<table>
<thead>
<tr>
<th><strong>Title</strong></th>
<th>Fuzzy k-mers and their application to comparative genome assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Author(s)</strong></td>
<td>Healy, John</td>
</tr>
<tr>
<td><strong>Publication Date</strong></td>
<td>2013-10-22</td>
</tr>
<tr>
<td><strong>Item record</strong></td>
<td><a href="http://hdl.handle.net/10379/4218">http://hdl.handle.net/10379/4218</a></td>
</tr>
</tbody>
</table>
Fuzzy $k$-mers and their Application to Comparative Genome Assembly

A dissertation submitted to the College of Engineering & Informatics in fulfilment of the degree of Doctor of Philosophy

By

John Healy

Research Supervisor
Dr. Desmond Chambers

October 2013
Contents

Contents................................................................................................................i
List of Figures......................................................................................................... v
Declaration............................................................................................................. vii
Acknowledgements............................................................................................. viii
Abstract...............................................................................................................  x

1 Introduction and Research Hypothesis.......................................................... 1
   1.1 Introduction................................................................................................... 1
   1.2 The Importance of k-mer Matching......................................................... 5
   1.3 Genomes and Jigsaws............................................................................... 7
   1.4 Research Questions...............................................................................  9
   1.5 Structure of Thesis.............................................................................. 10

2 Literature Review............................................................................................ 12
   2.1 K-mer Matching................................................................................... 12
       2.1.1 Seed and Extend.......................................................................... 14
       2.1.2 Spaced Seeds............................................................................... 16
   2.2 De Novo Sequence Assembly............................................................... 18
       2.2.1 Greedy Approaches to Genome Assembly................................. 20
2.2.2 The Overlap-Layout-Consensus Model.............................. 24
2.2.3 The de Bruijn Graph Model......................................... 33
2.3 Comparative Approaches to Genome Assembly......................... 42
2.4 Measures of Assembly Quality........................................... 46
2.5 Chapter Summary.................................................................. 48

3 Research Methodology.......................................................... 50
3.1 Fuzzy Seed Methodology..................................................... 51
  3.1.1 Selection of Seed Model................................................. 51
  3.1.2 Testing and Benchmarking the Fuzzy Approach............... 52
3.2 Comparative Assembly Methodology..................................... 54
  3.2.1 Selection of Assembly Model....................................... 54
  3.2.2 Assembly Testing and Evaluation Methodology................. 55
3.3 Chapter Summary.................................................................. 57

4 Fuzzy k-mers...................................................................... 58
4.1 K-mer Matching with Hash Tables...................................... 59
  4.1.1 Hashing and k-mers.................................................... 59
  4.1.2 Collision Resolution in Hash Maps............................... 61
4.2 The Fuzzy k-mer Model..................................................... 63
4.3 Implementing the Fuzzy k-mer Model.................................. 66
  4.3.1 Seed Design and Implementation.................................. 66
  4.3.2 Configuring Fuzzy k-mers.......................................... 73
  4.3.3 Aligning k-mers and Reifying Matches.......................... 76
4.4 Chapter Summary............................................................. 79

5 Integrated Assembly with Fuzzy k-mers................................. 80
5.1 Guided Assembly Pipeline................................................ 81
5.2 De Bruijn Graph Model..................................................... 83
5.3 Anchoring Reads with Fuzzy $k$-mers

5.3.1 Identifying and Extracting Anchor Sequences

5.3.2 Aligning Sequence Reads to Anchors

5.4 Contig Assembly

5.5 Contig Scaffolding

5.6 Chapter Summary

6 Evaluation of the Fuzzy $k$-mer Model

6.1 Speed Benchmarks

6.1.1 Effect of Hash Size on Running Time

6.1.2 Effect of Reference Genome Size on Running Time

6.1.3 Computational Overhead of KeyComparator

6.1.4 Effect of $\beta$ cut-off Threshold on Running Time

6.1.5 Impact of $k$-mer Size on Running Time

6.2 Sensitivity and Specificity

6.2.1 Effect of $\beta$ cut-off Threshold on Sensitivity & Specificity

6.2.2 Sensitivity & Specificity of Fuzzy Seeds

6.3 Results with Real Data Sets

6.4 Chapter Summary

7 Evaluation of Integrated Comparative Assembly

7.1 Genome Selection and Anchor Extraction

7.2 Comparison of Assembly Results

7.2.1 Assembly Running Time

7.2.2 Completeness of Assembly

7.2.3 Assembly Size and Quality

7.2.4 Comparison with the Alignment-Layout-Consensus Model

7.2.5 Comparison with de novo Assembly Models

7.3 Chapter Summary
8 Summary and Conclusions ......................................................... 163

8.1 Summary of Thesis and Structure ........................................ 163
8.2 Research Contribution ......................................................... 165
8.3 Conclusions and Future Research ......................................... 166

Publications ................................................................. 169

Supplementary Information ...................................................... 170

Bibliography ................................................................. 180
# List of Figures

2.1. Example of an overlap graph .................................................. 25  
2.2. Transitive reduction in an overlap graph ............................... 29  
2.3. Example of a de Bruijn graph .............................................. 34  
2.4. Elimination of singletons in a de Bruijn graph ...................... 38  
4.1. Hash table created from a set of sequence reads .................... 60  
4.2. Hash table created from a set of $k$-mers ............................. 62  
4.3. Example of a fuzzy hash map .............................................. 64  
4.4. UML class diagram of fuzzy $k$-mer model ............................ 67  
4.5. Fuzzy consecutive and spaced seeds ................................... 68  
4.6. Fuzzy seed configuration .................................................. 74  
4.7. Overview of fuzzy $k$-mer alignment in the prototype .............. 76  
4.8. Sequence diagram of the alignment process .......................... 78  
5.1. Comparative assembly pipeline .......................................... 82  
5.2. UML class diagram of graph implementation ....................... 85  
5.3. Read coherence in the prototype ...................................... 87  
5.4. Anchor identification and extraction .................................. 89  
5.5. Queue membership constraints ........................................ 93  
5.6. Node queue priority heuristics ........................................ 95  
5.7. Expanding nodes during graph transversal ........................... 96
5.8. Contig adjacency graph ................................................................. 99
6.1. Alignment time using a random reference sequence .................. 104
6.2. Load time using a random reference sequence ......................... 107
6.3. Alignment time using B. suis ATCC ........................................... 110
6.4. Multiplicity of hash keys generated ........................................... 111
6.5. Running times versus k-mer size .............................................. 115
6.6. Running times versus KeyComparator ................................ ........ 118
6.7. β cut-off threshold versus fuzzy hash key multiplicity ............... 124
6.8. Running time for seeds versus k-mer size ............................... 127
7.1. Assemblies of B. suis 1330 ......................................................... 156
7.2. Assemblies of S. epidermidis ..................................................... 157
7.3. Assemblies of other genomes .................................................... 158
Declaration

I, the Candidate, certify that this thesis is all my own work and that I have not obtained a degree in this University or elsewhere on the basis of any of this work.

John Healy

August 2013
Acknowledgements

I wish to express my profound gratitude to my supervisor, Dr. Desmond Chambers, for his invaluable advice, support and patience over the past four years. I would also like to thank the members of the examination and postgraduate review committees, Prof. Shaun Mahony, Prof. Gerard Lyons, Dr. Michael Madden and Dr. Tom Barry, for their insightful suggestions, comments and discourse.

This work would not have been possible without the help and support of my fellow colleagues at the Galway-Mayo Institute of Technology. Consequently, I am indebted to them for their advice, forbearance and general banter over the course of this PhD. In particular, I would like to thank Gabriel Hicks for his steadfast support and encouragement and Dr. Sean Duignan for his indispensable advice and for proofreading the thesis. I would also like to extend my appreciation to Dr. Owen Foley and my other colleagues at the Dept. of Computing & Mathematics, for their encouragement and good humour and to Denis Murphy for providing the support necessary to embark upon such a lengthy course of study.

I wish to acknowledge a special debt of gratitude to Dr. Malachy Thompson. In addition to being responsible for my foray into the discipline of bioinformatics, Malachy acted as a mentor during my early years at the Galway-Mayo Institute of Technology. Our weekly musings over Crosaire provided a welcome respite from the rigors of study and academic life - *Does the tiger around the hospital show gratitude after tea?* (5, 3).

Finally, I wish to thank my family, especially Emer, Sorcha and Meadbhbh, for their encouragement, support and humour. I have indulged your patience long enough!
for Emer, Sorcha and Meadhbh

Ad Majorem Dei Gloriam

MMXIII
Abstract

The application of k-mer matching to problems in the field of bioinformatics is long established, with k-mer techniques underpinning standard heuristic approaches to sequence alignment and genome assembly. Despite their broad application, conventional k-mer matching techniques lack a native mechanism for accommodating sequence variability, requiring an exact match at pre-defined indices in a k-mer seed.

This thesis presents a fuzzy approach for approximate k-mer matching and investigates its application to sequence alignment and comparative assembly. By combining the speed of hashing with the sensitivity of dynamic programming, fuzzy k-mers unify the two phases of the “seed and extend” strategy into a single operation that executes in average constant time. In contrast with existing methods of k-mer matching, fuzzy k-mers provide native support for string variability.

The fuzzy approach has been implemented in a prototype sequence aligner and genome assembler called Ferox. In addition to their exploitation for sequence alignment, the prototype directly integrates fuzzy k-mer alignments into the contig construction process by combining models of de novo and comparative genome assembly.
Chapter 1

Introduction and Research Hypothesis

“There are known knowns - there are things we know that we know. There are known unknowns. That is to say, there are things that we now know we don't know. But there are also unknown unknowns. There are things we do not know we don't know.” - Dr. Donald Rumsfeld, United States Secretary of Defence, 2002.

1.1 Introduction

The advent of second generation sequencing (SGS) technologies is transforming the disciplines of genetics and comparative biology by making available an enormous volume of biological sequence data [1]. As the mainstay of genome sequencing for three decades, Sanger technologies have been successfully employed to produce genome assemblies of both high contiguity and quality [2], but are too expensive, labour intensive and time consuming for wide scale application [3]. Historically, these constraints significantly limited Sanger sequencing projects, as evidenced by the large
number of genomes completed over the last five years using SGS platforms [4]. The advent of SGS technologies has resulted in the number of sequencing projects doubling every two years [5] and sequencing cost halving every five months [6, 7].

In the most recent study of the status of genomic projects in the GOLD repository, Pagani et al [5] reported 2,907 completed projects, comprised of 1,918 finished genomes and 989 permanent drafts. Despite an acceleration in the number of completed genomes [8], the sheer number of potential candidate species available implies that most will either never be sequenced, or will be sequenced to draft quality only [9]. Indeed Salzberg et al [10] argue that the objective of many de novo assembly projects is primarily to create a draft quality assembly that is the first representative of a species and that, regardless of their quality, these assemblies will remain for many years the only reference sequence available. This view is corroborated by Pagani et al [5] who cite the high cost of finishing sequences as likely to orientate sequencing projects towards the creation permanent draft rather than completed genomes.

Irrespective of the technology platform used in their creation, sequence reads are essentially worthless until they have been either assembled or aligned against existing sequences [11]. Sequence alignment remains the first and most fundamental component of biological sequence manipulation [12] and is essential and integral to genome assembly. Early solutions to the problem of computing an optimal pair-wise alignment of two sequences were provided by Levenshtein [13] and by Needleman and Wunsch [14]. Despite accommodating polymorphisms and indels, both approaches are formulated as solutions to the problem of global sequence alignment, limiting their application to short sequences only.

The publication of the Smith-Waterman algorithm [15] was a landmark in bioinformatics, providing a mechanism for computing an optimal local pairwise alignment of two sequences. Although heavily utilised in sequence alignment and genome assembly, the quadratic space and time complexity of the algorithm places limits on its application [11]. To address these
limitations, the algorithm is typically used in conjunction with divide-and-conquer techniques, with the “seed and extend” strategy preeminent in this respect. The “seed and extend” approach applies fast, exact-matching string comparison techniques to identify regions or “seeds” of similarity between two sequences. An approximate string-matching algorithm is then employed to evaluate and score the sequence between high-scoring seeds and identify candidate alignments. While the seeding phase of the alignment strategy has historically been dominated by index and tree-based techniques [16-19], the unique characteristics of SGS sequence reads has led to the application of compressed structures that exploit the Burrows-Wheeler transform to seed alignments [20-22]. Regardless of the approach used, current seeding techniques are based on exact matching substrings of a sequence and do not provide any native mechanism for approximate string matching.

The emergence of SGS platforms has lead to a re-appraisal of existing algorithms and approaches to sequence alignment [12, 23] and genome assembly [24]. In the context of alignment and assembly, the most salient characteristics of SGS technologies are the short length, high level of coverage depth and the error profile of sequence reads [25]. Until relatively recently, due to their short length, the assembly of SGS reads was restricted to resequencing projects, where a high quality reference genome is already available [4]. The objective of resequencing is to discover single nucleotide polymorphisms (SNPs) and other genetic variations between different strains of the same species, by mapping a set of sequence reads to a finished genome [23, 26]. In contrast, de novo assembly refers to the complete reconstruction of a genome from a set of randomly sampled sequence fragments, without recourse to a reference sequence or comparative genomics [27].

