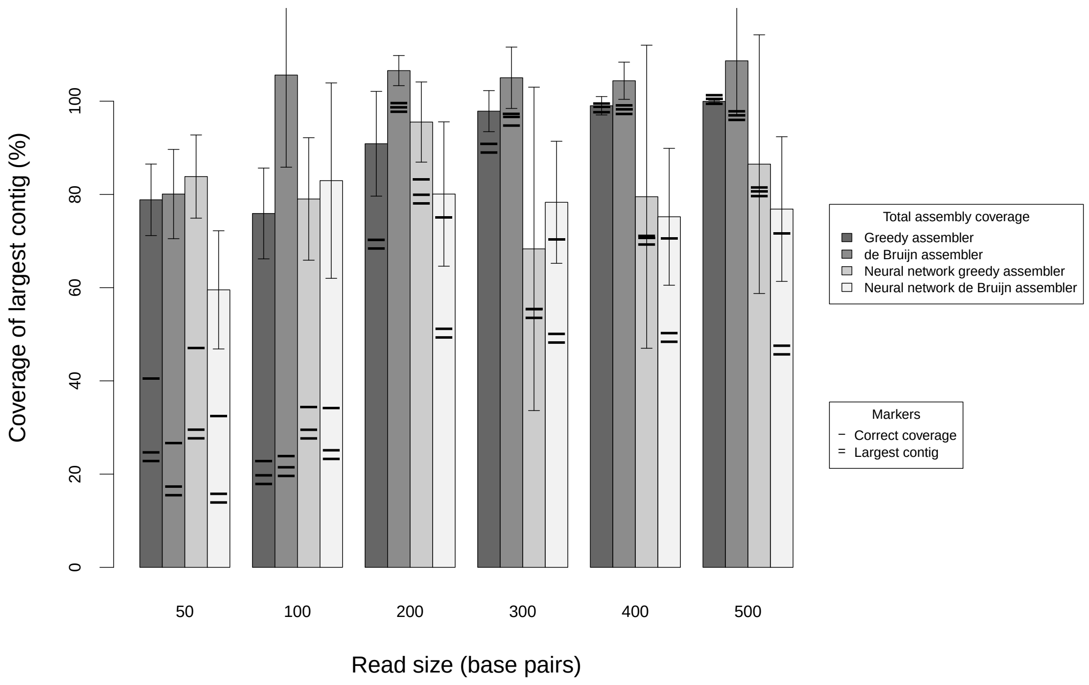
FIGURE A.12: Total contig coverage, correct contig coverage and largest contigs at varying read sizes, 15 times coverage depth and 1% sequencing error rate



Total number of contigs generated:

	50	100	200	300	400	500
Greedy assembler	9.3	5.3	2.6	1.7	1.5	1.3
de Bruijn assembler	1.4	1.05	1	1	1	1
Neural network greedy assembler	8.25	4.45	3.3	2.9	2.2	1.95
Neural network de Bruijn assembler	5.6	1.65	1	1	1.05	1

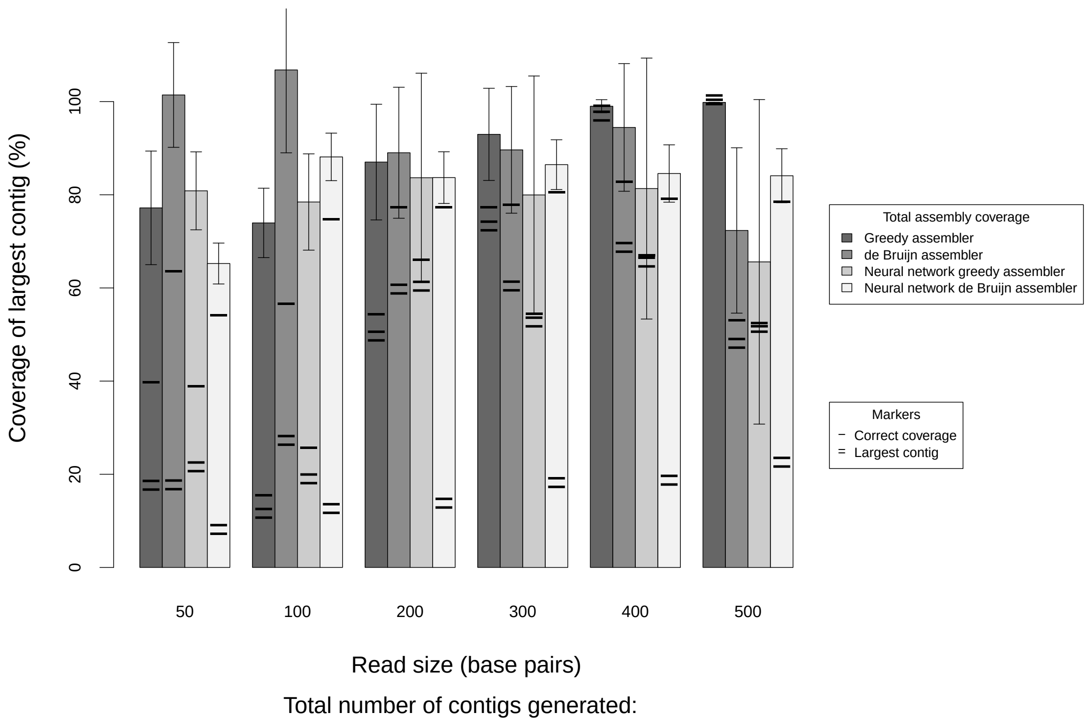
FIGURE A.13: Total contig coverage, correct contig coverage and largest contigs at varying read sizes, 20 times coverage depth and 0% sequencing error rate



Total number of contigs generated:

	50	100	200	300	400	500
Greedy assembler	10.1	4.9	2.7	1.7	1.3	1.1
de Bruijn assembler	6.4	3.45	2.15	2.05	1.95	2.5
Neural network greedy assembler	8.7	4.8	2.7	2.2	1.9	1.7
Neural network de Bruijn assembler	7.8	4.55	3.4	3.4	3.45	3.45

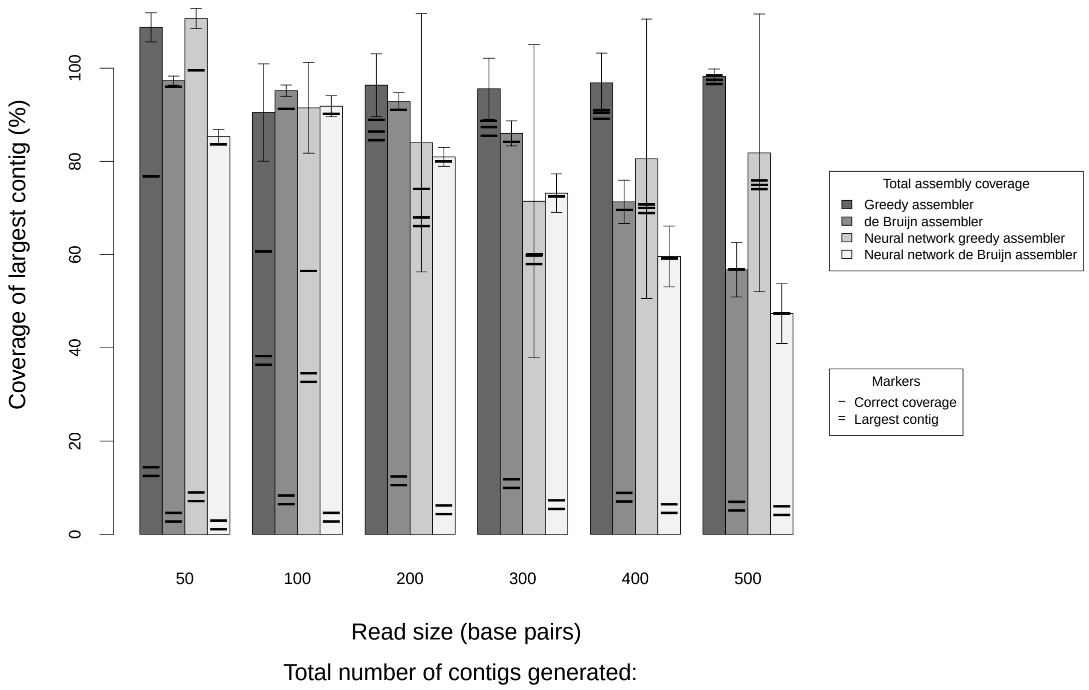
FIGURE A.14: Total contig coverage, correct contig coverage and largest contigs at varying read sizes, 20 times coverage depth and 0.01% sequencing error rate



Total number of contigs generated:

■ 10	■ 5.2	■ 2.8	■ 2.4	■ 1.5	■ 1
▩ 12.8	▩ 7	▩ 3.45	▩ 2.65	▩ 3.1	▩ 2.25
▨ 7.95	▨ 4.6	▨ 2.95	▨ 2.6	▨ 2.15	▨ 1.9
□ 30.55	□ 22.8	□ 18.85	□ 15.45	□ 13.5	□ 11.55

FIGURE A.15: Total contig coverage, correct contig coverage and largest contigs at varying read sizes, 20 times coverage depth and 0.1% sequencing error rate



Total number of contigs generated:

■ 122.8	■ 12.9	■ 2.2	■ 1.8	■ 1.3	■ 1.2
▩ 154.7	▩ 59.1	▩ 31.3	▩ 29	▩ 28.5	▩ 23.1
▨ 112.6	▨ 13.65	▨ 3.05	▨ 2.1	▨ 1.65	▨ 1.6
□ 234.75	□ 118.85	□ 53.4	□ 37.65	□ 29.35	□ 21.4

FIGURE A.16: Total contig coverage, correct contig coverage and largest contigs at varying read sizes, 20 times coverage depth and 1% sequencing error rate

Appendix B

Adjusted Coverage

The figures contained in this appendix display all supporting results presented and discussed in Chapter 6.

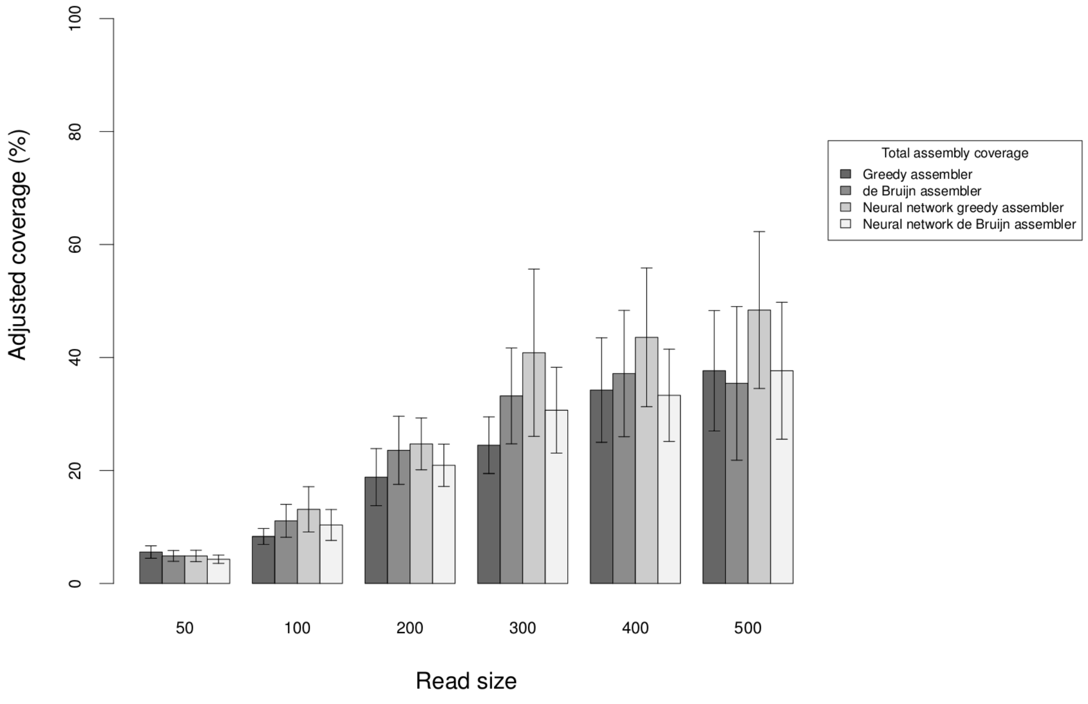


FIGURE B.1: Adjusted coverage at 5 times coverage depth and 0% sequencing error rate