The distinct characteristics of SGS data have also had major implications for the models of assembly designed for use with Sanger sequences [28]. Although SGS platforms yield higher levels of coverage to compensate for short read lengths, Sanger-based assemblers are optimised for longer length paired reads [29]. In addition, overlap detection in assemblers designed for Sanger sequences can be confounded by very short reads [30]. However, it is the vast number of sequence reads produced by SGS platforms that have
posed the most significant challenge to the application of established alignment and assembly techniques to SGS data [26]. In particular, as it is inherently read-centric, the traditional Overlap-Layout-Consensus [31] model of de novo assembly has proven unsuitable for assembling high coverage SGS sequences [30, 32, 33]. Miller et al [34] however, demonstrated that, with significant refactoring and a re-design of the overlap graph, the model can be adapted to accommodate the shorter reads produced by pyrosequencing platforms.

Despite the reported success of using SGS data for the Whole Genome Shotgun (WGS) sequencing and assembly of the Panda [35] and Turkey [36] genomes, doubts have been raised about the quality of SGS assemblies and the viability of de novo assembly based entirely on short reads [37]. In general, WGS assemblies of SGS reads have yielded poorer quality assemblies than those produced by Sanger technologies [26]. A study by Alkan et al [1] found major deficiencies in the ability of SGS assemblers to surmount the challenges imposed by repeats, with up to 16% of segmental duplications absent from the final assembly of the Panda genome. The study concluded that WGS sequencing of large genomes, entirely from short reads, may not be viable and warned against abandoning Sanger techniques until the issue of assembly quality has been addressed. Similar concerns have been raised by Bashir et al [38], who have even questioned the viability of sequencing and assembling prokaryotic genomes solely from short reads. In a recent analysis of the sequence characteristics required for the complete reconstruction of a genome, Bresler et al [39] cite a minimum read length range of 500-3000bp, substantially longer than the read lengths produced by the dominant Illumina sequencing platform. While SGS platforms are capable of generating read volumes à outrance, Li et al [40] have shown that, for most prokaryotes, ultra-high levels of coverage are unwarranted, as high quality draft assemblies can be produced from 400bp reads with 6-10X coverage.

Henson et al [25] argue that the imminent availability of third generation sequencing platforms, capable of generating long sequence reads at high coverage, will force algorithms and models of assembly to undergo further
evolution to adapt to new challenges. In particular, the importance of leveraging the advantages of two or more sequencing platforms has been emphasised by a number of different authors [1, 9, 25, 30]. Adapting such a hybrid sequencing strategy requires a major redesign of existing alignment and assembly models to exploit the advantages of each of the underlying sequencing platforms [34]. Schatz et al [30] suggest using a de Bruijn graph to assemble short reads into contigs and then employing an Overlap-Layout-Consensus assembler to join the contigs into scaffolds. A recent comparison by Li et al [4], of the two main models for genome assembly, concluded that the longer read lengths produced by third generation sequencing platforms will discriminate in favour of the Overlap-Layout-Consensus model over the de Bruijn graph approach. This finding is supported by a recent report from Bashir et al [38], who employed the Overlap-Layout-Consensus model to assemble a V. cholerae genome from hybrid sequences produced by SGS and third generation platforms. The study by Li et al [4] however, reported the superiority in scalability of k-mer centric techniques to manage the highly redundant datasets generated by high-throughput sequencing platforms. Along with Alkan et al [1], Li et al [4] suggest that we are at a turning point in sequence alignment and assembly, with the challenges and opportunities of third generation technologies requiring a radical reappraisal of existing algorithms and techniques.

### 1.2 The Importance of k-mer Matching

The manipulation of biological sequence data using k-mer centric techniques is fundamental to the solution of many problems in bioinformatics [6, 41]. A k-mer is a substring of length k extracted from a sequence [4]. The rationale for the wide application of k-mers arises from the prohibitive quadratic space and time complexity of dynamic programming methods for approximate string matching [19]. Using k-mer matching to align biological sequences is a viable alignment strategy, as overlapping regions of sequences must contain shared k-mer content [34].
In addition to their use in sequence alignment and genome assembly [30], k-mer centric techniques have proven indispensible to the solution of problems as diverse as the identification of repetitive regions in biological sequences [42, 43], motif discovery in proteins [44], sequence error correction [45] and the estimation of genome size [41, 46]. Given the short length of sequence reads produced by SGS platforms, k-mer centric models were quickly identified as ideal for aligning and assembling short sequence reads [11]. In particular, the k-mer centric de Bruijn graph model has been widely adapted for the de novo assembly of genomes using SGS data [29, 32, 47-49].

The short length of k-mers also permits their use as keys in fast, hash-based, data structures, enabling the rapid identification of regions of shared k-mer content between sequences [11]. Although hash-based data structures enable insertion, search and deletion operations to be executed in $O(1)$ average running time [50, 51], the rapidity of these operations is based on an exact match of a k-mer search key against the set of existing k-mers in the data structure. Consequently, while providing a cogent mechanism for rapid alignment, hashing structures also constrain alignment by imposing a requirement for an exact k-mer match.

The rapid search operations provided by hash-based data structures are heavily exploited in the “seed and extend” strategy to identify matching k-mer seeds between a query and subject sequence. Typically using values of $k$ between 10 and 13 bases, early manifestations of the approach employed consecutive k-mer seeds [19, 52, 53], with a requirement for an exact match at each index position in the seed. The seeding strategy was later extended to support spaced seeds [54, 55], requiring an exact match only at predefined indices in a k-mer, described by one or more seed patterns. This latter approach has been widely adapted for aligning SGS sequences reads, with some implementations eschewing the extension phase completely, relying solely on the use of multiple spaced seeds to identify and score alignments [22, 56].

Despite their utility to many problems in bioinformatics, k-mer matching techniques require a trade-off been speed, sensitivity and specificity [11, 12].
Using a small value of $k$ increases the sensitivity of $k$-mer matching, but has the concomitant effect of increasing running time and reducing specificity. When applied to the de Bruijn graph model of genome assembly, smaller values of $k$ give rise to fewer nodes in a $k$-mer graph, but a more fragmented assembly [29]. Increasing the value of $k$, reduces the sensitivity of $k$-mer matching, but improves the running time and specificity of alignment. While using larger values of $k$ facilitates the identification of repetitive sequences in a $k$-mer graph, it also requires either increasing read length or raising the level of coverage to avoid fracturing an assembly in regions where few sequence reads overlap [47].

Despite the ubiquity of $k$-mer matching in bioinformatics, current approaches are stymied by the requirement of an exact match either along all or predefined positions in a $k$-mer and do not provide any native mechanism for approximate $k$-mer matching. As sequencing errors, polymorphisms and indels are characteristic of biological sequences, the requirement for an exact match limits the potential of $k$-mer seeding and alignment techniques.

### 1.3 Genomes and Jigsaws

Genome assembly is one of the most complex and computationally difficult tasks in bioinformatics [24]. A commonly used analogy is that of a jigsaw puzzle [30, 57], where the task of an assembler is comparable to the placing of each jigsaw piece in its correct position. The assembly process is complicated by a number of factors, the most salient of which are sequence read length, sequencing errors, repetitive regions and coverage depth [27].

Read length is analogous to the size of individual jigsaw pieces and is arguably the most significant factor in genome assembly [27, 58, 59]. Sequencing errors can result in bases being erroneously added, removed or identified and may be likened to altering the interlocking parts of a jigsaw. Repetitive regions, of high or low fidelity, are characteristic of the genomes of most organisms [60] and are comparable to blue-sky pieces of a jigsaw puzzle. Coverage depth refers to the amount of oversampling of a genome.
and represents the degree of overlaps between adjacent sequence fragments [61, 62]. Overlaps are the primary mechanism employed by an assembler to establish links between the reads produced by a shotgun-sequencing project. In the jigsaw analogy, coverage refers to the multiplicity of each jigsaw piece. Sampling a genome at low coverage is analogous to discarding pieces of the jigsaw and thus makes the task of assembly more difficult [63].

Since the first sequence assembler was reported by Staden [64] in 1980, genome assembly has remained an active area of research, with no known optimal solution [24]. Current approaches to de novo genome assembly are dominated by two graph-centric formulations of the assembly problem. The Overlap-Layout-Consensus model [31] was the preeminent approach for de novo genome assembly until the advent of SGS sequencing platforms in the middle of the last decade. Although the Overlap-Layout-Consensus approach has been adapted to accommodate the short reads produced by SGS sequencing platforms [34, 65, 66], the k-mer centric de Bruijn graph model has emerged as the de facto standard for assembling short-length shotgun sequencing reads [6]. Both models have merits and shortcomings, but share the generality of identifying and merging uncontested nodes in an assembly graph during the construction of contigs [4].

The increasing availability of sequenced genomes provides a cogent argument for the use of comparative models of genome assembly [67]. The Alignment-Layout-Consensus approach [68] utilises a reference genome to detect overlaps between sequence reads and is the most prominent model of comparative assembly. In the jigsaw analogy, this approach is the equivalent to placing the pieces of a new jigsaw on top of a completed puzzle and then using the similarity of pictures to direct the assembly of the unfinished pieces. This approach requires a high degree of homology between the target and reference genome and is limited by the presence of structural variations, such as inversions and re-arrangements [30].

Regardless of the assembly model or sequencing platform used, the reconstruction of sequence reads into an assembly of high contiguity and quality is predicated on the accurate identification of repetitive regions and
their boundaries [69]. Repetitive regions induce ambiguity into an assembly graph and limit the amount of contiguous sequence that can be reconstructed from sequence reads [70]. Consequently, assemblers typically break contig extension at the boundaries of repeat nodes in an assembly graph and rely on a separate scaffolding phase to resolve repeat-induced conflicts.

1.4 Research Questions

This thesis addresses the question of approximate \(k\)-mer matching and its application to comparative genome assembly. Specifically, two interrelated research questions are addressed in this study. The primary research hypothesis states:

\[
\text{A fuzzy } k\text{-mer approach for approximate string matching can increase the specificity and sensitivity of } k\text{-mer alignments, without adversely impacting running time.}
\]

The methodology and techniques used to determine the veracity of the hypothesis are grounded in established best practice on the use and application of consecutive and spaced seeds. The following related question is also addressed by this thesis:

\[
\text{Can the contiguity of an assembly be significantly increased by exploiting fuzzy } k\text{-mer alignments to extend contig construction through the boundary of repeat nodes in an assembly graph?}
\]

As the question contains elements of both \textit{de novo} and comparative assembly, an integrated or hybrid approach to genome assembly is implied. Consequently, the methodology used to address this question combines the de Bruijn graph model of \textit{de novo} genome assembly with elements of the \textit{Alignment-Layout-Consensus} approach. Given the complexity and multifaceted nature of \(k\)-mer alignment and genome assembly, the methodology applied to the research included the construction of a prototype application, called Ferox, that embodies the key functionality required to address the research questions.
1.5 Structure of Thesis

The remainder of this document is structured as follows:

Chapter two provides a comprehensive appraisal and critique of the corpus of published literature relevant to the research questions. There is an emphasis therefore on the centrality and heavy exploitation of $k$-mers in sequence alignment and genome assembly. In addition to a review and explication of models of *de novo* assembly, alternative approaches to genome assembly, based on comparative genomics, are discussed and evaluated.

Chapter three describes how an experimental computer science methodology was employed to test and evaluate the research questions. The rationale for the design, implementation and benchmarking of the prototype are presented and discussed, with a particular emphasis placed on the incorporation of established best practice at each stage of the research.

Chapter four discusses the mechanics of hashing that underpin current index-based approaches to $k$-mer alignment and presents a novel method for approximate $k$-mer matching. The fuzzy $k$-mer model combines the speed of hashing with the sensitivity of dynamic programming. The fuzzy approach extends existing models of consecutive and spaced seeds to support variability in $k$-mer content. Chapter four also presents and details the design and implementation of the prototype application developed to test and evaluate the fuzzy approach.

Chapter five presents a hybrid approach to genome assembly, implemented in the prototype, that directly integrates fuzzy $k$-mer alignments into the contig assembly process. In addition to the design and implementation of the prototype, chapter five discusses how, using anchor alignments and read threading as path selection heuristics, the integrated approach allows the extension of contiguous sequences through the boundary of repeat nodes in a de Bruijn graph.

Chapter six provides an evaluation and analysis of the fuzzy $k$-mer approach, based on tests with both real biological sequences and synthetic sequence
reads produced by a test framework. The tests and benchmarks presented and discussed address the key features of $k$-mer matching and evaluate those aspects of the fuzzy $k$-mer model that impact on the speed, sensitivity and specificity of alignments. Chapter six also compares and analyses the results obtained using the fuzzy $k$-mer approach with those produced by established $k$-mer based aligners.

Chapter seven presents an evaluation of the results of benchmarks and tests on the assembly module of the Ferox prototype. The tests include benchmarking of the prototype against preeminent implementations of *de novo* and comparative assembly, capable of utilising both SGS and Sanger sequence reads. The analysis of the contiguity and quality of assemblies produced during testing includes both traditional metrics of assembly quality and recently reported, more robust and accurate measures and metrics.

Chapter eight underlines the contribution of the research to $k$-mer matching, genome assembly and the current corpus of literature in these areas. After summarising the key findings of the research, the chapter concludes by identifying how the research findings can be applied to other avenues of possible future research.

A bibliography of cited publications, references and appendices follow chapter eight.
Chapter 2

Literature Review

This chapter presents a review and evaluation of the corpus of published literature relevant to the research hypothesis. The hypothesis itself serves as a general guide to how this chapter is organised and orchestrated. There is therefore an emphasis on $k$-mer centric techniques and their application to sequence alignment and genome assembly. This chapter also presents a review and critique of consecutive and spaced seed models and discusses the ubiquity and centrality of “seed and extend” heuristics to both sequence alignment and genome assembly. The preeminent models of $de$ $novo$ and comparative assembly are reviewed and discussed, along with an appraisal and evaluation of current approaches for measuring assembly contiguity and quality. The discussion of $de$ $novo$ genome assembly emphasises the centrality of $k$-mer matching in both the de Bruijn graph and Overlap-Layout-Consensus models. Although there exists an expansive quantity of published material relating to the general problem and paradigms of $de$ $novo$ genome assembly, this is in contrast with a relative paucity of published research regarding comparative and orthologous assembly models. This is remarkable, given the rapidly increasing availability of published genomes of complete and draft quality.
2.1 K-mer Matching

The use of k-mers is a leitmotif in bioinformatics, with k-mer centric techniques used, *inter alia*, to seed sequence alignments [18, 19, 52, 55, 56, 71-76], screen sequence read errors [33, 34, 45, 47, 77, 78], identify repetitive DNA sequences [41-43] and model genome assembly [4]. A k-mer is a sequence of k consecutive bases, with k-mer s adjacent to k-mer t (s→t) if there is a (k+1)-mer in a sequence whose first k bases are s and whose last k bases are t [47]. It follows that s and t overlap by k−1 bases. The set of k-mers for a read, called the k-spectrum, can be extracted by its decomposition into a tiling of substrings of size k. For a read r of length L, the k-spectrum (\(r^k\)) = \{r [i : j + k - 1] | 0 ≤ i < L - k + 1\}, where r [i : j] denotes the substring from position i to j in r [45]. K-mers can thus represent a tiling of fixed-length substrings of a read or a complete genome, with algorithmic space and time complexity increasing exponentially with k [79].

Despite their importance as models of genome assembly [4], the use of k-mer centric techniques in bioinformatics is motivated primarily by the impracticability of using dynamic programming algorithms to align either long or large numbers of DNA sequences [12]. Irrespective of the sequencing platform used, sequence reads are essentially worthless until they have been synthesised and orchestrated into manageable and understandable pieces, by aligning the sequence reads against a set of existing sequences or a finished genome [11]. Although the problem of finding an optimal pair-wise alignment of two sequences was solved by Smith and Waterman [15], their algorithm has a space and time complexity of \(O(n^2)\), rendering its application unfeasible for sequences \(≥ 4\)Mbp [16]. This limitation also applies more generally to dynamic programming algorithms and is accentuated in the context of genome assembly, where the number of alignments required squares in proportion to the number of shotgun sequence reads [77]. Decomposing a sequence into a tiling of k-mers is a viable alternative alignment mechanism, as reads with high sequence similarity must share k-mers in their overlapping regions [27]. Moreover, the short length of k-mer sequences facilitates their exploitation in efficient hash-based data structures, vastly reducing the computational cost of alignment and assembly [19].
As k-mers represent k consecutive characters in a sequence, their application in sequence alignment and assembly is limited to exact string-matching techniques [45]. Although methods for exact string matching, based on suffix trees [16, 80, 81], prefix trees [82, 83], arrays [84] and, more recently, Burrows-Wheeler indexing [22, 85-87] have been developed, the manipulation of k-mers using hash-based techniques remains central to sequence alignment and genome assembly [23, 27]. Hash-based dictionary structures, such as hash tables and hash maps, exploit the $O(1)$ running time required to access an array element at a known array index [88]. Using these structures enables the fast detection of k-mer content and vastly reduces the computational cost of alignment and assembly [27].

In the absence of any mechanism for approximate string matching, k-mer alignment techniques require a compromise between speed and sensitivity [11, 12, 19, 55]. This compromise is controlled by the k-mer size, with smaller sizes of k increasing the possibility of detecting a local alignment, but also increasing the number of spurious matches [12]. If the size of k is too large, an alignment of two sequences will miss high-scoring matches that do not have k consecutive characters [79]. Furthermore, while larger values of k decrease access time to hash-based dictionary structures, smaller k-mer sizes increase the number of hash collisions, resulting in a subsequent escalation in the time complexity of search operations [51]. In practice, for a genome $G$, the value of k is typically selected so that $4^k > |G|$, reducing the expected number of occurrences of any k-mer in $G$ to one [45]. As the complexity of a k-mer increases exponentially with k, selecting values of k, where $10 \leq k \leq 16$, enables the full k-spectrum of a prokaryotic genome to be loaded into 4Gb of RAM, regardless of the number of sequence reads.

### 2.1.1 Seed and Extend

To circumvent the loss of sensitivity arising from the exactness constraint of k-mer matching [55], hashing techniques can be augmented with dynamic programming algorithms to improve sequence alignment. Known as “seed and extend”, this approach uses fast exact-matching data structures to identify
regions of sequence similarity, which can then be extended using approximate string matching algorithms [23]. An early implementation of the “seed and extend” strategy was reported by Pearson and Lipman [19] to align amino acid sequences. Their FASTA aligner used a hash table to rapidly identify candidate seeds, which were then extended using dynamic programming. A similar approach was employed by Altschul et al [52] in the BLAST family of sequence alignment tools. BLAST is an index-based local aligner and was, until recently, the de facto standard software for computing a local pair-wise alignment of two sequences. BLAST is capable of rapidly aligning pairs of nucleotide and protein sequences and can also compare translated nucleotide sequences against a database of proteins. Building on the robust statistical framework developed by Altschul [89], BLAST uses the “seed and extend” strategy to rapidly identify high scoring pairs (HSPs) of alignments. The seeding phase uses a hash table to identify pairs of exact $k$-mer matches between two sequences. The seeds, also known as words or word matches, are extended left and right until the alignment similarity, computed using the Smith-Waterman [15] algorithm, falls below a given cut-off threshold. Resulting local homologies greater than the threshold are then reported.

The seeding phase of BLAST was later refined by Altschul et al [53] to require a two-hit seed to initialise an extension. The hash table constructed from the query sequence uses $k$-mers, by default 11-mers, as a hash key and can search against a database of subject sequences in linear time. Increasing $k$-mer size increases the size of the hash key and has the effect of reducing the search time at the expense of sensitivity. While the memory complexity of BLAST is thus a function of query size, the running time is dependent on the number of seeds that initiate the extension phase. A similar alignment technique was developed by Kent [18] and used in the BLAT aligner. Seeding in BLAT is achieved by decomposing a query sequence into a tiling of $k$-mers that are compared against a hash table of non-overlapping $k$-mers from a subject sequence. While the approach described by Kent permits “near perfect” $k$-mer matches, with an edit distance of one, this approximation is achieved by repeatedly searching a hash table for each $k$-mer in the query, with a consequently more punitive time complexity of $O(k\Sigma)$ [18]. Moreover,
Rasmussen et al [90] note that indexing non-overlapping $k$-mers only reduces memory complexity at the cost of reduced sensitivity.

Brudno and Morgenstern [72] reported a more refined “seed and extend” implementation that employs a variant of the Aho-Corasick [91] algorithm to locate inexact seeds in a keyword trie created from a reference sequence. Their CHAOS aligner uses heuristics to rapidly identify chains of $k$-mers that can be used as anchor points for an alignment. More sensitive dynamic programming techniques are then used to align seeded regions and join together links in the chain. Similar chaining and anchoring techniques, based on suffix trees, were developed by both Delcher et al [16] and Bray et al [92]. Although the trie data structure used in CHAOS permits a degree of $k$-mer mismatch, this is implemented by extending the prefix of the search $k$-mer and has no implicit support for gaps, indels or polymorphisms. Furthermore, while chaining and anchoring techniques can be used to simplify a global alignment [12] and reduce the number of $k$-mers that must be extended [93], using anchoring indices to chain seeds assumes a high degree of synteny between the query and subject genomes [68].

### 2.1.2 Spaced Seeds

Despite their success and longevity, many of the established “seed and extend” implementations were developed when protein sequences comprised the majority of data in biological sequence repositories [72]. Even with the larger alphabet of protein symbols, the requirement for an exact match of $k$ continuous characters to seed an alignment limits the sensitivity of the “seed and extend” approach [19]. This limitation is accentuated by SGS reads, where high levels of coverage and short read length require more rapid search speed and even greater sensitivity [74, 94]. In a seminal work, Ma et al [55] proposed that alignment sensitivity could be significantly increased by seeding regions of high similarity with non-consecutive $k$ matches, called patterns or spaced seeds. Spaced seeds are binary strings, the weight of which is determined by the number of 1’s in the seed. Spaced seeds are analogous
to masks, with 1’s corresponding to a required match and 0’s indicating a “don’t care” position [95].

Noting the low specificity exhibited by the use of a single spaced seed, Li et al [54] extended the model to support multiple spaced seeds, allowing greater sensitivity and specificity during the seeding phase of alignment. The effectiveness of the approach was corroborated in a study by Sun and Buhler [96] who showed that the use of a small number of spaced seeds leads to significantly more sensitivity than an optimised single spaced seed. Rasmussen et al [90] later described a method for increasing the sensitivity of spaced seeds by filtering. More recently, Ilie [95] has shown how a hill climbing method based on an overlap complexity heuristic can significantly reduce the overall running time of a multiple spaced seed alignment.

In recent times, the pioneering work of Ma et al [55] has lead to a proliferation of alignment tools based on the exploitation of multiple spaced seeds [73-76, 94, 97-101]. Consistent with the established practice of using consecutive seeds in conjunction with hash-based data structures, spaced seed aligners typically build an index of seed alignments from either a set of sequence reads [56, 73, 75, 76, 94] or a reference genome [74, 99, 100]. A variation of the spaced seed approach was reported by Strömberg [102] and employed in the Mosaik aligner and assembler. Mosaik uses $n$ hash positions per spaced seed and a clustering algorithm to seed a full Smith-Waterman alignment. This approach is similar to the established “seed and extend” technique employed by consecutive $k$-mer seed aligners such as BLAST [52] and BLAT [18].

As homologous sequences exhibit diverse levels of similarity, the pattern used in a spaced seed can have a significant impact on both sensitivity and specificity. Consistent with consecutive seeds, spaced seed design requires a trade-off between sensitivity and specificity, with both parameters controlled by the seed weight. A study by Choi et al [103, 104] analysed spaced seeds over a variety of different levels of homology and identified an optimal set of spaced seeds for a given seed weight and hit probability. A later study by Zhang [105] proved the superiority of spaced seeds over consecutive $k$-mers,
provided the seed size is not too large. Mak and Benson [106] showed that the most efficient spaced seeds have a time complexity of $O(LPwS)$, where $L$ is the sequence length, $P$ is the number of patterns per seed, $w$ is the seed length and $S$ is the number of seeds.

Pol and Kahveci [79] suggest that the requirement for an exact match at each 1’s position in a seed results in some high scoring matches remaining undetected. Although they provide higher sensitivity without a loss in specificity, the patterns applied in spaced seeds assume only polymorphic mutations between homologous sequences or that indels are widely spaced. The requirement for an exact match at 1’s positions was partly addressed by Noé and Kucherov [107] who, noting that transition mutations occur more frequently than transversions, proposed the use of transition-constrained seeds to accommodate mismatches. Implemented in their YASS aligner, transition-constrained seeds employ a three letter alphabet, where “#” and “-” symbols indicate “must match” and “don't care” positions and an “@” symbol allows for a transition mutation. A more robust solution was reported by Mak et al [108], who extended the spaced seed model to include support for indels. The indel seeds they proposed are an extension of the spaced seed model that permit some separating “don’t care” positions to be of variable size, enabling some accommodation of insertions and deletions in sequences.