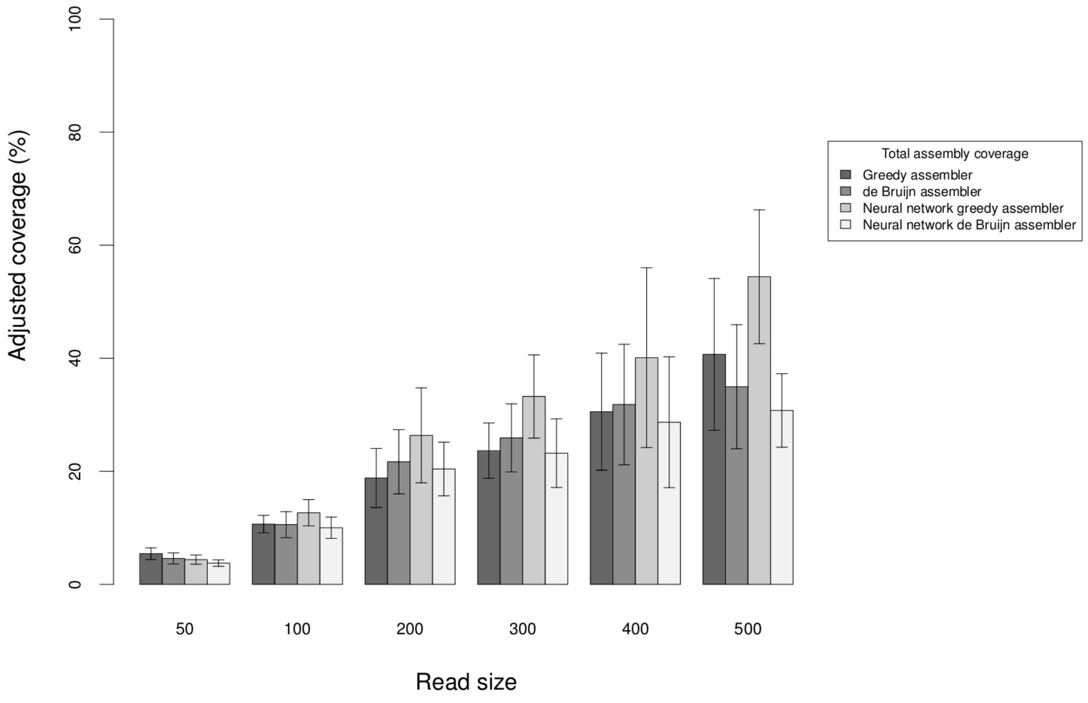


FIGURE B.2: Adjusted coverage at 5 times coverage depth and 0.01% sequencing error rate

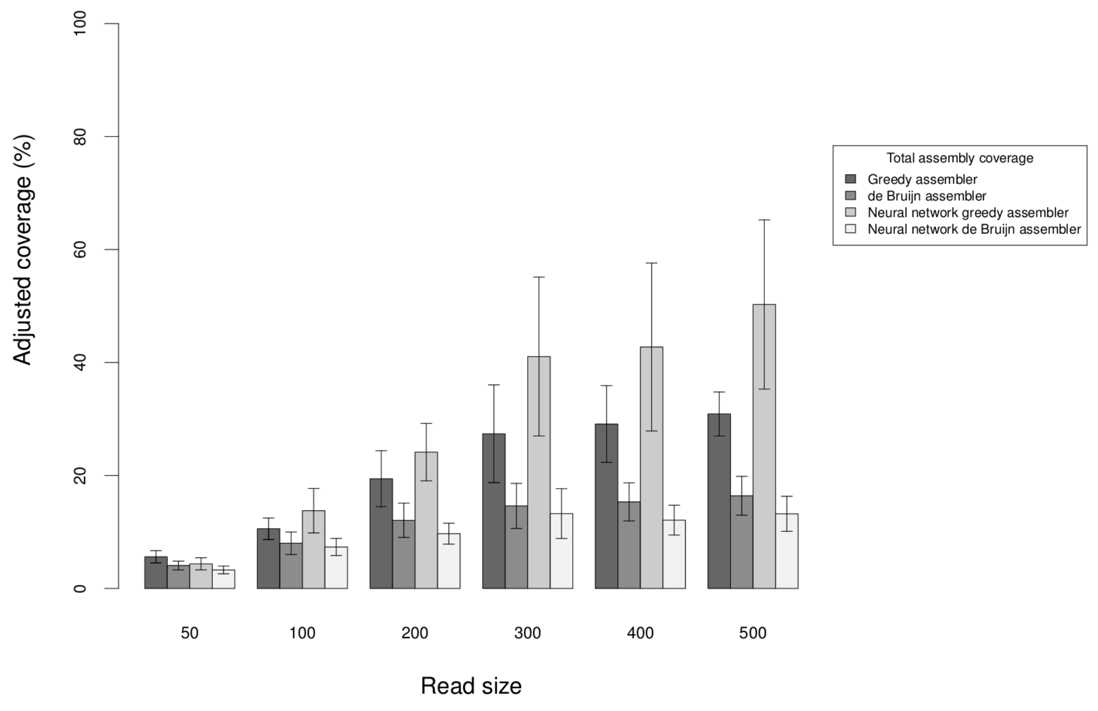


FIGURE B.3: Adjusted coverage at 5 times coverage depth and 0.1% sequencing error rate

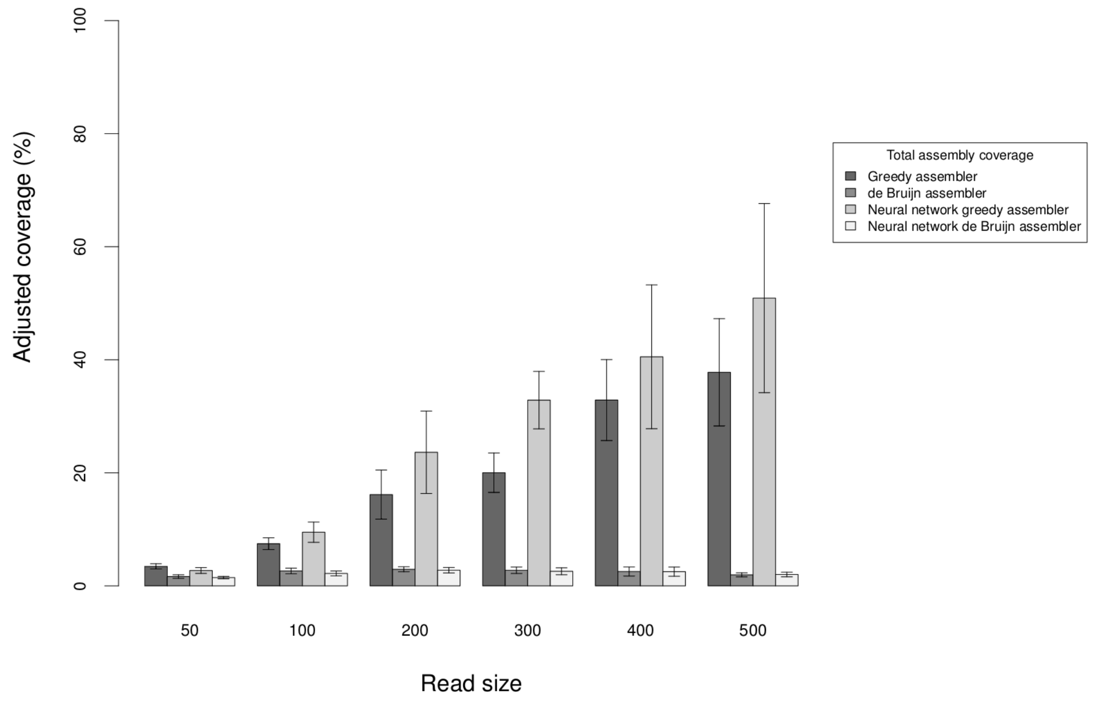


FIGURE B.4: Adjusted coverage at 5 times coverage depth and 1% sequencing error rate

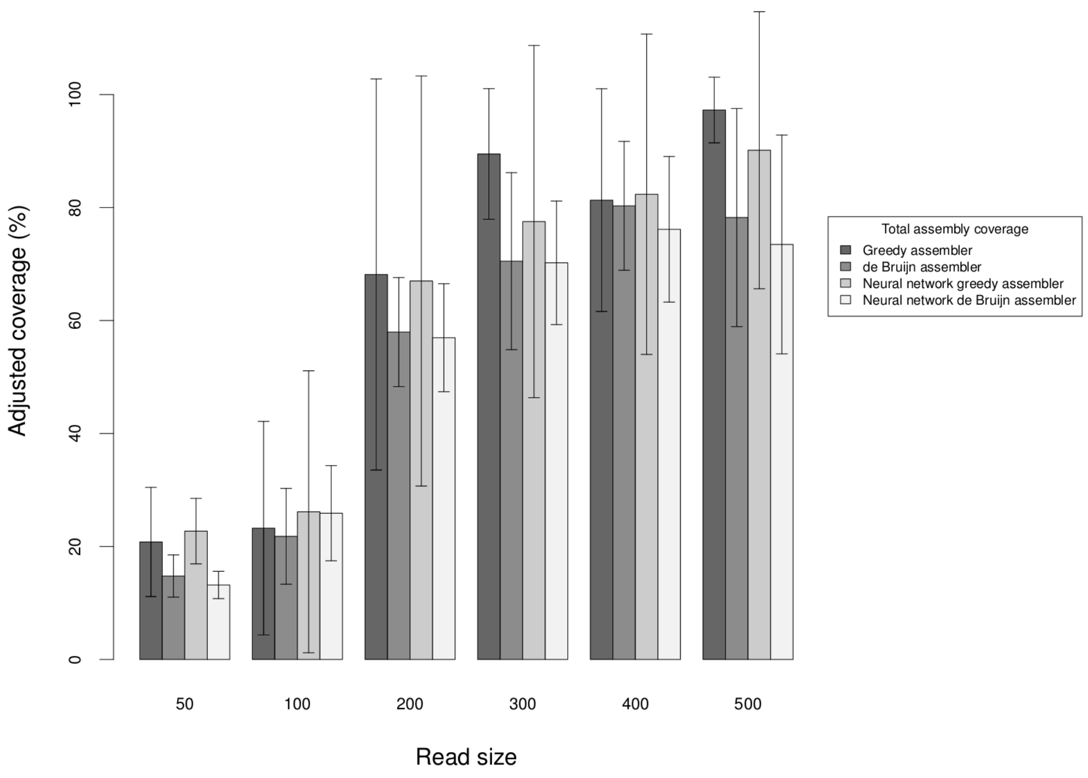


FIGURE B.5: Adjusted coverage at 10 times coverage depth and 0% sequencing error rate

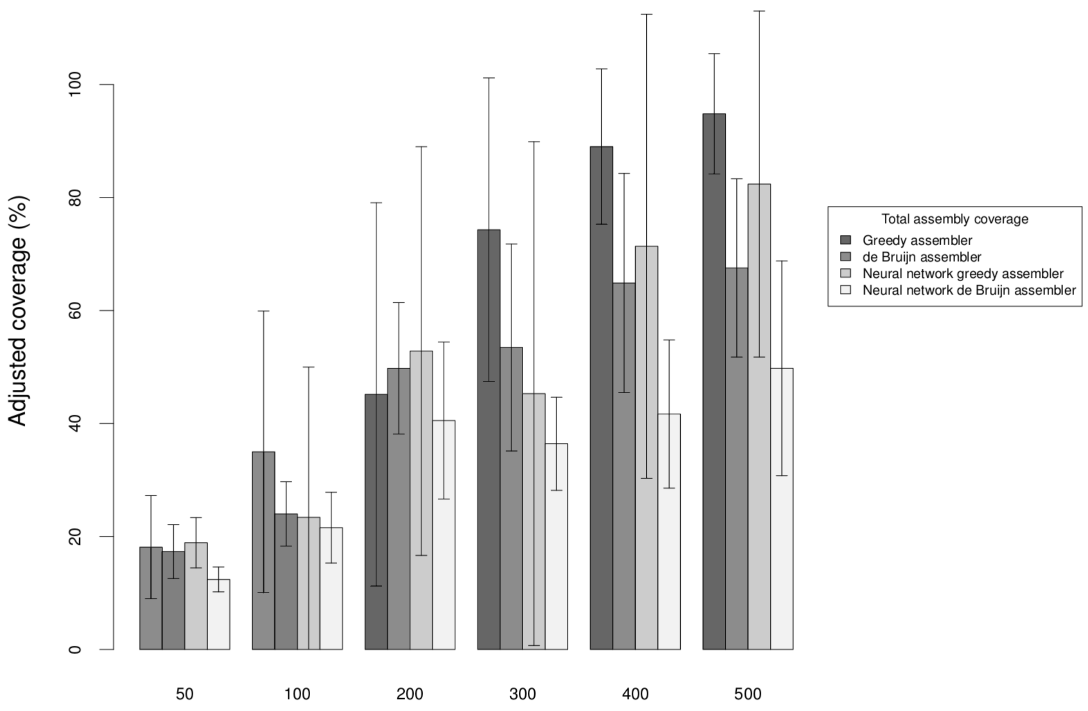


FIGURE B.6: Adjusted coverage at 10 times coverage depth and 0.01% sequencing error rate