Notwithstanding their superiority over consecutive seeds, Misra et al [100] argue that most of the spaced seed aligners designed for SGS reads do not perform well for read lengths $\geq 200\text{bp}$, and permit very few mismatches, typically $\leq 2$ nucleotides. Li and Homer [23] posit that, like exact $k$-mer matching techniques, conventional spaced seeds do not permit gaps, limiting the technique in the presence of transpositions, polymorphism and indels. This observation is corroborated by Batzoglou [12], who also notes that, while pattern–based seeding and indexing is superior to seeding with exact $k$-mers, their utility decreases when the homology to be detected is very short.
2.2 De Novo Sequence Assembly

*De novo* assembly refers to the reconstruction of a DNA sequence from a collection of randomly sampled sequence fragments, without consultation to previously resolved sequences such as genomes, transcripts and proteins [27, 57]. Despite the rapid advancement in sequencing technologies over the last decade, computationally, *de novo* assembly is NP-hard, with no known efficient solution [24]. Although SGS technologies are capable of producing very large numbers of sequence reads at low cost [109], they do not address the difficult problem of generating a correct assembly from a set of randomly generated sequence reads [37]. An assembly is inherently hierarchical [27], mapping sequence reads to a tentative reconstruction of a target genome, by grouping sequence reads into contigs and joining contigs into scaffolds. The output of an assembler is a collection of contigs, where each contig represents a tiling path of its underlying constituent reads [110].

Irrespective of the sequencing platform used, the successful *de novo* assembly of sequence reads is predicated on the unambiguous identification of repetitive regions and repeat boundaries [69]. Repetitive regions confound the assembly of contiguous sequences and limit the amount of sequence that can be effectively reconstructed [70]. This is particularly relevant for SGS sequence data, as a larger proportion of a genome is repetitive at short lengths, forcing an assembler to make more contig breaks at repeat boundaries [30]. Haiminen *et al* [111] argue that short read assemblies of repeat-rich genomes are highly fragmented, causing repeat expansions, collapses and misjoins in contigs. In addition, genomic regions that share perfect repeats are difficult to distinguish, especially if a repetitive region is longer than the read length [27]. This problem is exacerbated by the short length of SGS sequence reads [112]. Indeed, Vezzi *et al* [59] argue that short reads have made the assembly problem harder, due to the complexity involved in resolving long repeats with short sequences. Moreover, SGS platforms have sequence-dependent coverage biases resulting in non-uniform coverage rates [82]. Coupled with the higher number of sequence errors associated with SGS data [27], uneven high coverage can confound the classification of a region as being repetitive or not [32].
Although Whiteford et al [70] demonstrated that the *de novo* sequencing of the majority of a bacterial genome was possible with read lengths of 20-30bp, consistent with the earlier opinion of Myers *et al* [113], both Chaisson *et al* [58] and Medvedev and Brudno [114] reported the viability of the Whole Genome Shotgun (WGS) approach with short reads, provided the reads were paired. Even with paired reads however, the ability of an assembler to resolve a repeat-induced conflict, by circumnavigating or spanning a repetitive region, remains constrained by read length [4]. The study by Whiteford *et al* [70] also showed that the percentage of unique reads has a sigmoidal dependence on read length, further illustrating the difficulty of a *de novo* assembly of repeat-rich genomes with SGS data. Kingsford *et al* [115] analysed the complete genomes of 408 prokaryotes and calculated the theoretical optimal assemblies possible from simulated error-free reads. Their study demonstrated that, while longer reads yield larger contigs, the largest increase in contig size was observed when read length was increased from 25 to 35 bases, with diminishing returns thereafter.

The remainder of this section reviews and appraises the dominant models of *de novo* genome assembly and emphasises the centrality of *k*-mer matching to the different approaches.

### 2.2.1 Greedy Approaches to Genome Assembly

The assembly of shotgun sequence reads was initially formulated as an approximation of computing the *Shortest Common Superstring* (SCS) for a set of sequence fragments [57, 116, 117]. For a given set $S$ of sequences $\{s_1, s_2, \ldots, s_n\}$, the SCS problem is to find the shortest string $T$ such that every $s_n$ is a substring of $T$ [117, 118]. In the absence of sequencing errors and repetitive regions, the SCS formulation of the sequence assembly problem is valid [31, 116]. Unfortunately, as sequencing errors and repetitive sub-sequences are characteristic of sequence reads, modification of the formulation is required. In addition, the SCS problem is NP-hard [119, 120], but can be solved in polynomial time by heuristically informed greedy approximations [57].
Motivated by parsimony [119], greedy approximation algorithms typically construct solutions incrementally, by iteratively expanding a partially constructed solution until a complete solution to a problem is reached [121]. Depending on the shape and size of the state-space, greedy algorithms work quite well and can find a good solution very quickly [122]. Greedy assembly algorithms are implicitly graph-based, but significantly simplify a graph by considering only high-scoring edges [27].

Greedy assemblers construct an assembly of contigs incrementally, by computing all possible overlaps between read fragments and assigning a score to each putative match [31, 57, 116, 123]. The highest scoring sequences are then recursively merged and re-added to a pool of candidate sequences. The process iterates until there are no more sequences left to merge. After each merge, the overlapping substring is typically heuristically corrected by applying clone length and paired read distance constraints.

Without mechanisms to identify spurious overlaps and repeats, greedy strategies are particularly prone to collapsing an assembly scaffold into a single, large erroneous contig [82, 124, 125]. Notwithstanding this shortcoming, early genome assemblers successfully applied greedy approximation techniques to sequence reconstruction [123, 126-130]. The Phrap assembler [126] employs a greedy algorithm to iteratively merge sequence reads. Using a “seed and extend” approach, similar to many other alignment and assembly techniques [11, 12, 23], Phrap identifies all 14-mer words from a set of input sequence fragments, and uses these as seeds to detect overlaps between reads. A variant of the Smith-Waterman algorithm [15], SWAT, is employed to extend and score matching $k$-mers. Base-calling quality values are then utilised to compute log likelihood ratio (LLR) scores and identify spurious overlaps. The correct ordering and orientation of aligned reads is constructed by sorting all matching pairs in order of decreasing LLR score. A greedy algorithm then progressively merges high scoring pairs until no more merges are possible.

A similar approach was employed by Huang and Madan [127] in the design of the CAP3 assembler, which utilises quality scores to compute overlaps and
identify chimeric reads. CAP3 also employs a “seed and extend” strategy and uses paired reads to apply orientation constraints when joining contigs. Despite its jejune approach to identifying repetitive regions, the Phrap assembler was originally designed to be used with a hierarchical sequencing strategy [131]. This “divide and conquer” assembly strategy effectively reduces the genome assembly problem into a BAC assembly problem and eliminates the possibility of distantly dispersed repetitive regions convoluting a final assembly.

A more sophisticated greedy assembly approach was used by Fleischmann et al [61] for the WGS sequencing of the 1.8Mbp *Haemophilus influenza* and by Fraser et al [132] to assemble the 0.58Mbp genome of *M. genitalium*. The TIGR assembler, developed by Sutton et al [130] compares every sequence fragment to every other fragment to find potential pairwise overlaps, requiring $O(n^2)$ comparisons. A $k$-mer search and a variant of the Smith-Waterman algorithm are used to identify and score overlaps. The TIGR assembler uses the median distribution of overlaps as a mechanism for identifying repeats. The PCAP assembler, developed by Huang et al [128], extends this idea by imposing the requirement of a two-hit seed for overlaps, analogous to the high-scoring pairs used by BLAST [52, 53]. Imposing such a constraint enables the detection of repetitive regions based on deep coverage by longer approximate matches instead of shorter exact matches. After identifying and labelling repetitive sequences, the TIGR assembler uses an overlap list to greedily merge non-repeat fragments. After exhausting the list of fragments, constraints such as clone length and mate-pair distance are imposed in an attempt to add repetitive fragments to a consensus assembly. In contrast with the approach described by Green [126], the TIGR and PCAP assemblers endeavour to identify repetitive regions early in the assembly process and use this information to increase the stringency of overlap criteria for suspect read fragments. Furthermore, the use of paired reads and clone length constraints enables these assemblers to walk into a repetitive region from both ends, relying on repeat-spanning clones to join two contigs.

An alternative greedy strategy was proposed by Mullikin and Ning [129] and used for the assembly of the 104Mbp *Caenorhabditis briggsae* genome. The
Phusion assembler parses a set of sequence reads for all $k$-mers at all base locations in every read. The resultant map of $k$-mers enables the detection of repetitive sequences and the construction of data structures called read relation matrices. A read relation matrix is used to determine, for each sequence read, all other reads that share any of the same $k$-mers. Each matrix effectively clusters overlapping reads without recourse to the quadratic space and time complexity of dynamic programming alignment algorithms. Halvak et al [124] used a similar $k$-mer centric strategy in the Atlas assembler, which tabulates $k$-mers to identify and label repeats. Decomposing sequence reads in such a manner, early in the assembly process, enables the analysis of genome-wide oligonucleotide frequencies [78], provides an estimate of true coverage depth and facilitates the identification and suppression of repeats.

While greedy approaches represent the simplest and most intuitive solution to the assembly problem, they are essentially manifestations of hill climbing algorithms and can be side-tracked by local optima [27, 133]. By focusing on the optimisation of a local objective function during contig assembly, greedy algorithms can be confounded by repeats, leading to erroneous decisions and subsequent misassemblies [24]. Consistent with greedy algorithms in general, this is particularly true if a poor decision is made early in the assembly process. The kernel of the problem is the inability of greedy algorithms to utilise long-range distance information, such as paired reads, BAC joins and synteny relationships during contig assembly. In the absence of global knowledge of the state space, greedy algorithms depend entirely upon local heuristics to guide assembly [133]. It should be emphasised however, that early greedy assemblers such as Phrap were designed for use in hierarchical sequencing projects, where assembly is inherently local in scope. Indeed, some assemblers designed for larger eukaryotic genomes, where the utilisation of long-range distance and positional constraints are essential, use greedy algorithms for initial contig assembly before imposing long-range constraints in a subsequent scaffolding phase [123, 124, 129]. GigAssembler, developed for the original assembly of the public draft human genome [131] delegates contig construction to Phrap, but uses its own internal algorithms to construct contig scaffolds using BACs, ESTs and paired read constraints.
Both the Phusion and Atlas assemblers use Phrap in a similar manner for local contig assembly.

Despite the limitations of greedy assembly algorithms, the longevity of the approach is a testimony to its general effectiveness. Indeed, many next-generation sequence assemblers, such as those developed by Warren et al [83] and Dohm et al [82] use greedy heuristics to direct the assembly of short reads. These newer k-mer centric greedy algorithms apply a BLAST-like “seed and extend” strategy to greedily merge short reads. SSAKE [83] constructs a sorted hash map of k-mers and their multiplicities from a set of short input sequences and searches through a prefix tree [87] for the largest possible k-mer between two reads. Once a seed is selected from the hash map, it is greedily extended in its 3’ direction to create a contig. A more refined version of the same approach was used by Jeck et al [134] in the VCAKE assembler, which considers all reads that overlap with a seed sequence and generates contigs by extending seeds one base at a time, using the most commonly represented base from the matching set of reads. Furthermore, k-mer centric greedy assemblers designed for Sanger sequence reads can be adapted to SGS platforms, as evidenced by the recent sequencing and assembly of the Tasmanian Devil genome by Murchison et al [135] from Illumina sequence reads using the Phusion assembler.

2.2.2 The Overlap-Layout-Consensus Model

In a seminal work, that was to lay the foundation for a whole generation of genome assemblers, Kececioglu and Myers [31] proposed the overlap graph as a solution to the Sequence Reconstruction Problem, a superset of the SCS. Given a collection $C$ of fragment sequences and an error rate $0 \leq \varepsilon < 1$, the problem is to find a shortest sequence $S$ such that for every fragment $A \in C$, there is a substring $B$ of $S$ such that $\min (d(A, B), d(\tilde{A}, B)) \leq \varepsilon |A|$, where $d$ is a function of the edit distance between two sequences. This formulation has proved durable and underlies one of the two main paradigms for genome sequence assembly. The Overlap-Layout-Consensus (OLC) model [118] modularises genome assembly into three discrete phases and models read
fragments as nodes in a graph and overlaps as bi-directional edges [24, 30, 57, 69]. For two strings $s$ and $t$ over an alphabet $\Sigma$, if a suffix of $s$ matches a prefix of $t$, then $s$ is said to overlap $t$. If $o(s, t)$ is the length of the longest such overlap, then the overlap graph of a set of strings $S$, is a weighted directed graph, where each string is a vertex and the edge $s \rightarrow t$ exists in the graph with a weight of $|t| - o(s, t)$ if $o(s, t) \geq k$, for a minimum overlap threshold of $k$ [120].

Fig 2.1. An overlap graph created from a set of ten 8bp reads. Each read is represented as a graph node. Edges join nodes where reads overlap by a minimum of 5bps. Transitive overlaps are denoted by green edges. Edges induced by repetitive sequences are depicted in red.