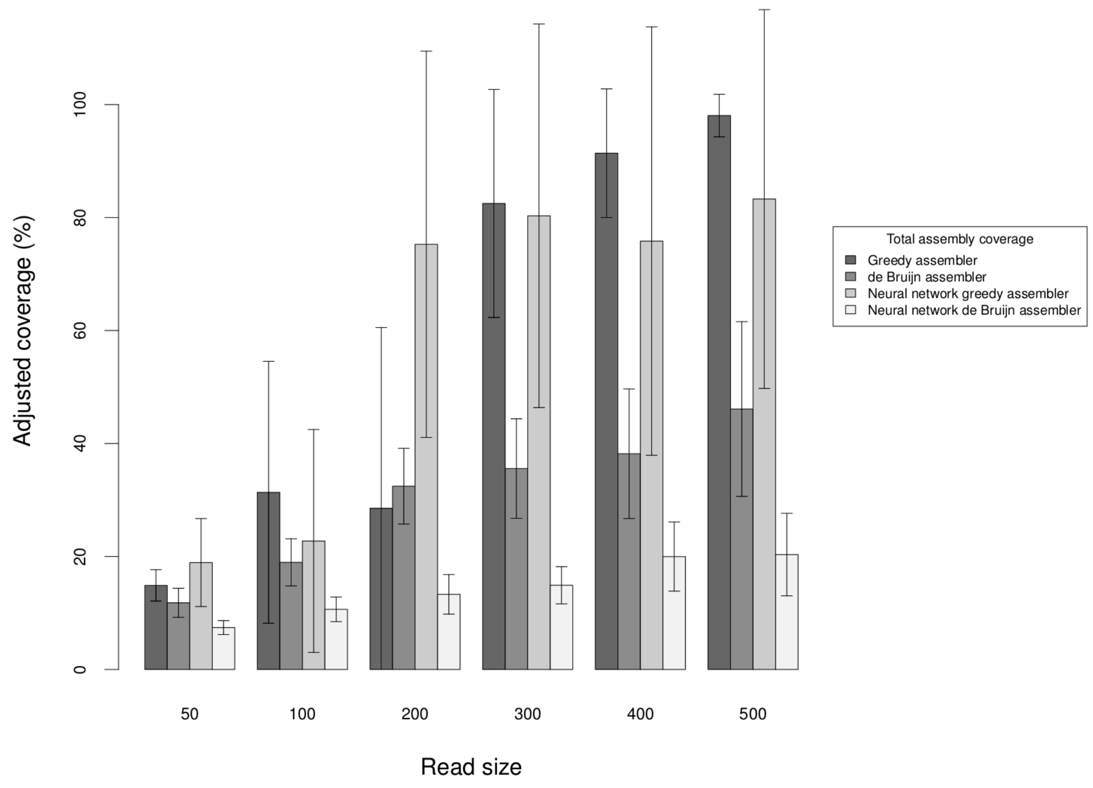


FIGURE B.7: Adjusted coverage at 10 times coverage depth and 0.1% sequencing error rate

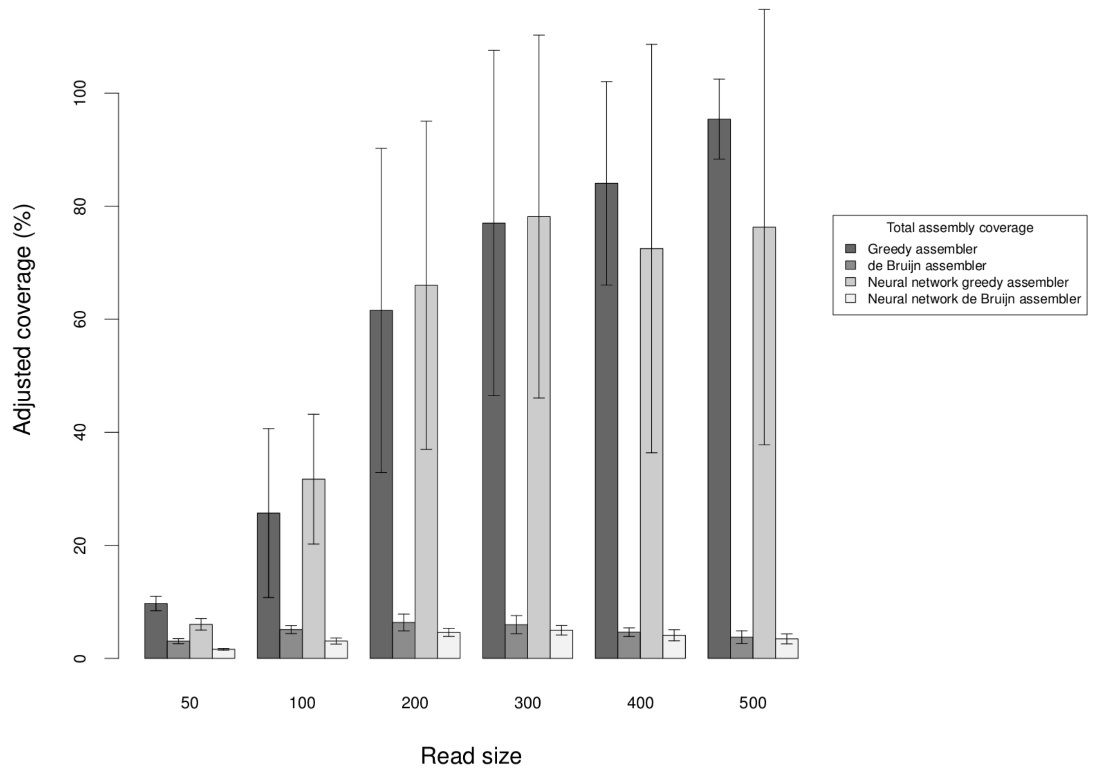


FIGURE B.8: Adjusted coverage at 10 times coverage depth and 1% sequencing error rate

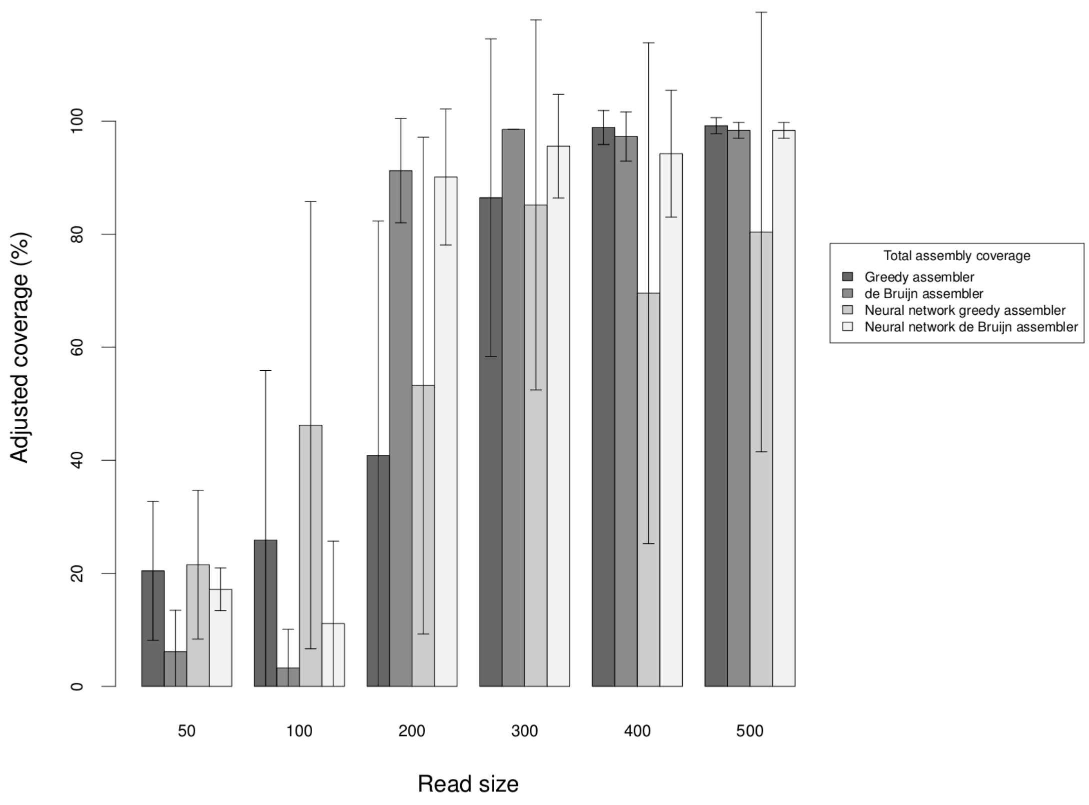


FIGURE B.9: Adjusted coverage at 15 times coverage depth and 0% sequencing error rate

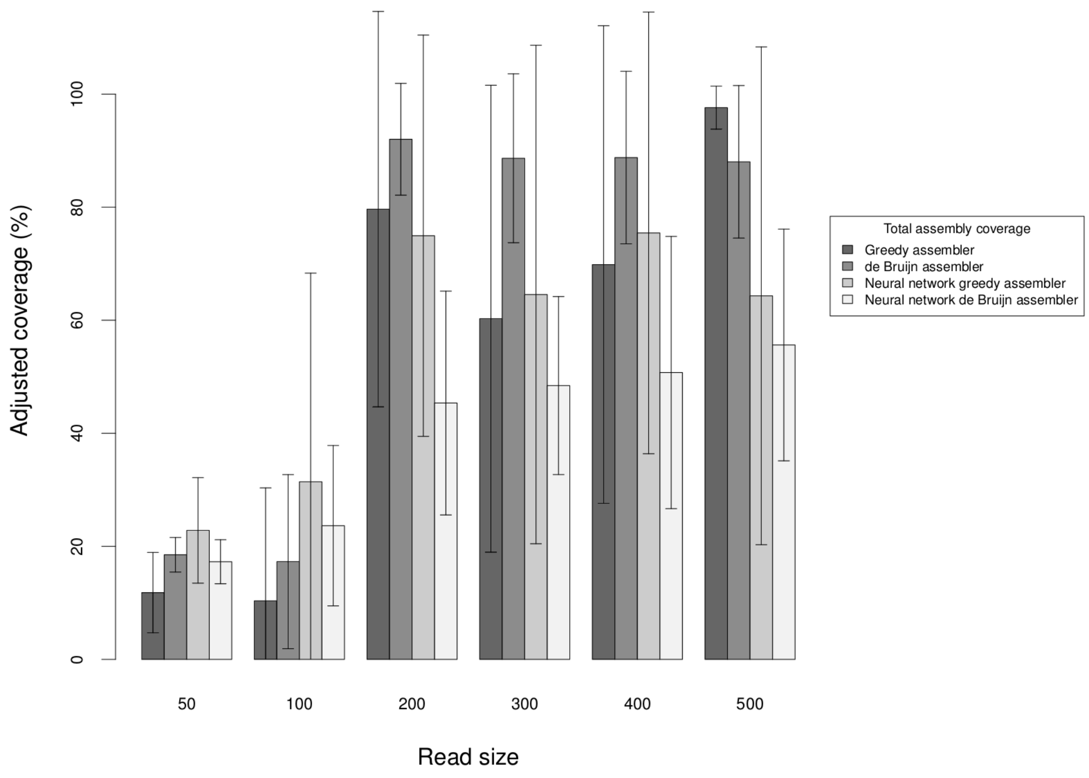


FIGURE B.10: Adjusted coverage at 15 times coverage depth and 0.01% sequencing error rate

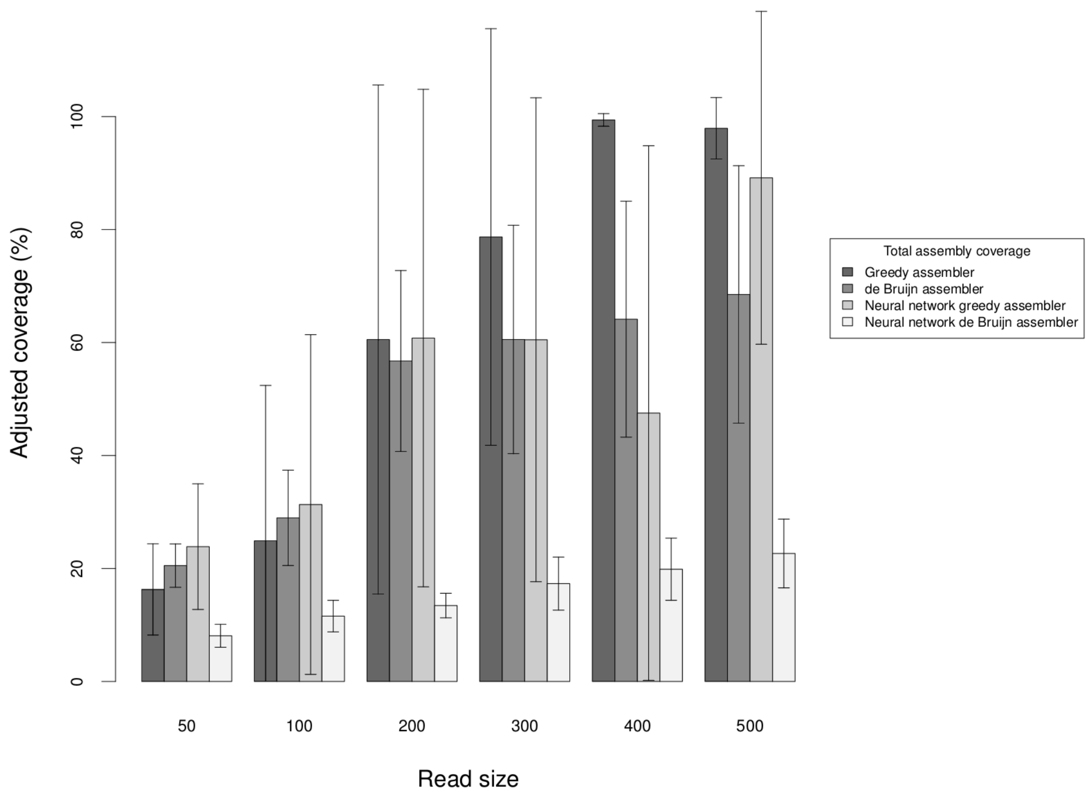


FIGURE B.11: Adjusted coverage at 15 times coverage depth and 0.1% sequencing error rate