With an adequate level of coverage, each contig is represented in the graph as a path that contains each node once. Thus, assemblers based on this strategy attempt to identify a Hamiltonian Path through the graph as a mechanism for sequence reconstruction [57]. As the Hamiltonian Path problem is NP-complete [78, 136], implementations of this approach must resort to heuristics and constraints when expanding nodes during graph transversal. Despite this limitation, the OLC paradigm is one of the two de facto standard models for genome sequence assembly and has been successfully adapted to accommodate the short read length, deep coverage and error characteristics of SGS data [34, 137].
The overlap phase of assembly is primarily concerned with the creation of a directed, edge-weighted graph by computing all pairwise alignments between reads [31]. Error correction procedures, such as base quality scoring, typically precede the construction of the overlap graph [69]. The computation of overlapping reads invariably involves the application or either dynamic programming algorithms, \(k\)-mer extension techniques or a combination of both [24, 27, 31, 69, 116]. As read orientation is unknown, all overlaps between reads, in both forward and reverse complement directions, must be computed. To circumvent the \(O(n^2)\) time complexity required to compute an “all against all” alignment [77], “seed and extend” heuristics are often utilised to compute \(k\)-mer overlaps with dynamic programming then employed to allow inexact matching. The overlap phase is thus sensitive to parameters such as \(k\)-mer size, minimal overlap threshold and the minimum percentage identity required for an overlap [27, 30]. Larger sizes of \(k\) increase minimal exact matching criteria, resulting in reduced assembly errors but shorter contig sizes [27]. This limitation is not a shortcoming of the OLC approach, but applies to all \(k\)-mer centric techniques in general and to de Bruijn graphs in particular [4, 47].

The primary function of the layout phase is the simplification of the overlap graph, by identifying and removing transitive edges and resolving ambiguities [27]. Given a sufficient level of coverage, the ultimate goal of this phase is the identification of a single path through the graph that visits each node once. If such a Hamiltonian Path exists, it corresponds to a tiling path through the genome allowing for the reconstruction of the entire sequence of the DNA molecule [24]. Myers [65] refers to the reduced layout graph as a string graph and has gives algorithms for both removing transitive edges and heuristically computing a minimum weighted path. Although Medvedev et al [119] have shown the minimum walk problem in a string graph to be NP-hard, Simpson and Durbin [66] have described how the heuristic techniques used to compute paths through de Bruijn graphs can also be applied to a string graph.

Repetitive regions and incomplete information confound the reconstruction of the genome as a single contig [110]. Commensurate with other approaches, OLC assemblers output a collection of contigs, where each contig represents
a tiling of sequence reads. The set of contigs produced corresponds to a set of non-intersecting simple paths in the reduced overlap graph [57]. Further processing of the collection of generated contigs is possible however, through the creation of an assembly scaffold. A scaffold is a set of contigs that are ordered, oriented and positioned relative to one another through the judicious enforcement of paired read constraints and, if available, long-range distance constraints, such as BAC links [69]. Koren et al [138] define scaffolding as a process through which paired read information is used to increase contig size and to compute a global alignment of contigs along a genome. The scaffolding process is an important element of the OLC paradigm, as it permits both global and hierarchical analysis of the assembly before a final consensus sequence is computed [69]. Furthermore, by rigorously enforcing constraints, scaffolding can help identify and fill gaps in an assembly that arise from adjacent sequence reads that share an insufficient length of overlap [139].

The final phase of the OLC approach is the computation of a consensus sequence [140]. A consensus sequence can be generated for each contig by constructing a multiple alignment of the reads, consistent with a given path though the overlap graph [118]. This typically involves each overlapping read “voting” for the consensus base at each position in a contig.

The Celera assembler, originally developed by Myers et al [69] for the assembly of the 120Mbp Drosophila melanogaster genome, epitomises the OLC approach. In addition to being the first software capable of assembling large eukaryotic genomes from WGS sequencing projects, Celera introduced a number of design innovations and concepts that have since been incorporated into most genome assemblers. Celera employs a “seed and extend” algorithm to align each sequence read during the overlap phase. A module called “Unitigger” then searches the overlap graph for collections of reads whose arrangement is uncontested by other reads. These read collections, called unitigs, represent sections of a genome that are entirely contained in either unique sequences or repetitive regions [138]. More formally, a unitig is a maximal interval subgraph of the graph of all fragments, where there are no conflicting overlaps with an interior node [118].
definition therefore, unitigs do not cross the boundaries between repetitive sequences and unique regions and thus enable the unambiguous reconstruction of the DNA sequence they contain [138]. Unitigs may be formed from repetitive regions where a collection of sequence reads is sampled within the boundaries of a very high fidelity repeat. Celera determines these unitigs through the application of statistical measures and uses this information to identify and label repeat boundaries [69].

A k-mer based approach to unitig construction was proposed by Batzoglou et al [77, 139] and used for the assembly of the 2.6Gbp mouse genome [141]. The Arachne assembler uses a “sort and extend” strategy, with an $O(n)$ time complexity, to detect overlaps and identify unitigs [77]. Arachne generates a table of $k$-mers, by default 24-mers, from the set of input sequence reads and sorts the table so that identical $k$-mers appear consecutively. As high-fidelity repetitive regions give rise to $k$-mers with a high copy number, these can be easily identified and are eliminated [139]. Arachne then identifies all instances of read pairs that share one or more overlapping $k$-mers. These shared $k$-mers are merged and then extended using dynamic programming techniques. Identifying and merging unitigs in this manner is conceptually similar to the $k$-mer compression techniques used by many de Bruijn graph assemblers [27]. While the Minimus assembler described by Sommer et al [140] also uses a $k$-mer based hash-overlap to identify unitigs, this lightweight assembler is primarily designed for gap closure of draft genomes and lacks the sophistication and rigour of an ab initio genome assembler.

As unitigs are generally small, OLC assemblers must use additional information, such as paired reads, to increase contig size [138]. An assembly scaffold can be created, by linking together all unique unitigs with mutually confirming paired reads or BAC joins. Where left and right paired reads belong to different unitigs, their distance relation allows the orientation of the two unitigs to be determined and provides an accurate estimate of the distance between them [69]. If the relationship between two unitigs in a scaffold is unambiguous, they can be safely merged into a single contig. Scaffolding also facilitates the filling of gaps between contigs. Both Celera and Arachne utilise sequence quality scores and aggressive repeat resolution procedures to merge
together unitigs and contigs that are separated by a distance less than the minimal required overlap. In contrast with the approach used by Myers et al [69], Arachne relies on the assembly scaffold to identify repeat boundaries and merges reads, using paired read constraints, into contigs up to these boundaries. The Minimus [140] assembler also uses scaffolding to order and orient reads, but delegates this task to the modular and highly configurable Bambus [110] package. A notable feature of Bambus is the use of edge bundling, a scaffold simplification and reduction technique, where all links between two adjacent contigs are combined into a single weighted edge. A greedy algorithm is then used to order and orientate the contigs, using the weighted bundled edges as a contig selection heuristic. The correct order and orientation of contigs in an assembly scaffold is predicated on the accuracy and correctness of the links that joint them. If the linking information contains errors, the general problem of globally satisfying all scaffold constraints becomes intractable [110].

Fig 2.2. Transitive reduction in an overlap graph. The set of 10 nodes shown earlier in Fig 2.1 is reduced to 6 nodes by removing edges that are implied by longer overlaps. Unitigs are formed by merging the reads in unambiguous nodes up to the boundaries of a repeat.

One of the more salient features of the OLC paradigm is the removal of transitive edges from the overlap graph [118]. While essential for simplifying the overlap graph, transitive reduction has major implications for the use of the OLC paradigm for the assembly of short or mixed-length reads [138].
Transitive edges represent contained reads that are aligned end-to-end inside another read [69, 77] and their removal reduces the complexity of the overlap graph by a factor of the level of genome coverage [65]. By removing these edges, information is effectively lost from the overlap graph. When assembling SGS data, the vast majority of connections in an overlap graph correspond to transitive edges [137]. In addition, the over-collapsed alignments of repetitive short reads can induce both correct and spurious overlaps inside longer reads, thereby introducing errors into the graph [34].

Furthermore, traditional mechanisms for detecting repeat boundaries may fail with SGS data as, due to their short length, reads extending beyond a repeat boundary may not overlap sufficiently. In such a case, an assembler may detect a misassembly, resulting in unnecessarily shorter contigs [34].

The different error characteristics of SGS data can also constrain correct assembly. Homopolymer runs in short reads give rise to the formation of tips or spurs in an overlap graph, while single-base polymorphisms result in the formation of bubbles or “bulges” [137]. Homopolymer runs can also confound the exact matching k-mer based techniques used for computing overlaps and detecting repetitive regions [34]. In addition, the high levels of coverage, typical of SGS data, result in larger numbers of reads with exactly the same k-mer prefix or suffix. This can hamper the tie-breaking heuristics employed by more sophisticated assemblers [69] and can hinder the imposition of constraints and the identification of global optima during scaffolding [34]. OLC assemblers designed for use with Sanger sequences are thus particularly sensitive to variability in average read length from different sequencing platforms. For these reasons, the initial assemblers designed for SGS data primarily employed the de Bruijn graph model of assembly [29, 32, 33, 47-49].

A greedy solution to the issues with the OLC paradigm arising from mixed and short read assembly was implemented by Miller et al [34] in the Cabog assembler. Cabog is a hybrid assembler, capable of assembling a mixture of long Sanger reads and the shorter read lengths produced by SGS sequencers. Cabog is a refactoring of the Celera assembler, developed by Myers et al [69],
but narrows the source of hybrid assembly problems by only reusing the scaffolding and consensus modules of the original assembler [34]. Cabog applies an aggressive, greedy, approach to unitig construction to address the problems of hybrid assembly using an overlap graph. In particular, to accommodate homopolymer runs and transitive reads, the assembler uses novel algorithmic techniques during the overlap and layout phases. Although the assembler uses a k-mer centric approach for overlap detection, read trimming is used to compensate for homopolymer runs by compressing runs of identical consecutive bases into a single nucleotide symbol. The set of k-mers produced after read trimming is tabulated to identify unitigs and to avoid merging repeats.

Cabog also makes substantial modifications to the traditional implementation of the layout phase, by supplanting the overlap graph with a Best Overlap Graph (BOG) [34]. A BOG is a multigraph, consisting of reads and both directed and undirected edges. Each read is modelled as a pair of nodes connected by an undirected edge, where the nodes represent the two ends of a read. Directed edges correspond to dovetail overlaps [31], spanning just one end of each read. Unitigs are merged aggressively, using mate-pair constraints as a greedy heuristic. The underlying Celera modules are then used for scaffolding and computing a consensus sequence. It is notable that a transitive reduction step is still applied during construction of the BOG. Cabog creates only one directed edge for each node, by selecting the read at a given base with the “best” overlap, thus discarding transitive reads [34]. For a hybrid assembly, this clearly creates a bias towards longer Sanger reads.

A tree-based OLC assembler, designed for short sequence reads, was developed and reported by Hernandez et al [137]. The Edena assembler indexes all sequence reads in a prefix tree [87], where each read and its reverse complement are merged in the same tree key. The approach used by Hernandez et al effectively applies two stages of transitive reduction. Initially, a non-redundant and lossless set of reads is extracted from the prefix tree. The occurrence frequency of each read is also computed and used later to estimate coverage depth during contig assembly. The overlap phase is completed by indexing the non-redundant set of reads in a lexicographically sorted suffix
array [84], revealing exact matches between sequences at low space and time complexity. Transitive reduction is also applied to the bi-directional layout graph created from overlapping reads, by removing all transitive edges. Tips and bubbles in the graph, caused by homopolymer runs and single base polymorphisms, are removed using path length heuristics.

One of the defining properties of the OLC paradigm is the capability of applying a level of global direction to the sequence reconstruction problem [31]. This contrasts strongly with greedy approaches to assembly, where decisions relating to contig construction, order and orientation are constrained by a localised scope. Global information about the graph model is particularly important when assembling larger eukaryotic genomes, where repetitive regions can easily cause the collapse of an assembly scaffold [24, 116]. The OLC approach has also proven adaptable to both hierarchical and WGS sequencing strategies [11]. The use of BAC libraries provides OLC assemblers with a much-simplified computational task, allowing long-range global information to be incorporated into the overlap graph. In addition, the OLC paradigm is inherently modular and the overlap graph structure lends itself to multiple types of analysis [57]. Modularity allows for the parallelisation of the overlap phase [24] and for alternative scaffolding algorithms to be used [138]. Furthermore, paired read information can be added as edges to the overlap graph allowing for more informed decisions when merging unitigs and applying scaffold constraints [24].

Despite the merits of these properties, the OLC approach is not without it limitations. The computation of overlaps is typically undertaken using both k-mer centric techniques and dynamic programming, the former requiring an exact match and the latter having a quadratic space and time complexity [69, 77, 118]. Furthermore, without redesign of the overlap and layout implementations, transitive reduction renders this approach unsuited to the assembly of short or mixed-length reads [30]. In addition, as OLC assemblers retain information about each read, this can lead to scalability issues when assembling SGS data at very high levels of coverage [4]. The central importance of read pairs during the layout phase also raises issues concerning
the suitability of the approach for SGS data, where sequencing using paired reads can be difficult and expensive [24].

2.2.3 The De Bruijn Graph Model

In 1989, Pevzner [142] suggested a fundamentally different approach to genome assembly – the reconstruction of a DNA sequence from its k-mer composition. Idury and Waterman [143] later expanded this idea, advocating the use of a sequence graph to assemble a genome from the k-mers generated from sequencing by hybridisation on DNA microchips. Although sequencing by hybridization failed as genome sequencing platform [109], the mathematical model proposed by Idury and Waterman provided a conceptual framework that was later used by Pevzner and Tang [78] to revolutionise genome assembly. The kernel of the approach proposed by Pevzner is the reformulation of the fragment assembly problem as a Sequencing by Hybridization (SBH) problem: Given a set of k-mers, \( S = \{s_1 \ldots s_n\} \), find the shortest string \( s \) such that every k-mer \( s_i \) appears as a substring of \( s \) and every k-mer from \( s \) appears as a k-mer in \( S \) [125]. Although it appears similar to the Shortest Common Superstring problem [117], the SBH formulation transforms the contig assembly problem from finding a Hamiltonian Path in a layout graph, to that of finding an Eulerian path in a de Bruijn graph [57].

A de Bruijn graph is analogous to the output of a shotgun sequencing experiment that perfectly samples a genome, generating a set of fixed-length reads with a read starting at each base position [24]. The assembly process can be viewed as the merging of unambiguous nodes in the graph [48]. The k-mer centric nature of this approach makes it particularly apposite for assembling SGS data, where short reads are sampled from genomes at very high levels of coverage [32, 33, 48, 49, 144, 145]. A k-dimensional de Bruijn graph, \( G = (V, E) \), on an alphabet \( \Sigma \), is a directed graph that contains all possible strings of length \( k \) over the alphabet as vertices, i.e. a complete graph will contain \( \Sigma^k \) vertices [29]. An edge exists from vertex \( u \) to \( v \) if, by deleting the first character of \( u \) and the last character of \( v \) both vertices share the same string, i.e. \( E = \{(u, v) \mid u, v \in V \text{ and } u[i + 1] = v[i], 0 \leq i < k - 1\} \).
Fig 2.3. The 6-mer de Bruijn graph created from a set of ten 8bp reads. Repetitive nodes are depicted in yellow and are easily identified from the multiplicity of edges incident on or emanating from a node.

For a genome of size $n$, the de Bruijn graph will have $O(n)$ nodes and $O(n)$ edges, regardless of the number of reads in the assembly [78]. In theory therefore, the graph size and corresponding memory consumption is determined by the size and repeat content of the genome alone and is unaffected by the high levels of redundancy induced by deep coverage [33]. In practice however, the actual implementation of the de Bruijn graph will have a significant impact on memory consumption and can easily exhaust available memory, especially on large genomes [27]. The set of $k$-mers for a genome or collection of reads is easily extracted by sliding a window of size $k$ along a target sequence [48]. As nodes in the graph are joined with an edge where they overlap by $k – 1$ bases, the additional sequence information stored by each node is its last base [49].

A de Bruijn graph is highly sensitive to the minimum overlap determined by the value of $k$ and thus to both read length and genome coverage [47]. In essence, the size of $k$ represents a Morton’s Fork between sensitivity and specificity [12]. Smaller values of $k$ increase the connectivity of the graph, but can result in excessive tangling, fragmentation and shorter contigs if the value of $k$ is too small [29]. While a smaller value of $k$ lowers the overlap
threshold for sequence reads, it also increases the number of ambiguous repeats in the graph [49]. Larger values of $k$ facilitate the resolution of repetitive sequences, but can fracture an assembly in regions of low coverage [27]. In addition, increasing the value of $k$ requires either increasing read length or raising the level of coverage [47]. This is especially pertinent for short reads, where an increase in genome coverage is necessary to ensure that a sufficient number of reads overlap, with the number of true overlaps increasing quadratically with coverage [25]. Peng et al [146] have described an iterative method for constructing a de Bruijn graph that captures the merits of all $k$ values between lower and upper bound thresholds. While capable of producing assemblies with a high contiguity, the approach they describe requires the maintenance of an accumulative de Bruijn graph at each iteration.

The de Bruijn graph model has several properties that make it meritorious of consideration for genome assembly. A de Bruijn graph obviates the need for an overlap phase, as overlaps are already implicit in the graph as edges that connect adjacent nodes [30]. As a de Bruijn graph is $k$-mer centric, the graph topology is unaffected by the length of read fragments [49]. Thus, in contrast to the OLC approach, assemblies can be created from mixed-length fragments [32]. Moreover, by virtue of the one-to-one relationship between sequences and paths, overlapping sequences necessarily follow the same path [144]. This simplifies the search for consistently overlapping reads [30]. The decoration of edges with edge-weights allows for the rapid identification of repetitive regions [43]. In addition, mate-pairs, essential for the assembly of more complex genomes, can easily be incorporated as edges into the graph [78]. Indeed, Chaisson et al [58] have suggested that, provided a form of mate-pair sequencing technology is used, read lengths in excess of 40 bases may be unwarranted. Furthermore, the de Bruijn is highly parallelisable, as the de Bruijn graph for the union of two sets of reads is the union of the de Bruijn graphs for each independent set of reads [29]. Given the space complexity of the de Bruijn graph, parallelisation provides a mechanism for scalability and was used by Li et al [35], Simpson et al [48], Boisvert et al [32] and, more recently, by Gnerre et al [147] to reconstruct large mammalian genomes from sets of short sequence reads.
Employing a de Bruijn graph structure for fragment assembly requires an algorithmic solution that is guaranteed to visit each edge in the graph [32, 57]. The identification of such a path can be formulated as an instance of the Chinese Postman Problem [148]: finding a minimum weighted path through the graph that visits each edge at least once [119]. This problem is closely related to the more accurate Eulerian path formulation of fragment assembly in a de Bruijn graph - the identification of a path through the graph that visits each edge exactly once [136]. In their original proposal, Pevzner and Tang [78] argued that the k-mers should be mapped to edges in the graph, with overlapping k-mers joined at their tips. As an exponential number of Eulerian paths may exist, corresponding to the various ways a sequence can be arranged around repetitive regions [120], they proposed a Eulerian Superpath solution [78]. In a Eulerian Superpath, each read corresponds to a path in the de Bruijn graph, called a read path. Sequence reconstruction can then be formulated as the identification of a Eulerian path that is consistent with all read paths. An alternative approach was employed by Zerbino and Birney [49] for the design of the short-read assembler Velvet. The Velvet assembler maps k-mers to nodes and relies on a heuristically informed “Tour Bus” algorithm [149] to compute an optimal path through the graph. Used in this manner, where nodes represent sequences and paths possible assemblies, a de Bruijn graph shares many similarities with the layout graph used in the OLC approach.

In the de Bruijn graph model, assembly is a by-product of graph construction and more than one type of construction is possible [27]. The original proposal by Idury and Waterman [143] modelled sequences as edges, with separate nodes and edges for each DNA strand. In this construction, care must be taken to ensure that a component or the entirety of a genome is not assembled twice [27]. This type of construction is used in the Euler [78] assembler, which searches for two complementary walks, each corresponding to one of the DNA strands. Medvedev et al [119] suggest that, because DNA is double-stranded, it is necessary to model k-molecules instead of k-mers and shows how a bi-directed de Bruijn graph can be constructed to represent complementary strands. A variation of this approach was used by Zerbino and
Birney [49] who employed a twin-node for both the forward and reverse complements of each $k$-mer, with the constraint that paths must enter and exit from the same half node. A bi-directed graph was also used by Simpson et al [48], who modelled opposite strands as a single node with two sides, constraining paths to enter and exit from opposite sides. More recently, Medvedev et al [150] have proposed a generalisation of the de Bruijn graph that incorporates mate pair information into the graph structure itself. This paired de Bruijn Graph allows mate pair constraints to be enforced during the computation of putative paths, instead of analysing paired read constraints as a post-processing step. While these different constructions may appear relatively insignificant, the manner in which $k$-mers are represented in a de Bruijn graph has notable implications for both the merging of unambiguous nodes and the computability of an assembly path [27].

As the fundamental structure of a de Bruijn graph is based on $k$-mers, the graph topology facilitates both the identification of repetitive regions [43] and the initial assembly of contigs [143]. In contrast with the OLC paradigm, where heuristics such as $k$-mer frequencies and coverage metrics are used to identify repetitive genomic regions [34, 69, 77], each repeat in a de Bruijn graph is only present once and contains explicit links to its starting and end points [49]. Consistent with a layout graph, all copies of a repeat will collapse into a single high-coverage node in a de Bruijn graph [30]. Repeats longer than $k$ induce tangles in a $k$-mer graph [142], with high fidelity repeats having a topological effect on the de Bruijn graph that leaves a local graph structure resembling a rope with frayed ends [27]. Palindromic sequences can further confound assembly, by inducing paths in the graph that fold back upon themselves. A solution to the problem of palindromes was provided by Zerbino and Birney [49], who noted that odd-length $k$-mers cannot be palindromic. However, as cycles and tangles are easily identified as branching nodes, the graph topology itself provides a succinct representation of the repetitiveness of a genome [151].

The topological features of $k$-mer graphs were identified by Idury and Waterman as key properties for facilitating genome assembly [143]. The sequence graph they described can be created from a de Bruijn graph, by
merging together nodes in the graph when they are unambiguously connected, i.e. consecutive nodes that have an outdegree and an indegree of one [49]. Compressing nodes in this manner is logically equivalent to the formation of unitigs in the OLC paradigm and forms the first phase of contig assembly in a de Bruijn graph [27]. Called the “elimination of singletons” by Idury and Waterman [143], in contrast with the OLC approach, node compression is unambiguous and can be accomplished in linear time, with low space complexity, using a depth-first search [152]. Quitzau and Stoye [144] have described algorithms for creating a sequence graph directly from a set of sequence reads, obviating the need to first construct a de Bruijn graph. Formally, for a de Bruijn graph $G$ and a given size of $k$, a unitig may be defined as follows [47]: A genomic sequence $S$ of length $l \geq k$ can be represented as a sequence of successively adjacent $k$-mers $<s_1, s_2, \ldots, s_n>$ where $n = l - k + 1$. If $<s_1, s_2, \ldots, s_{n-1}>$ has an outdegree of 1 and $<s_2, s_3, \ldots, s_n>$ has an indegree of 1 and $S$ cannot be extended without violating these constraints, then $S$ is a unitig.

**Fig 2.4.** Elimination of singletons in a de Bruijn graph. Nodes with an indegree and an out-degree of one represent unambiguous paths and can be merged together. Merging nodes is logically equivalent to finding unitigs in the OLC model and creates an initial set of proto-contigs.

The formation of unitigs by the elimination of singletons not only reduces the graph size and memory consumption, but also greatly simplifies the graph, facilitating the identification of sequencing errors, correct path transversal and subsequent assembly [27, 32]. Furthermore, as unambiguous nodes are
merged by this method, any nodes in the graph of size \( k \) are easily identifiable as repetitive sequences. Indeed, both Butler \textit{et al} [47] and Gnerre \textit{et al} [147] used this technique in the AllPaths assembler to build contigs and, after discarding the de Bruijn graph, scaffold the contigs in a manner consistent with the OLC paradigm. The elimination of singletons and formation of unitigs further enables the unambiguous identification of repeat boundaries in the de Bruijn graph model and helps prevent the creation of artificial paths that do not exist in the genome [4]. It is noteworthy that the modelling of nodes in a de Bruijn graph directly affects the degree to which the elimination of singletons may be used to construct unitigs. Modelling the forward and reverse complements of a DNA strand as a single node [48, 149] induces branches and reduces the number of unambiguous paths in the graph.

One of the most important facets of de Bruijn graphs is their intolerance of sequencing errors [30]. This intolerance is a direct consequence of the \( k \)-mer centric nature of the data structure, where a single base error induces up to \( k \) spurious nodes in a \( k \)-mer graph [27]. In a layout graph, error correction is implicitly contained in the dynamic programming parameters used to validate the overlaps between reads [118]. Because a layout graph contains global assembly information, isolated sequencing errors are too small to adversely affect the assembly [4]. They may, however, give rise to issues during the consensus phase, by complicating the computation of a multiple sequence alignment. In a de Bruijn graph, every variant of a \( k \)-mer will directly affect the topography of the graph by creating distinct features such as tips, bubbles or branches [144, 153]. Sequencing errors, including polymorphisms, insertions and deletions can transform a simple de Bruijn graph into a tangle of erroneous edges [78]. An error correction mechanism is therefore essential, as the additional paths induced by sequencing errors can confound assembly, resulting in smaller contigs than necessary [4]. In addition, the elimination of erroneous \( k \)-mers, prior to constructing the de Bruijn graph, greatly reduces memory consumption, allowing for the assembly of larger sets of sequences [4].