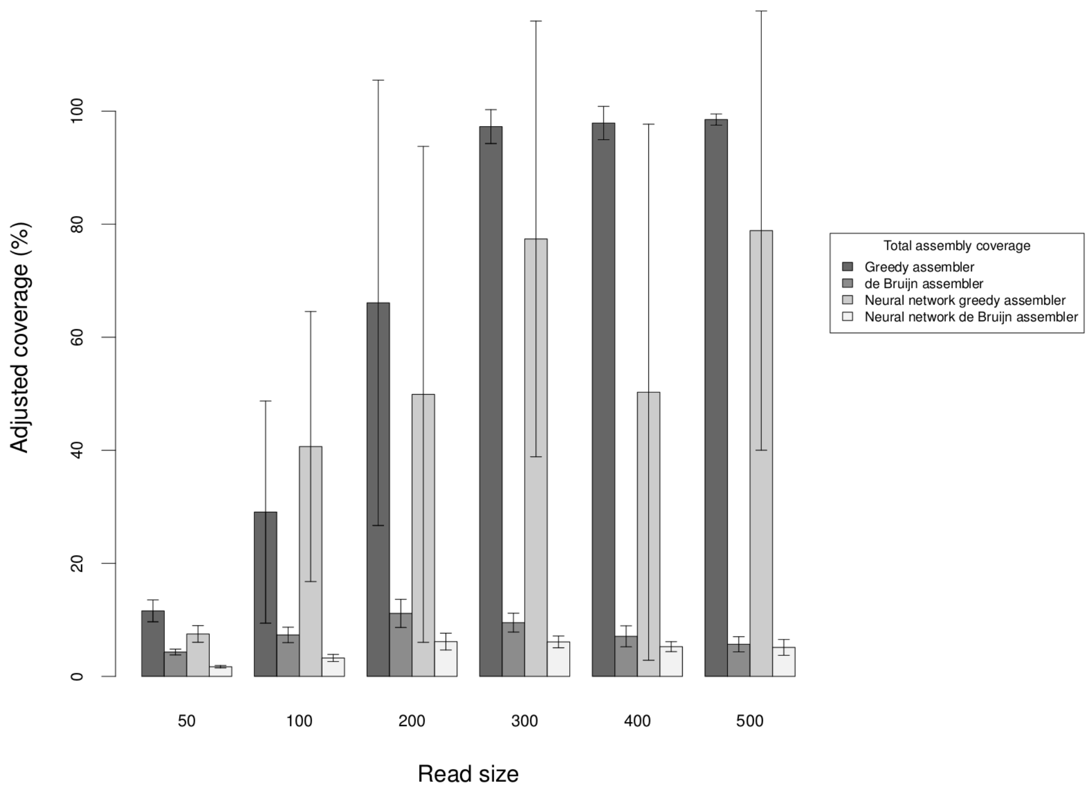


FIGURE B.12: Adjusted coverage at 15 times coverage depth and 1% sequencing error rate

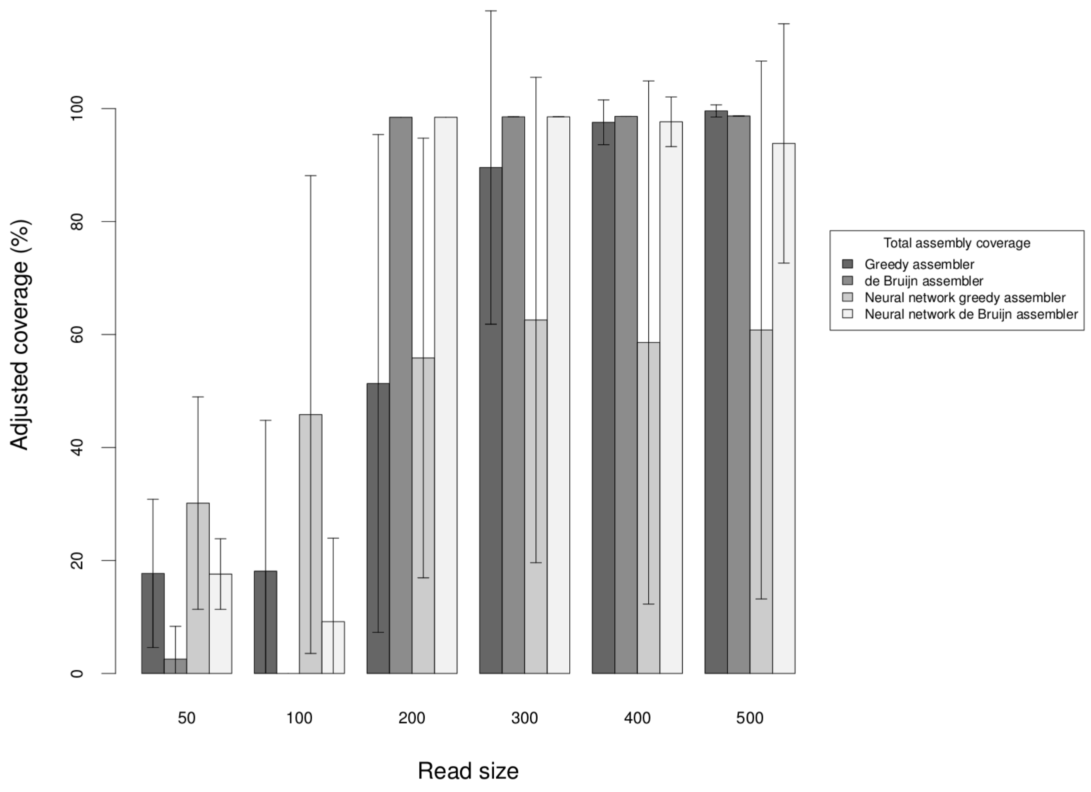


FIGURE B.13: Adjusted coverage at 20 times coverage depth and 0% sequencing error rate

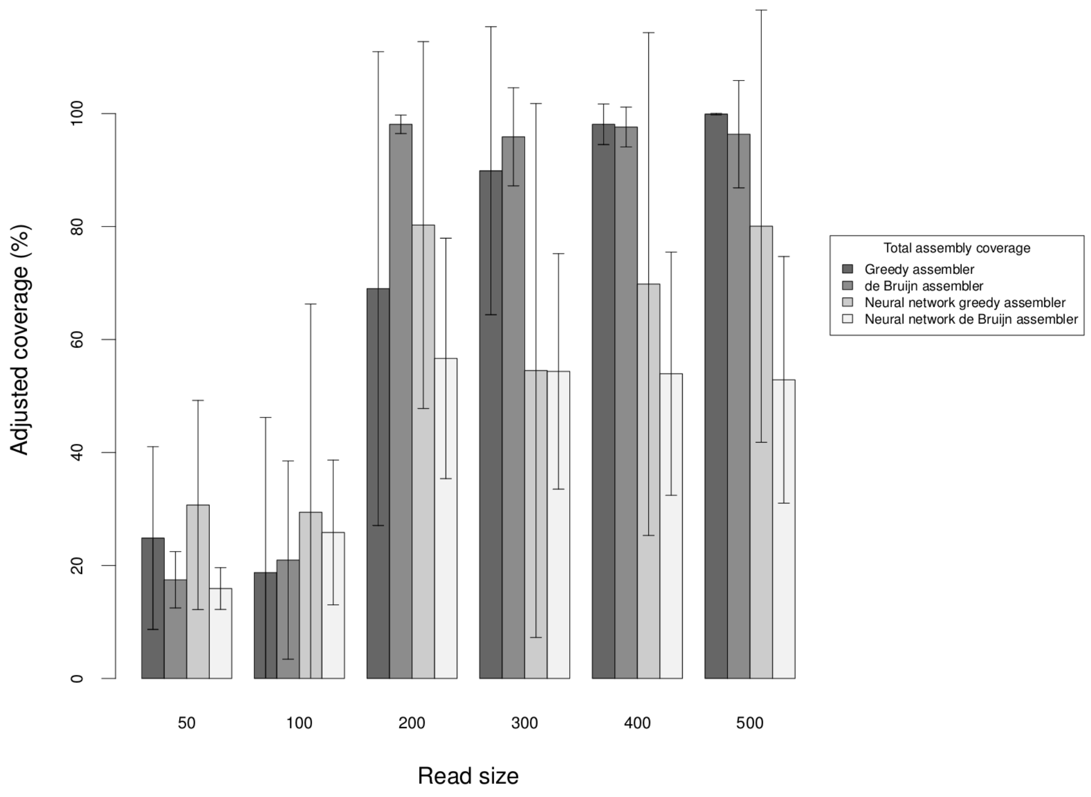


FIGURE B.14: Adjusted coverage at 20 times coverage depth and 0.01% sequencing error rate

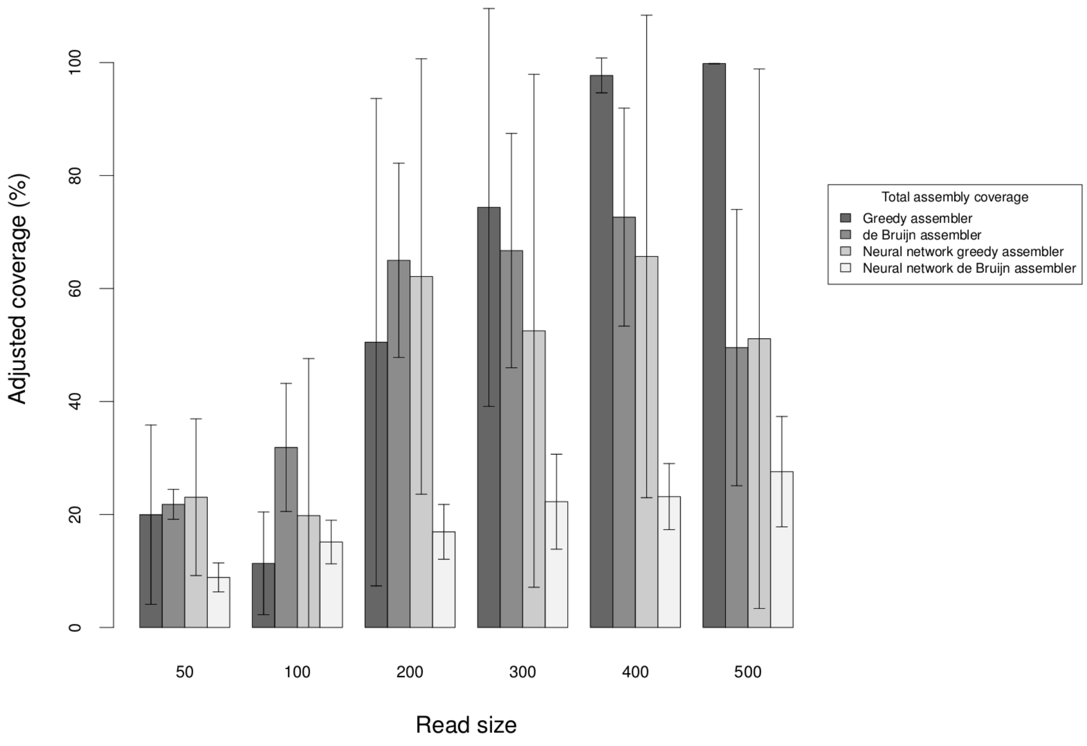


FIGURE B.15: Adjusted coverage at 20 times coverage depth and 0.1% sequencing error rate

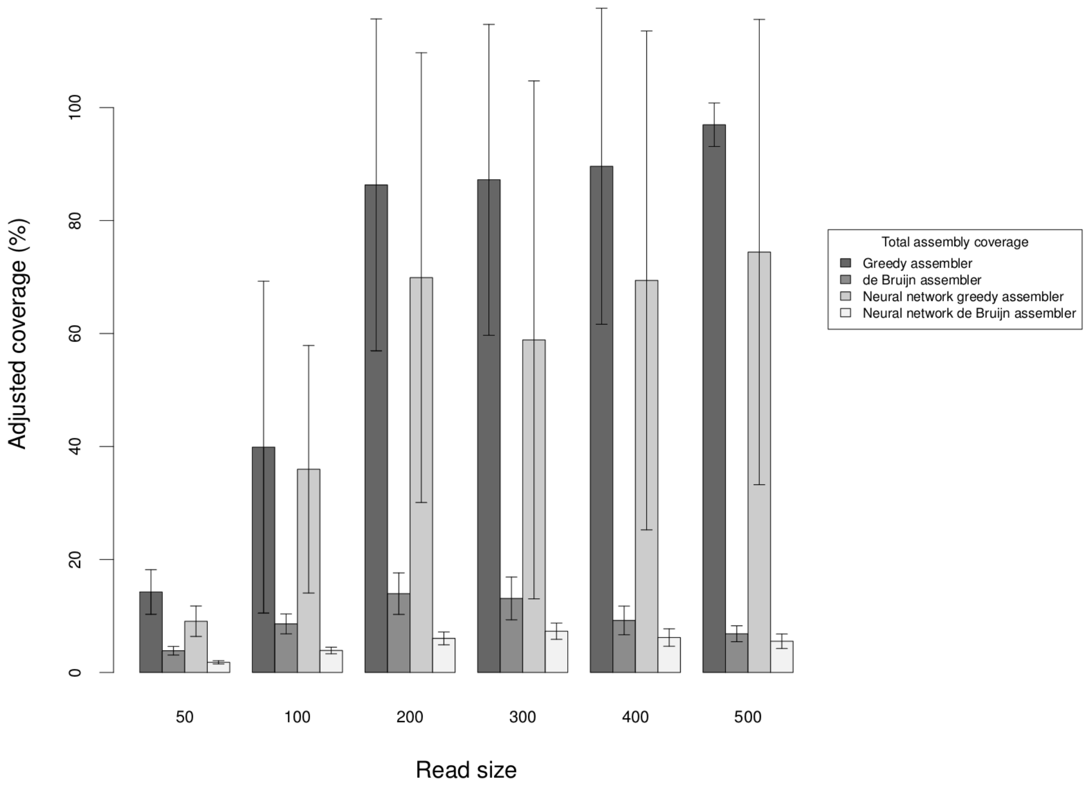


FIGURE B.16: Adjusted coverage at 20 times coverage depth and 1% sequencing error rate

Appendix C

Assembly Errors

The figures contained in this appendix display all supporting results presented and discussed in Chapter 6.

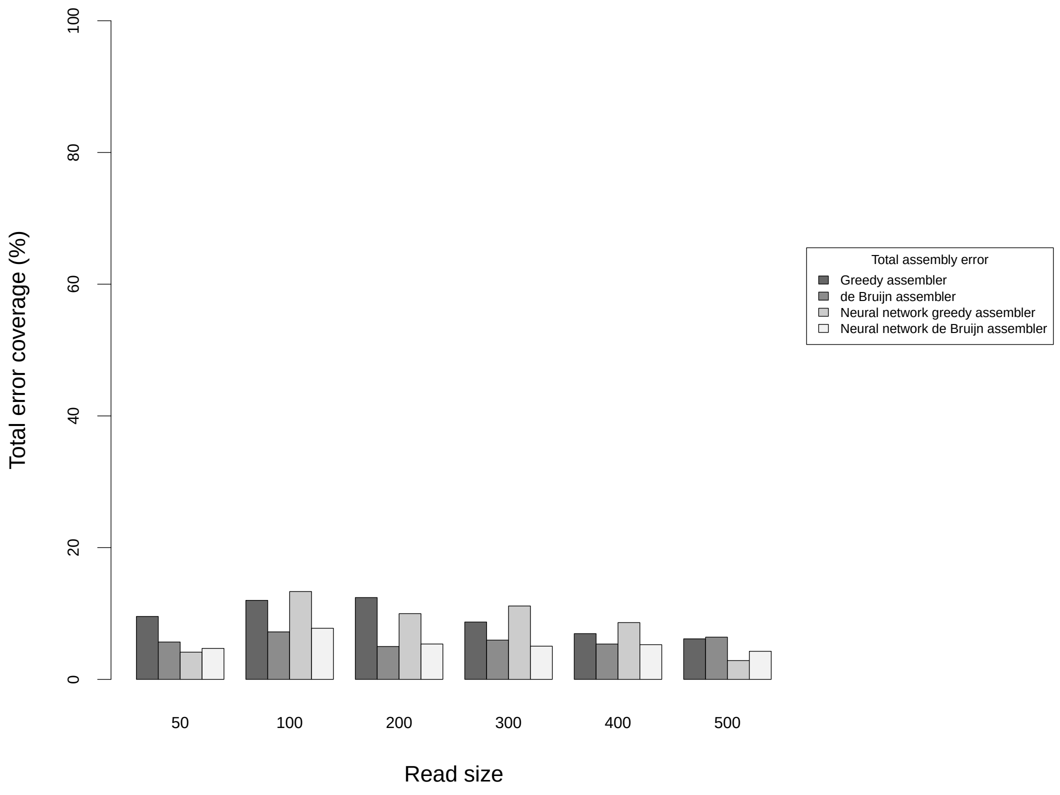


FIGURE C.1: Assembly and sequencing errors at 5 times coverage depth and 0% sequencing error rate

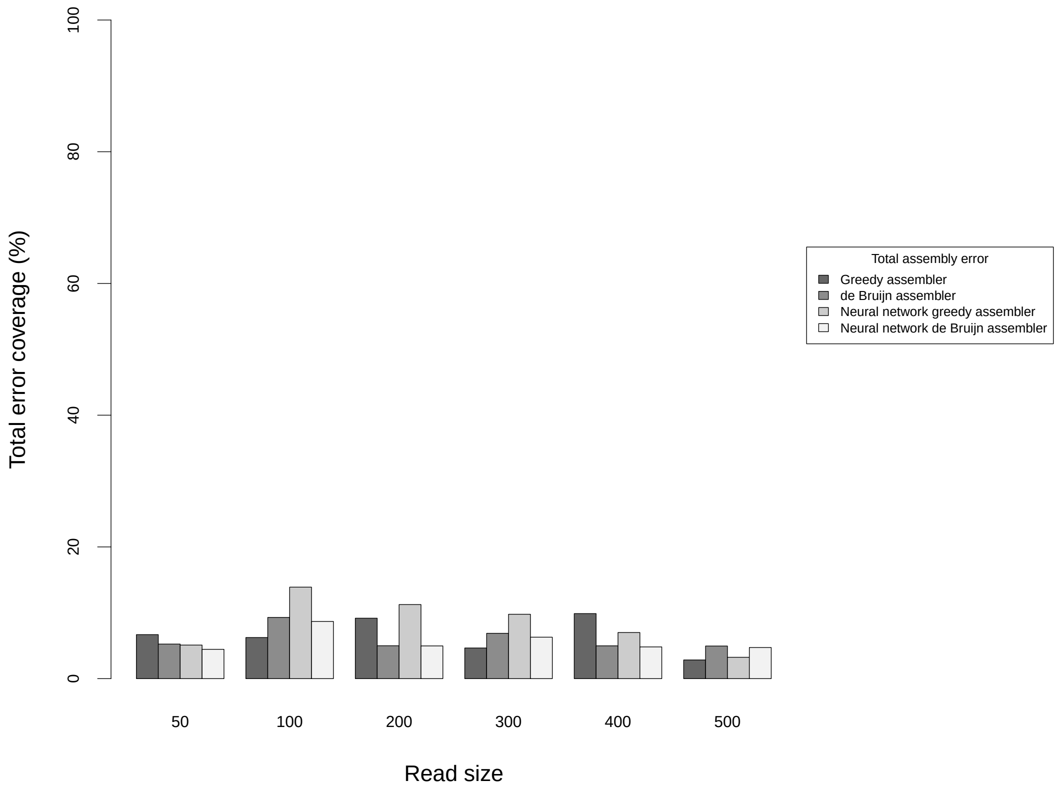


FIGURE C.2: Assembly and sequencing errors at 5 times coverage depth and 0.01% sequencing error rate

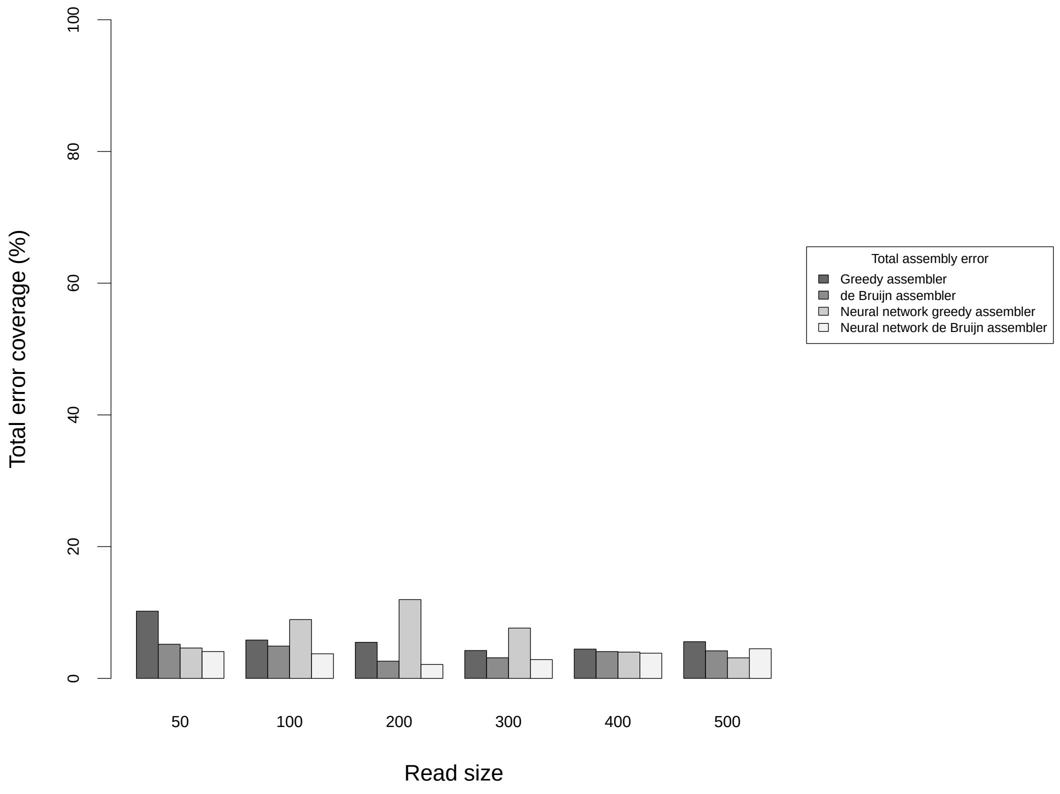


FIGURE C.3: Assembly and sequencing errors at 5 times coverage depth and 0.1% sequencing error rate

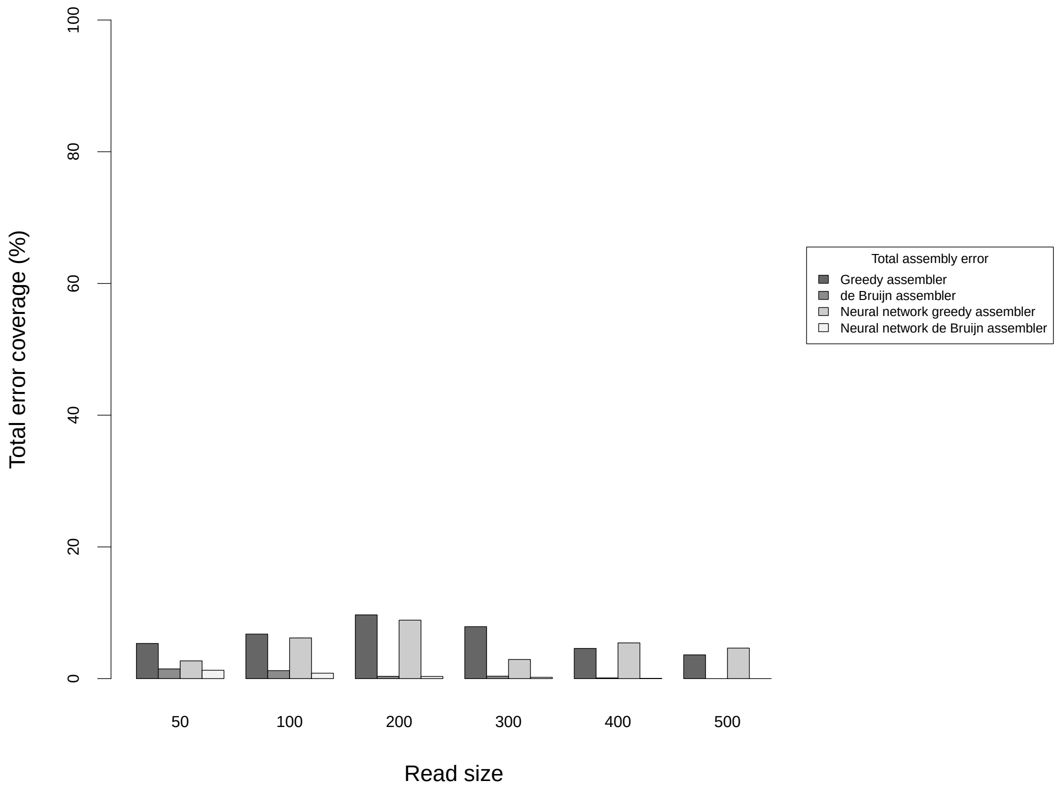


FIGURE C.4: Assembly and sequencing errors at 5 times coverage depth and 1% sequencing error rate