In their original formulation of the Eulerian Superpath problem, Pevzner and Tang [78] proposed filtering errors from reads prior to assembly using a
method called Spectral Alignment (SA). The SA method identifies sequence errors by analysing the occurrence frequency of \( k \)-mers between different reads and then creating a spectrum of \( k \)-mers that occur with a multiplicity above a given threshold. For each read, the SA method computes the minimum number of insertions, substitutions and deletions required to make every \( k \)-mer of the read belong to the spectrum. Reads containing low frequency \( k \)-mers are either discarded, or corrected by coercing erroneous \( k \)-mers to conform to the spectrum. While this type of error correction eliminates a large number of spurious potential nodes, Yang et al [45] argue that the twin assumptions of uniform coverage and error distribution are violated in read data sets, giving rise to errors in the de Bruijn graph by masking true polymorphisms. Both Butler et al [47] and Li et al [33] employed the SA error correction method in the AllPaths and SOAPDeNovo assemblers. The versatility of the technique is illustrated by its application to the OLC approach by Batzoglou et al [77] and Miller et al [34], who used variant of the SA method to filter reads prior to constructing an overlap graph.

In addition to filtering sequence errors prior to constructing a de Bruijn graph, *post hoc* error correction is also possible, by analysing the topological features of the graph [30]. The Velvet [49] assembler relies exclusively on this method of error correction, using heuristics to prune tips and spurs from a graph and to deflate bubbles [149]. Spurs or tips are typically identified as low coverage branches in a graph, whose combined contig length is less than a specified threshold [30]. Pruning spurs using path length heuristics has been employed in a number of de Bruijn graph assemblers [29, 33, 48, 49, 78] and has also been applied to the OLC model [137]. Detection and resolution of redundant paths or bubbles requires more complex algorithms. The Tour Bus algorithm, described by Zerbino and Birney [49], identifies bubbles by fanning out from branching nodes in a breadth-first manner, searching for redundant paths that start and end at the same nodes. Once a bubble has been identified, the lowest coverage path can be removed and its constituent reads re-aligned with the remaining path. Both Simpson et al [48] and Li et al [33] reported using similar techniques for removing bubbles.
Raphael et al [148] suggested an alternative post hoc error correction mechanism, based on dynamic programming. However, the quadratic space and time complexity involved renders this approach expensive to apply to highly redundant data structures and to sequencing projects with short reads. Rather than attempting to correct and eliminate errors, other approaches seek to incorporate k-mer differences into the de Bruijn graph. Challa and Thulasiraman [153] reported a protein motif discovery application that uses hamming distance as a metric for determining k-mer similarity. In this approach, k-mers within a specified hamming distance are collapsed into a single edge, allowing the detection of subtle or weak motifs.

The problem of motif discovery in proteins also led to the proposal by Patwardhan et al [44] of an “approximate” de Bruijn graph. In a conventional de Bruijn graph, the weight of each edge is defined as the number of identical matching k-mers in the input sequence [119]. The approximate de Bruijn graph is a generalisation of this concept, where the graph has the same topology, but a different mechanism is used for computing edge weights [44]. Used in the context of motif detection in proteins, edges are adjudged to be similar if every pair of corresponding residues in two sub-sequences has a positive amino acid substitution score. Patwardhan et al [44] also developed an algorithm for gapped motif discovery in a de Bruijn graph. Using a set of predetermined masks, that are added as nodes to the graph, k-mers are filtered and edges added using a logical AND operation. The application of masks in this manner is very similar to the use of spaced seeds in many of the newer sequence aligners and results in a much more connected graph.

Although the unique properties of de Bruijn graphs offer cogent advantages over alternative approaches to de novo assembly, the model has its limitations. Without parallelisation, the high space complexity precludes the de Bruijn graph approach from being used to assemble large mammalian-sized genomes [30, 57]. Even with parallelisation, distributing the graph over a cluster can lead to a fragmented assembly [33]. Moreover, de Bruijn graphs are not read coherent, i.e. there may exist paths through the graph that are not supported by the underlying reads [65]. Identifying and avoiding such paths is crucial for a correct assembly. Depending on the construction of the graph,
decomposing a read into a tiling path of $k$-mers results in a loss of information [30]. In addition, although the de Bruijn model exhibits intrinsic high efficiency in identifying repeats, due to the lack of read coherence it has the corresponding weakness of low efficiency in utilising longer reads [4].

2.3 Comparative Approaches to Genome Assembly

As the number of sequenced organisms increases, alternative approaches to genome assembly, based on orthologous relationships, become ever more viable [67]. Auch et al [154, 155] have demonstrated how the degree of homology between two genomes can be accurately measured, by aligning the two sequences and then inferring the genetic distance between them from the set of high-scoring pairs produced. The multiplicity of completed genomes and the availability of accurate metrics to compute homology provide cogent reasons for the consideration of comparative approaches to genome assembly alongside the de novo models already discussed.

In a seminal work, Pop et al [68] suggested an alternative to de novo assembly that exploits the shared synteny between a set of shotgun sequence reads and a finished genome. Comparative assembly algorithms map sequence reads to a high-quality reference genome and use the resultant anchoring information to direct the assembly process [137]. Known as Alignment-Layout-Consensus, the model developed by Pop et al [68] is based on the assumption that if two reads align to the same part of a reference genome, they must overlap. Their approach, implemented in the AMOS assembler, supplants the traditional overlap phase of assembly, which has a high space and time complexity, with a faster alignment phase, thereby drastically reducing the running time of the assembly process. The AMOS assembler delegates read alignment to MUMmer [16, 80, 81], which builds a suffix tree to expedite the rapid identification of unambiguous anchoring regions in a reference genome. If AMOS cannot unambiguously align a read against a reference, paired read information is used to determine an anchoring position. Refinement of the layout graph is followed by the creation of an
assembly scaffold. AMOS also delegates this task, using the Bambus [110] package to order and orientate contigs.

The AMOS package was later augmented by the work of Salzberg et al [156], who developed an assembly method that exploits the high degree of conservation in protein sequences. Known as “gene-boosted assembly”, the ABBA assembler employs this technique, using AMOS [68] to perform an initial comparative assembly, before joining contigs and closing gaps with the protein sequences from a related species. A variation of the Alignment-Layout-Consensus model was also employed by Strömberg [102] in the Mosaik assembler. Designed for genome re-sequencing, the reference-guided assembler uses multiple spaced seeds to align sequence reads against a reference genome. In contrast with the original proposal by Pop et al [68], regions of the reference genome that reads align with are also assembled into contigs.

Comparative approaches have also been used to order and orientate the set of contigs generated from a de novo assembly [157-162]. Using one or more reference genomes, these techniques are designed to facilitate closing gaps in an assembly, by comparing a set of contigs against the finished assemblies of closely related species. If the correct order and orientation of \( n \) contigs are known, the number of primer walks required for gap closure is reduced from \( O(n^2) \) to \( O(n) \), greatly decreasing the time and expense required to finish a genome [158]. Assemblers that utilise BLAST [52, 53] to determine matched synteny between a contig set and a reference genome have been reported by van Hijum et al [161, 163], Richter et al [159] and Zhao et al [162, 164]. An alternative approach using \( q \)-gram filters [90] was used by Hauseman and Stoye [157] to match and visualise the synteny of contigs and a reference assembly. The Mauve assembler, developed by Rissman et al [160], filters and sorts alignments into local co-linear blocks (LCBs), representing homologous sequences in two or more genomes, analogous to the concept of anchors described by Pop et al [68]. However, these post hoc methods of contig processing are limited to the assembly of contigs derived from different strains of the same species, which share either identical or highly conserved gene arrangements [164]. More recently, Husemann and Stoye
reported a technique that uses the phylogenetic distance to a reference genome to order and orient contigs in the presence of structural variations and genomic re-arrangements. In addition, Zhao et al [162, 164] has demonstrated how genetic algorithms can be employed to search globally for an optimal placement of contigs, using a reference genome from a more distantly related species.

The related, but distinct concept of assisted assembly was proposed by Gnerre et al [63] to assemble the genome of *Canis familiaris* from 2X coverage sequence reads. Designed for use with low-coverage sequences, assisted assembly reinforces information already present in reads to detect erroneous or missed overlaps during the initial phase of genome assembly. Simultaneously constructing both a *de novo* and a comparative assembly, proximity relationships between reads are exploited to guide the assembly process. A local anchoring of each read against a reference genome is followed by the grouping of reads into proto-contigs. These grouped reads are then used to enlarge the contigs created by the *de novo* assembly. The approach proposed by Gnerre et al uses BLASTz [165] to align a set of shotgun sequence reads to a reference genome and delegates *de novo* assembly to Arachne [77, 139]. A similar method was used by Reinhardt et al [166] to assemble the genome of *Pseudomonas syringae* from a hybrid assembly of low coverage Sanger sequences and SGS reads. They used the VCAKE [134] assembler to generate a set of contigs and build an assembly scaffold. The scaffold was subsequently refined, by aligning the scaffold assembly against the finished genome of *Pseudomonas oryzae* using BLAST [52, 53].

More recently, reference guided assembly was used by Schneeberger et al [167] to assemble the genome of the *Arabidopsis thaliana* plant from 36-80bp Illumina sequence reads, by aligning the contigs generated from a de Bruijn graph assembler to a reference genome using GenomeMapper [168]. A similar method was described by Cattonaro et al [169], who merged the contigs generated by a de Bruijn graph assembler with the set of alignments produced by mapping shotgun reads against a reference genome with SOAP [22]. A refinement of this idea was described by Nijkamp et al [170] who
developed an algorithm for integrating *de novo* and comparative assemblies, by constructing an overlap graph from the pairwise alignment of contigs and using a reference genome to guide the integration of the two assemblies.

To facilitate the analyses of different draft assemblies of the same species, Zim *et al* [171] proposed a method called “assembly reconciliation”. As different assemblers may produce different results from the same set of sequence reads [172], assembly reconciliation attempts to improve the quality and congruence of a genome assembly by merging together two or more draft assemblies. The study by Zim *et al* [171] demonstrated that assembly reconciliation can boost contig size and facilitate the identification and resolution of misassemblies. A later study by Casagrande *et al* [173] also showed how the approach can increase assembly quality, by removing contradictory components from an assembly and integrating the remaining elements. More recently, Yao *et al* [172] have demonstrated how an accordance graph can be employed to encapsulate the mapping of a draft assembly to a reference genome. The accordance graph they describe allows contigs and scaffolds from the draft assembly to be extended, merged or bridged together.

Despite the proliferation of completed genomes, current comparative approaches to assembly are limited by the requirement that the genome, or that of a closely related species, has already been sequenced and assembled [137]. Furthermore, current approaches require a high degree of structural fidelity between the genome being sequenced and the reference genome [30]. This is particularly true for the *Alignment-Layout-Consensus* model, which requires a very high degree of synteny between sequence reads and a reference genome [174]. In addition, as species become more diverged, it becomes more difficult to accurately align sequence reads against a reference [63]. Moreover, structural variations such as conserved synteny breakpoints, repeat insertions and segmental duplications may give rise to alignments that do not reflect the sequence of the novel genome [63]. Consequently, repetitive regions and structural variations in the reference genome can stymie comparative approaches and may give rise to errors in an assembly [30]. Notwithstanding the success of reference guided assembly reported by
Schneeberger et al [167] and others [63, 169, 170], the methods they describe are essentially post-assembly operations on contigs or scaffolds and do not directly integrate anchoring information from a reference genome into the de novo contig assembly process.

2.4 Measures of Assembly Quality

Based on previous work by Anderson [175] and Clarke [176], a robust mathematical model for single-end shotgun sequencing was proposed in 1988 by Lander and Waterman [62]. In their model, Lander and Waterman made assumptions that sequence fragments are entirely random and independent of one another. Assembling these fragments to accurately reconstruct a genome is predicated on the amount of oversampling employed [77]. The oversampling of a source results in overlaps between adjacent fragments and is the primary mechanism employed by an assembler for establishing links between sequence reads [64]. The level of oversampling is referred to as coverage [62] and is a function of the amount of sequence data generated.

The Lander-Waterman model enables the prediction of sequence coverage, genome size and the expected number and average size of contiguous sequences. The model also shows that base coverage follows a Poisson random distribution [177]. Later, Li and Waterman [42] demonstrated that k-mer coverage is also Poisson distributed. The Lander-Waterman model was augmented by the work of Fleischmann et al [61], who provided equations that can be used to estimate, inter alia, average gap size and total gap length. These models have proven indispensable for shotgun sequencing and assembly projects and have been used to monitor progress and to control deviations from expected values [61, 131].

Due to the short read lengths produced by SGS platforms, much higher levels of coverage are required to satisfy detectable overlap criteria in order to produce a high quality assembly [27]. Schatz et al [30] have argued that, even with high coverage, the basic assumptions of the Lander-Waterman model are violated by short reads, as a larger percentage of a genome is repetitive when
sampled at short read length. Moreover, the Lander-Waterman model acts as a general guide for sequencing and assembly and cannot provide any information about the level of contiguity and quality in an actual assembly.

Since the initial sequencing and assembly of the human genome [131], the N50 contig size has been the standard metric used to compute assembly quality. Salzberg et al [10] define the N50 value as the size of the smallest contig in an assembly, such that 50% of the genome is contained in contigs of size N50 or larger. Despite its widespread adaption to report the performance of different assembly algorithms, the N50 metric emphasises contig size and is a poor metric for capturing contig quality [57]. Greedy assembly strategies, such as those used by SOAPDeNovo [33], can produce assemblies with a high N50 value, but may contain a significant number of misassemblies [1]. In contrast, a highly conservative assembler, that only reports unitigs will have a low N50 value, but produce high quality contigs. Haiminen et al [111] raise a more fundamental concern about the limited confidence of de novo assemblies, as assembled contigs represent just one possible way of mapping reads to contiguous sequences.