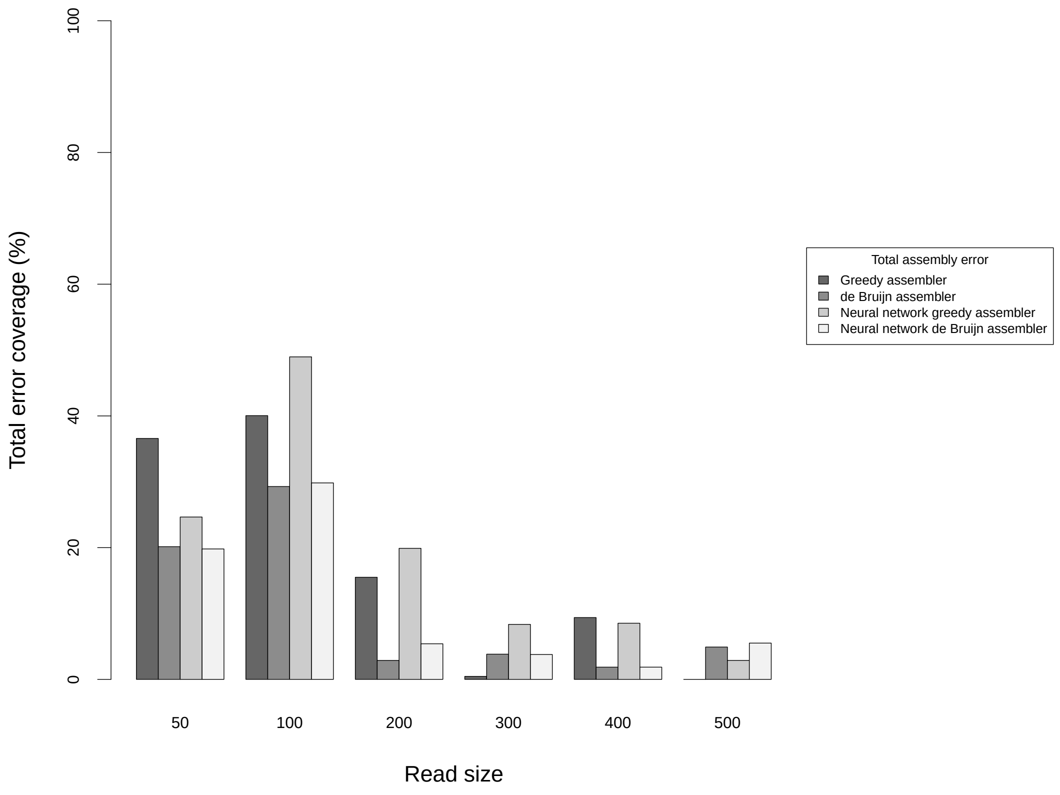


Figure C.5: Assembly and sequencing errors at 10 times coverage depth and 0% sequencing error rate

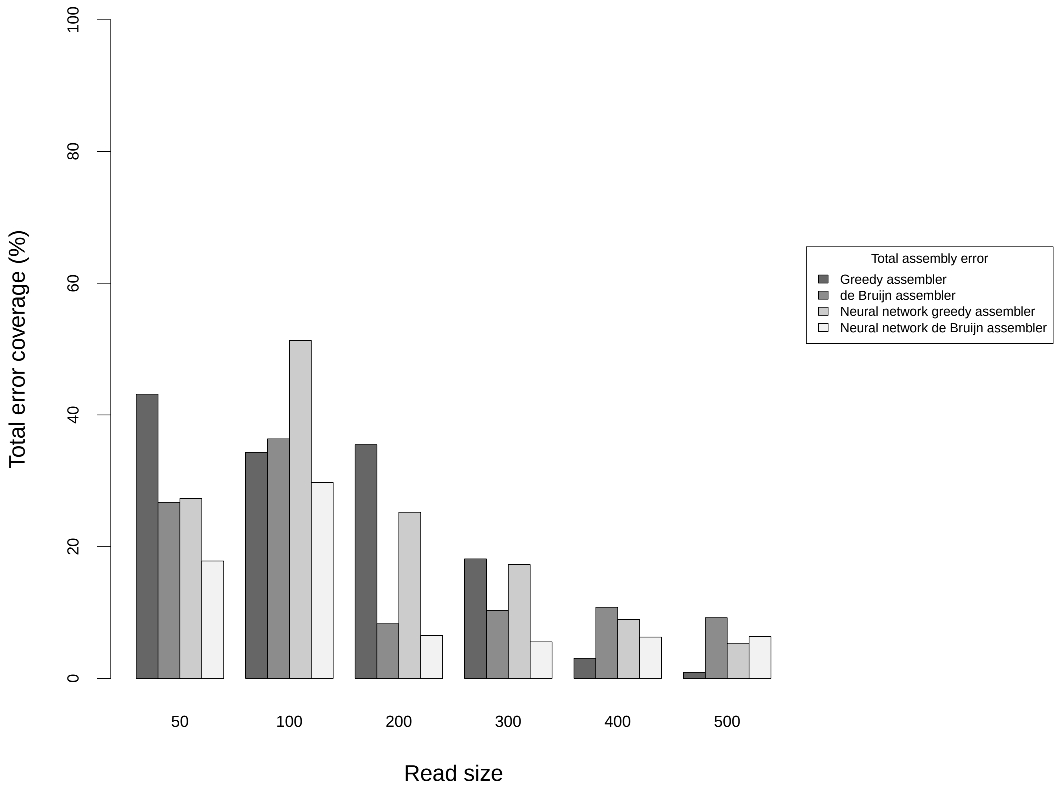


Figure C.6: Assembly and sequencing errors at 10 times coverage depth and 0.01% sequencing error rate

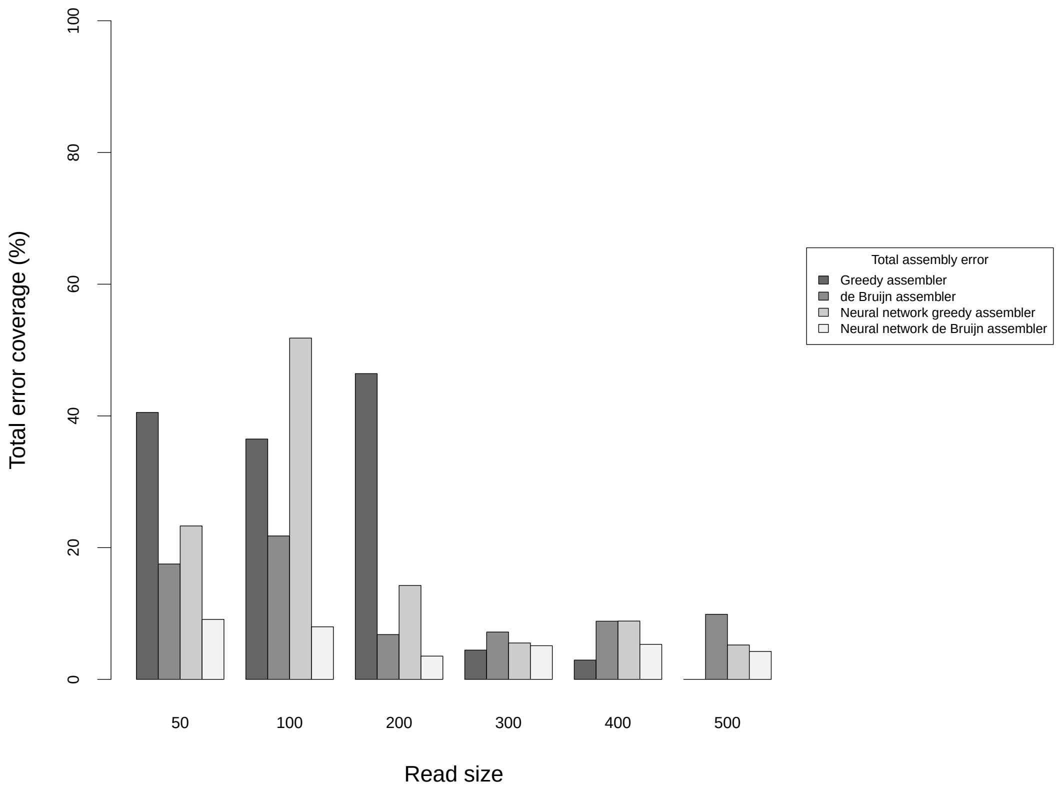


FIGURE C.7: Assembly and sequencing errors at 10 times coverage depth and 0.1% sequencing error rate

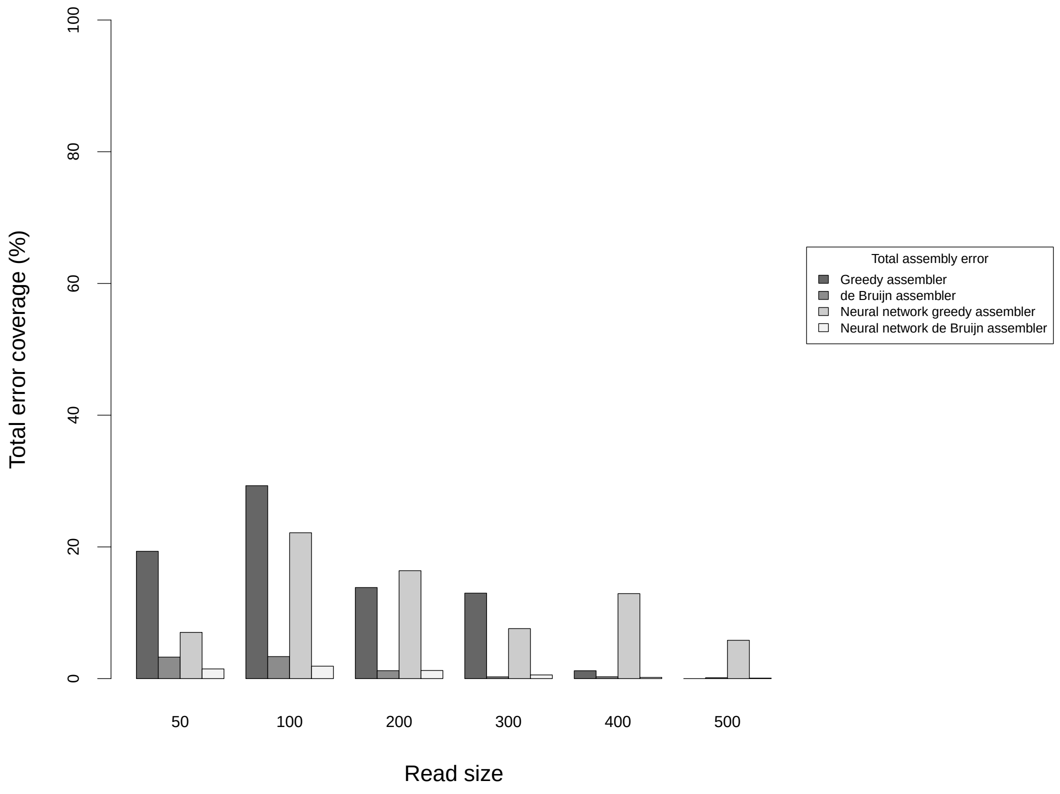


FIGURE C.8: Assembly and sequencing errors at 10 times coverage depth and 1% sequencing error rate

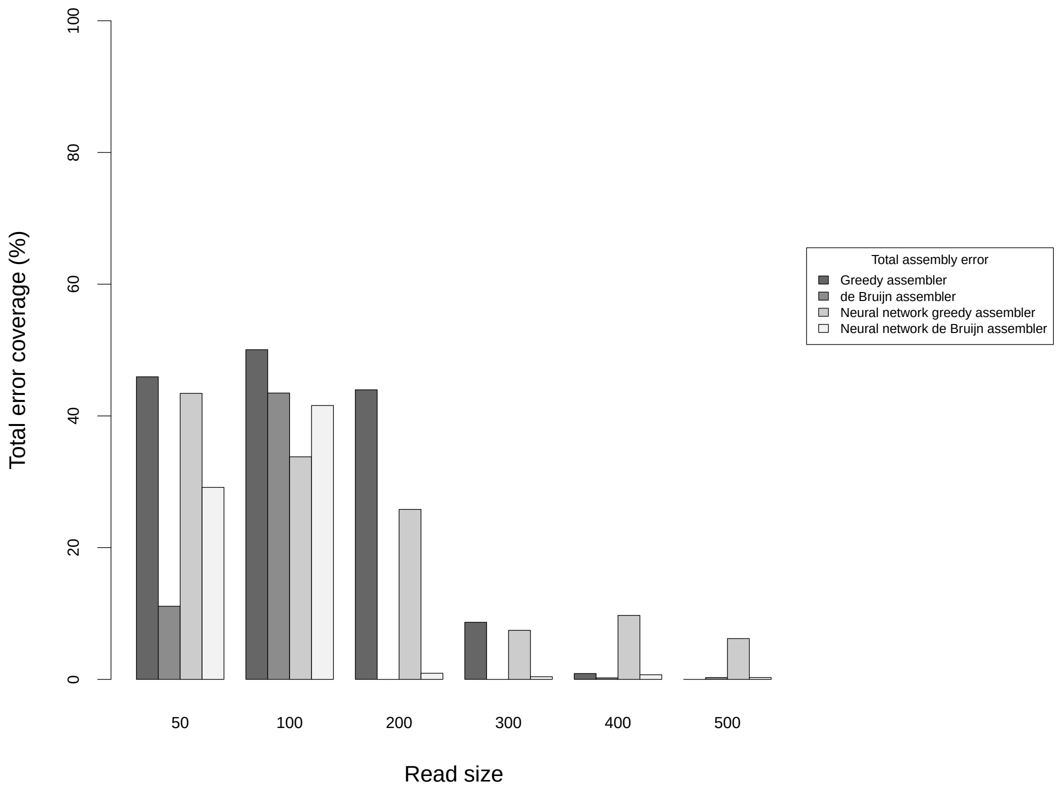


FIGURE C.9: Assembly and sequencing errors at 15 times coverage depth and 0% sequencing error rate

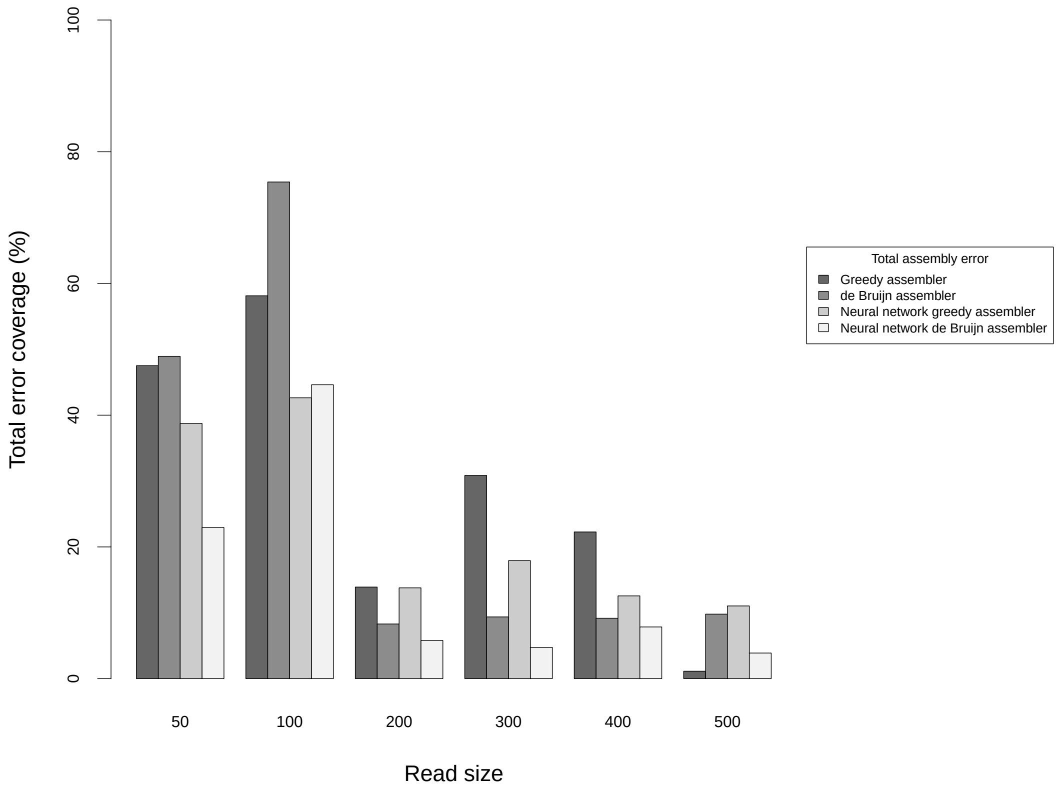


FIGURE C.10: Assembly and sequencing errors at 15 times coverage depth and 0.01% sequencing error rate

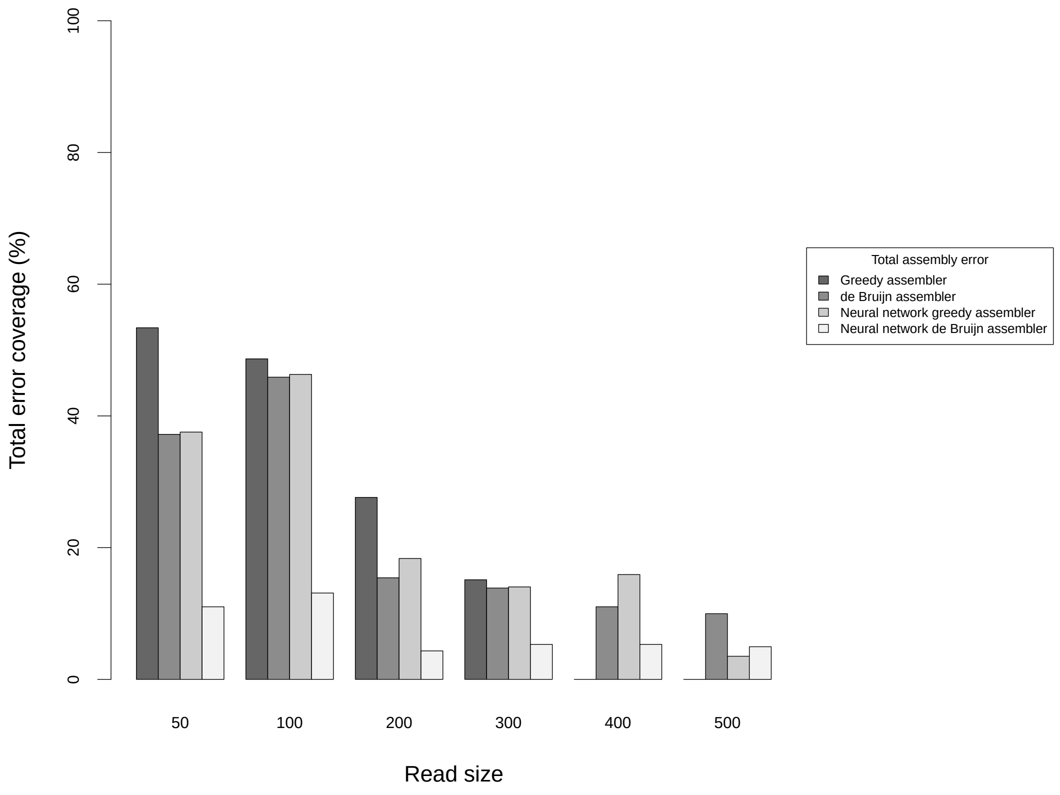


FIGURE C.11: Assembly and sequencing errors at 15 times coverage depth and 0.1% sequencing error rate

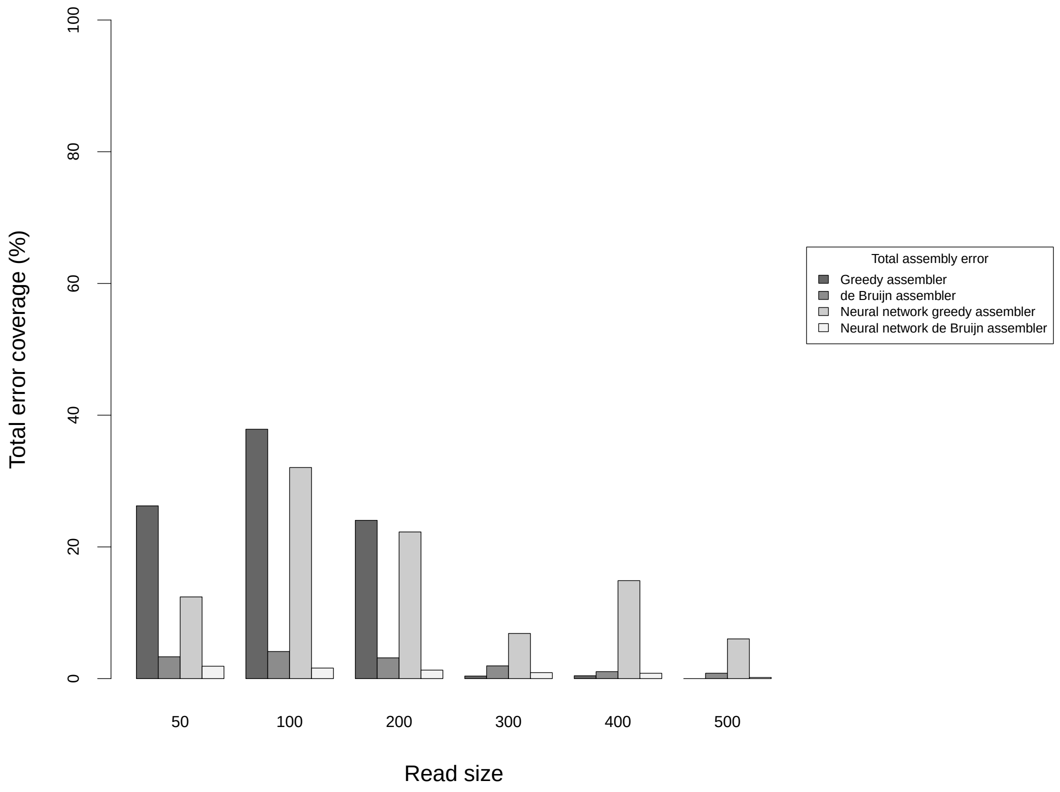


FIGURE C.12: Assembly and sequencing errors at 15 times coverage depth and 1% sequencing error rate

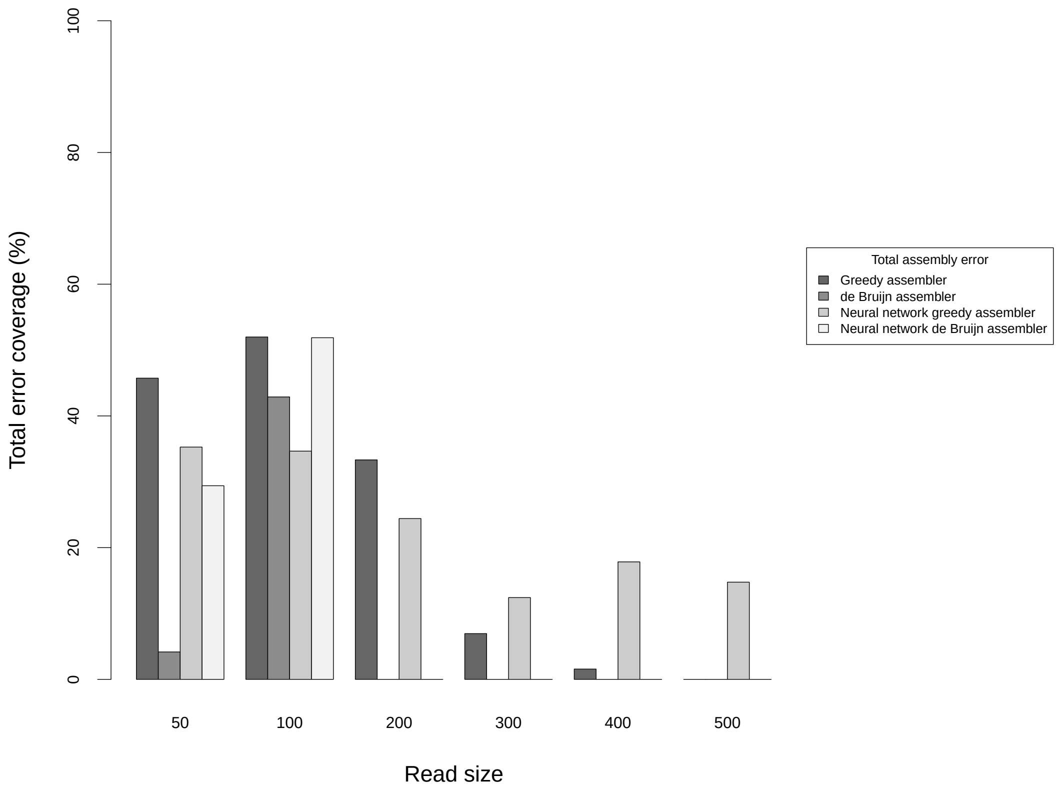


FIGURE C.13: Assembly and sequencing errors at 20 times coverage depth and 0% sequencing error rate

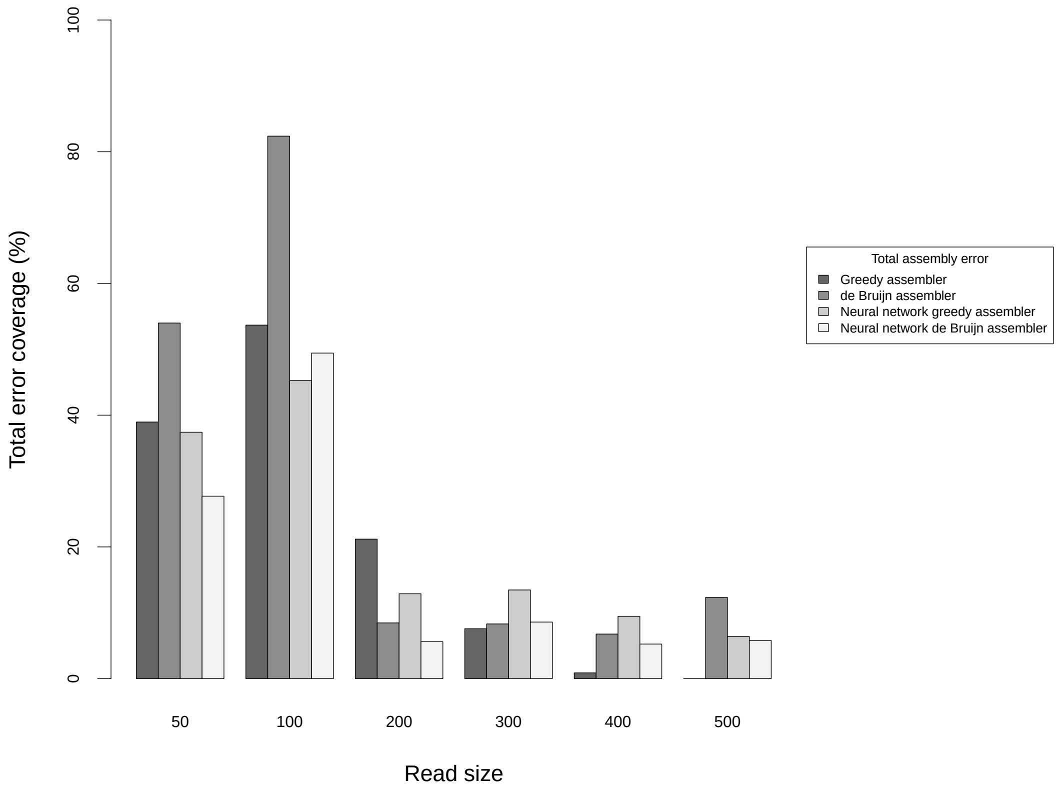


FIGURE C.14: Assembly and sequencing errors at 20 times coverage depth and 0.01% sequencing error rate

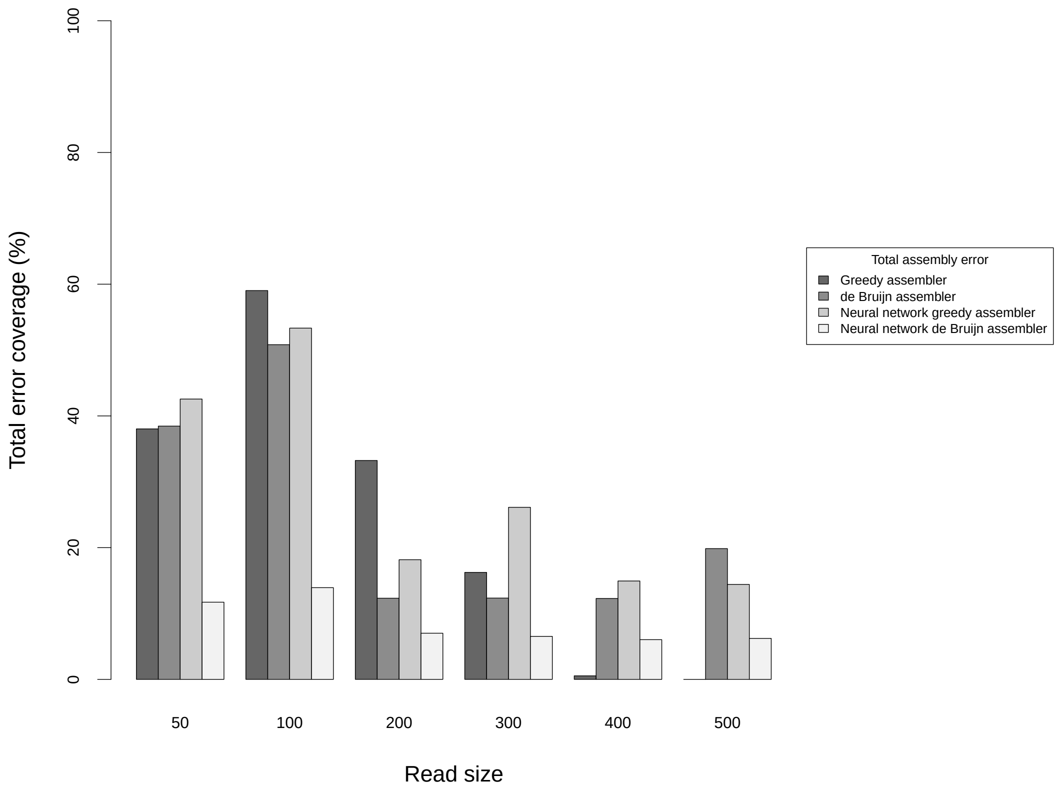


FIGURE C.15: Assembly and sequencing errors at 20 times coverage depth and 0.1% sequencing error rate

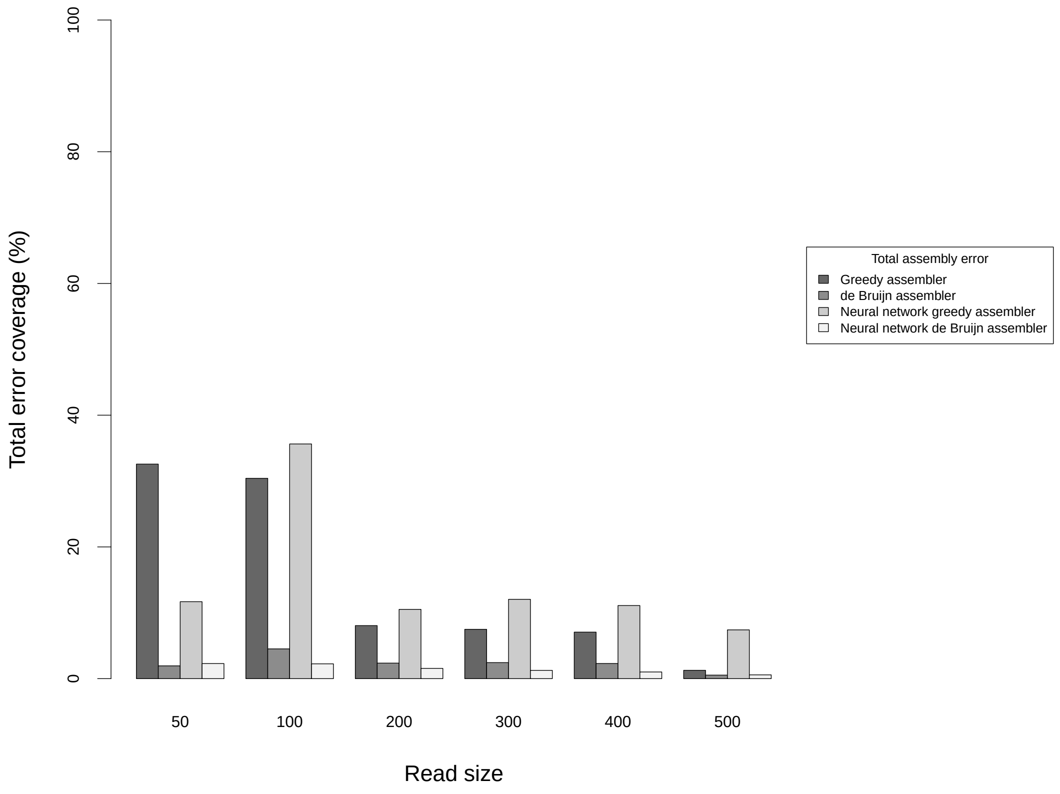


FIGURE C.16: Assembly and sequencing errors at 20 times coverage depth and 1% sequencing error rate

Bibliography

[1] P. A. Pevzner, H. Tang, and M. S. Waterman. An Eularian Path Approach to DNA Fragment Assembly. *PNAS*, 98(17):9748–9753, 2001.

[2] P. Compeau, P. Pevzner, and G. Tesler. How to Apply De Bruijn Graphs to Genome Assembly. *Nat Biotech*, 29(11):987–991, 2011.

[3] N. A. Campbell and J. B. Reece. *Biology*, chapter 16, pages 293–308. Pearson Education International, seventh edition, 2005.

[4] E. W. Myers. Toward Simplifying and Accurately Formulating Fragment Assembly. *Journal of Computational Biology*, 2(2):275–290, 1995.

[5] A. M. Maxam and W. Gilbert. A New Method for Sequencing DNA. *Biotechnology*, (2):99–103, 1992.

[6] F. Sanger, S. Nicklen, and A. R. Coulson. DNA Sequencing With Chain-terminating Inhibitos. *Proc. natn. Acad. Sci. USA*, (74):5463–5467, 1977.

[7] J. Shendure and H. Ji. Next-generation DNA Sequencing. *Nat Biotech*, 26(10): 1135–1145, 2008.

[8] A. S. Motahari, G. Bresler, and D. Tse. Information Theory of DNA Sequencing. *CoRR*, abs/1203.6233, 2013.

[9] X. Huang. A Contig Assembly Program Based on Sensitive Detection of Fragment Overlaps. *Genomics*, 14(1):18 – 25, 1992.

[10] X. Huang and A. Madan. CAP3: A DNA Sequence Assembly Program. *Genome Research*, 9(9):868–877, 1999.

[11] E. Angeleri, B. Apolloni, D. De Falco, and L. Grandi. DNA Fragment Assembly Using Neural Prediction Techniques. *International Journal of Neural Systems*, 9 (6), 1999.

[12] NIH National Human Genome Research Institute. Human genome sequence quality standards, 2012. URL http://www.genome.gov/10000923. Accessed: 28-05-2013.

[13] FlyBase. A Database of Drosophila Genes and Genomes, 2014. URL `http://flybase.org/reports/FBgn0267428.html`. Accessed: 26-11-2014.

[14] International Human Genome Sequencing Consortium. Finishing the Euchromatic Sequence of the Human Genome. *Nature*, 431(7011):931–945, 2004.

[15] M. D. Adams et al. The Genome Sequence of Drosophila melanogaster. *Science*, 287(5461):2185–2195, 2000.

[16] S. Xinwei et al. Shotgun Sequence Assembly and Recent Segmental Duplications Within the Human Genome. *Nature*, 431(7011):927–930, 2004.

[17] R. Arritia, D. Martin, G. Reinert, and M. S. Waterman. Poisson Process Approximaiton for Sequence Repeats, and Sequencing by Hybridization. *Journal of Computational Biology*, 3(3):425–463, 1996.

[18] J. Gallant, D. Maier, and J. A. Storer. On Finding Minimal Length Superstrings. *Journal of Computer and Sustem Sciences*, (20):50–58, 1980.

[19] P. Pevzner. 1-tuple DNA sequencing: computer analysis. *Journal of Biomoleculare Structure*, (7):63–73, 1989.

[20] E. S. Lander and M. S. Waterman. Genomic Mapping by Fingerprinting Random Clones: A Mathematical Analysis. *Genomics*, 2:231–239, 1988.

[21] N. Whiteford et al. An Analysis of the Feasibility of Short Read Sequencing. *Nucleic Acids Res.*, 33(19):e171, 2005.

[22] R. L. Warren, G. G. Sutton, S. J. Jones, and R. A. Holt. Assembling Millions of Short DNA Sequences Using SSAKE. *Bioinformatics*, (23):500–501, 2007.

[23] D. R. Zerbino and E. Birney. Velvet: Algorithms for de Novo Short Read Assembly Using de Buijn Graphs. *Genome Research*, 18(18):821–829, 2008.

[24] C. Dohm, C. Lottaz, T. Borodina, and H. Himmelbauer. SHARCGS, A Fast and Highly Accurate Short-Read Assembly Algorithm for de Novo Genomic Sequencing. *Genome Res.*, (17):1697–1706, 2007.

[25] P. Somboonsak and M. Munlin. A New Edit Distance Method for Finding Similarity in DNA Sequence. *World Academy of Science, Engineering and Technology*, 58, 2011.

[26] V. I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions, and Reversals. *Soviet Physics - Doklady*, 10(8), February 1966.

[27] S. B Needleman and C.D. Wunsch. A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins. *J. Mol. Biol.*, 48: 443–453, 1970.

[28] W. I. Chang and E. L. Lawler. Approximate String Matching In Sublinear Expected Time. In *Foundations of Computer Science, 1990. Proceedings., 31st Annual Symposium on*, volume 1, pages 116–124, Oct 1990.

[29] R. Bellman. The Theory of Dynamic Programming. *Bull. Amer. Math. Soc.*, 60: 503–515, 1954.

[30] T. F. Smith and M. S. Waterman. Identification of Common Molecular Subsequences. *J. Mol. Biol.*, 147:195–197, 1981.

[31] G. G. Sutton, O. White, M. D. Adams, and A. Kerlavage. TIGR Assembler: A New Tool for Assembling Large Shotgun Sequencing Projects. *Genome Science and Technology*, (1):9–19, 1995.

[32] W. R. Jeck, J. A. Reinhardt, D. A. Balatrus, M. T. Hickenbotham, V. Magrini, E. R. Mardix, J. L. Dangl, and C. D. Jones. Extracting Assembly of Short DNA Sequences To Handle Error. *Bioinformatics*, (23):2942–2944, 2007.

[33] R. M. Idury and M. S. Waterman. A New Algorithm for DNA Sequence Assembly. *Journal of Computational Biology*, 2(2):291–306, 1995.

[34] G. Bresler, M. Bresler, and D. Tse. Optimal Assembly for High Throughput Shotgun Sequencing. *CoRR*, abs/1301.0068, 2013.

[35] S. Batzoglou and D. B. Jaffe et al. ARACHNE: A Whole-Genome Shotgun Assembler. *Genome Research*, 12:177–189, 2002.

[36] C. M. Bishop. *Pattern Recognition and Machine Learning*, chapter 7. Springer, 2006.

[37] P. J. Werbos. Backpropagation Through Time: What It Does and How to Do It. *Proceedings of the IEEE*, 78(10), 1990.

[38] C. E. Shannon. A Mathematical Theory of Communication. *BSTJ*, 27:379–423, 1948.

[39] A. Renyi. On Measures of Entropy and Information. In University of California Press, editor, *In Proc. Fourth Berkeley Symposium*, volume 1, pages 547–562, 1960.

[40] M. S. Waterman. *Introduction to Computational Biology: Maps, Sequences and Genomes*. Chapman and Hall/CRC Interdisciplinary Statistics Series. Chapman & Hall/CRC, 1995.

[41] E. Ukkonon. Approximate String-matching With q-grams and Maximal Matches. *Theoretical Computer Science*, 92:191–211, 1992.

[42] P. A. Pevzner. DNA physical mapping and alternating Eulerian cycles in colored graphs. *Algorithmica*, 13(1-2):77–105, 1995.