Narzisi and Mishra [57] proposed an alternative method that measures both the contiguity and quality of assembled contigs. The Feature Response Curve (FRC) is based on the premise that features or suspicious substrings of contigs, such as incorrect paired read distances and collapsed repeats, can be identified. After sorting the set of assembled contigs by size and then applying a threshold value to each feature, only the longest contigs whose sum of features is less than the threshold are used to compute the curve.

Despite offering a measure of assembly quality lacking in traditional metrics, Vezzi et al [59, 178] have identified limitations with the FRC. In particular, they argue that the FRC method does not discriminate between the different features, their correlation and their relative importance. They also identified the requirement of a read layout as problematic. Although routinely available with Sanger sequence assemblers, the majority of de novo assemblers designed for SGS reads do not provide information on the read orientation and composition of assembled contigs. Vezzi et al [59] proposed an
enhancement of the FRC that computes the curve from an alignment of the original sequence reads against the assembled contigs and demonstrated the utility of the approach for assessing the quality of an assembly.

An alternative assembly quality assessment method was described by Salzberg et al [10]. Their technique requires the alignment of assembled sequences against a reference genome to identify misjoins in contigs, such as inversions, relocations and translocations. A corrected N50 value can then be computed that takes cognisance of any misassemblies. A similar technique for measuring assembly quality was recently described by Gurevich et al [179] and implemented in the QUAST tool. Although QUAST can compute metrics on a de novo assembly without recourse to a reference genome, the metrics produced yield little information about the quality of the contigs constructed. Using a reference genome however, enables QUAST to perform a robust and highly accurate assessment of assembly quality. The metrics produced using a reference genome include an extension of the N50 metric called the NGA50 value. This value is computed by a Nucmer [16, 80] alignment of contigs against a reference sequence and represents a recalculated N50 size after contigs have been assessed for misassemblies.

2.5 Chapter Summary

The utilisation and exploitation of k-mer centric techniques is long established in genome assembly and sequence alignment. K-mer matching underpins both the “seed and extend” and spaced seed strategies used by the majority of sequence aligners. The centrality of k-mer matching also applies to genome assembly where overlap detection, repeat identification and assembly models rely heavily on the approach. Despite the ubiquity of k-mer matching in bioinformatics, exact match criteria limit the sensitivity of the approach and it remains an active and open area of research.

After three decades of research, genome assembly remains an open problem, with no known optimal solution. Significant and on-going advances in sequencing platforms and technologies have enabled the rapid sequencing of
large genomes at high levels of coverage and at low cost. Assemblers designed for use with Sanger sequence reads are predominantly based on greedy algorithms or the *Overlap-Layout-Consensus* model and have proven ill suited to the high levels of read coverage and short read length produced by SGS sequencing platforms. Since the advent of SGS technologies, in the middle of the last decade, the *k*-mer centric de Bruijn graph has become the preeminent model for *de novo* genome assembly. Graph-theoretical approaches, the merging of unambiguous graph nodes up to repeat boundaries and path-finding heuristics are shared and salient characteristics of the main models of *de novo* genome assembly.

Comparative approaches to genome assembly have also been developed, but typically require a high degree of structural fidelity between the genome being sequenced and a reference genome. Consequently, repetitive regions and structural variations in the reference genome, such as inversions and rearrangements, limit the approach and may give rise to significant errors in an assembly. With the exception of the *Alignment-Layout-Consensus* model, existing comparative approaches do not directly integrate orthologous relationships into the contig assembly process, but use anchoring and alignment information in a *post hoc* scaffolding process.

The imminent availability of third generation sequencing platforms, capable of producing read lengths that greatly exceed those achieved by Sanger technologies, will require a radical reappraisal of algorithms and models of sequence alignment and genome assembly.
This chapter presents the methods and techniques employed to test, evaluate and validate the research questions addressed by the thesis. The rationale for the methods chosen was directed and informed by the corpus of established best practice discussed in Chapter 2. As the construction of a prototype application was required to test the research hypothesis, the methods described in this chapter are interleaved with and influenced by the philosophy, principles and practices that underpin agile software development methodologies [180]. Agile methodologies address the shortcomings of the traditional Waterfall [181] and Spiral [182] models through refactoring, continuous testing and integration over short development cycles [183, 184]. Regarded as lightweight practices by Ratkin [185] and the adherents of traditional process-oriented development methodologies [186], agile methods are based on the long established best practice of incremental and iterative software development [187].

Although the best practice discussed in Chapter 2 was applied, where appropriate, to each component of the research, a formulation of the scientific method known as experimental computer science was used as an overarching methodology. Experimental computer science is defined by Dodig-Crnkovic
as “most effective on problems that require complex software solutions.... The approach is largely to identify concepts that facilitate solutions to a problem and then evaluate the solutions through construction of prototype systems”. Such an applied approach was warranted by the degree of complexity involved in the design, construction and testing of a prototype capable of both sequence alignment and genome assembly.

The remainder of this chapter discusses the methods and techniques applied to each component of the research. In addition to establishing the rationale for the chosen methods, an examination of their implementation and limitations are presented and discussed.

3.1 Fuzzy Seed Methodology
The principal objective of this thesis is to establish the viability of approximate k-mer matching using fuzzy seeds. This section discusses the methodology used to design and construct fuzzy seeds and the techniques used to test and evaluate the approach.

3.1.1 Selection of Seed Model
A fundamental concept in object-oriented software development is the Open-Closed Principle (OCP), expounded by Martin [189]. The OCP requires that existing software artefacts are reused through extension, rather than modified or adapted to accommodate new features. The OCP was applied to the design of fuzzy seeds by extending the existing established models of consecutive and spaced seeds. Specifically, the k-mer matching techniques described by Kent [18] and Altschul et al [52] were directly incorporated into the design of consecutive fuzzy seeds. The fuzzy approach was also extended to include the spaced seed model described by Ma et al [55] and others [73-75, 100]. Support for multiple consecutive or spaced seeds was added to the fuzzy approach to address the limited specificity of single seeds identified by Li et al [54]. This application of the OCP is crucial to addressing the research
question, as it guaranteed that the basic assumptions of existing seed models were not violated by the addition of fuzzy behaviour.

The concept of combing fast exact-matching seeds with an approximate string-matching algorithm is based on the “seed and extend” strategy described by Pearson and Lipman [19], Altschul et al [52] and Strömberg [102]. The selection of approximate string-matching algorithms to use with the fuzzy model was informed by the detailed discussion of sequence alignment algorithms given by Gusfield [17] and by the string manipulation techniques described by Skiena [88].

The OCP was also applied to the prototype implementation of the fuzzy k-mer approach. Rather than modifying the internal mechanisms that control operations on dictionary data structures, the prototype extended existing software artefacts. This application of the OCP allowed the addition of fuzzy behaviours to a hash-based data structure without compromising the established time complexity of search and insertion operations proven and discussed by Cormen et al [50].

3.1.2 Testing and Benchmarking the Fuzzy Approach

The benchmarking and evaluation of the fuzzy k-mer model required the development of a mechanism to exercise the key features of the approach. A test-driven development methodology [190] was used to create a framework to measure the running time, sensitivity and specificity of fuzzy seeds, enabling an objective comparison with alternative techniques. By extracting synthetic sequence reads from completed genomes and recording the index position of each sequence, the test framework provided the a priori knowledge necessary to objectively and accurately assess the performance, key characteristics and limitations of the fuzzy approach. The framework also included the functionality required to parse the output generated by the prototype and the other aligners used in the study and to compute the metrics required for analysis. The sensitivity and specificity of alignments were
calculated using the same method applied by Brundo and Morgenstein [72] to measure alignment accuracy.

The genomes used in the evaluation were selected using the criteria of size, homology and the degree of repetitive sequence. The homology between query and subject sequences was calculated using the DDH method reported by Auch et al [154, 155]. Applying the agile practices of boundary value analysis and equivalence partitioning, described by Williams et al [191], genomes with very low, very high and medium DDH distances were chosen. The selection of repeat-rich genomes was guided by the results of studies by Haubold and Wiehe [60], Darling et al [192] and Achaz et al [193]. To evaluate the robustness of the approach with real biological data, sets of Illumina, 454 and Sanger sequence reads were downloaded from GenBank and included in the study.

The fuzzy approach for approximate k-mer matching unifies the “seed and extend” strategy into a single operation on a hash map. Consequently, the choice of candidate aligners to use in a comparative analysis was limited to k-mer based aligners that also implement the “seed and extend” strategy and are capable of processing both Sanger and SGS sequence reads. The single-seed BLAT aligner described by Kent [18], not only fulfilled all these criteria, but also provides a limited form of approximate string matching, by accommodating a single polymorphism at the end of a k-mer. For similar reasons, Mosaik [102], which employs a k-mer centric “spaced seed and extend” alignment strategy was also chosen for inclusion in the evaluation.

The test framework was also used to eliminate the effect of homological bias, by creating a set of randomly generated sequences when benchmarking the speed of the fuzzy approach. In addition to benchmarking the prototype against alternative aligners, the time complexity of the fuzzy approach was benchmarked against the guaranteed $O(\log(n))$ time of a tree map [194] and the average $O(1)$ time of a hash map [50] using conventional exact-matching seeds.
3.2 Comparative Assembly Methodology

This thesis also investigates how fuzzy k-mer alignments can be applied to the de Bruijn graph model of genome assembly, to enable the extension of contigs through the boundaries of repeat nodes in an assembly graph. This section presents the rationale for the assembly model chosen and the methods used to test and evaluate the integrated approach.

3.2.1 Selection of Assembly Model

Integrating fuzzy k-mer alignments directly into the contig assembly process necessitated a combination of comparative and de novo assembly models. The application of fuzzy k-mer alignments in the design and implementation of the prototype was based on the assisted assembly method described by Gnerre et al [63] and guided by the concepts and techniques discussed by Pop et al [68] in the Alignment-Layout-Consensus model. The extraction of unitig anchoring sequences from a de Bruijn graph used the “Elimination of Singletons” method described by Idury and Waterman [143].

Although any de novo model can potentially be employed in an integrated genome assembler, the selection of the de Bruijn graph model was warranted by the shared k-mer centric approach to fuzzy seed alignment. Moreover, as the fuzzy alignment technique employed in the prototype involves the decomposition of sequence reads into a tiling of k-mers, using a de Bruijn graph assembly model enabled both alignment and graph construction processes to be executed in parallel. Applying the Overlap-Layout-Consensus model [31] to such a context would require the unorthodox mixing of k-mer information with read nodes in an overlap graph. While k-mer centric techniques are heavily employed in the model, for detecting repeat sequences [69] and overlapping reads [34], the overlap graph does not provide any native k-mer support mechanism. An ab initio greedy assembly method was excluded from consideration, as the approach is prone to collapsing assemblies around repeats [124, 125], the identification of which is crucial for integrated assembly.
Although the de Bruijn graph approach represents a \( k \)-mer centric view of sequence fragment assembly, the model is flexible enough to accommodate different formulations of the assembly problem. The method described by both Pevzner \textit{et al} [78] and Idury and Waterman [143] recommend the representation of \( k \)-mers as edges in the graph and \( k \)-mer overlaps as nodes. Using this method, the sequence reconstruction problem can be formulated as finding a Eulerian path through the graph that visits each \( k \)-mer edge once. An alternative method, described by Simpson \textit{et al} [48] and Zerbino and Birney [49], represents \( k \)-mers as nodes in the graph and overlapping sequences as edges. This latter node-centric approach was used in the construction of the prototype, as it permits the representation of nodes in an adjacency list implementation of the graph and simplifies the application of anchoring heuristics during contig assembly. Furthermore, by modelling the adjacency list as a mapping of \( k \)-mers to nodes, complementary nodes are implicit in the graph as the reverse complement of a \( k \)-mer key.

The compact twin node implementation of a de Bruijn graph applied by Simpson \textit{et al} [48] and Zerbino and Birney [49] was not used in the prototype, as it limits the optimisation of node compression when applying the “Elimination of Singletons” method [143]. Furthermore, mate pair information was not directly incorporated into the graph, as suggested by Medvedev \textit{et al} [150], but supported implicitly by object references, enabling the exploitation of read threading techniques during contig assembly.

\subsection*{3.2.2 Assembly Testing and Evaluation Methodology}

The methodology used to assess the quality of contigs included the benchmarking of the prototype against the preeminent implementations of \textit{de novo} and comparative assembly, capable of utilising both Sanger and SGS sequence reads. As the \textit{de facto} standard implementation of the \textit{Alignment-Layout-Consensus} model, AMOS [68] was included in the evaluation to enable a direct comparison of hybrid and orthologous approaches to genome assembly. Velvet [49] was selected over other de Bruijn graph assemblers, as it also models \( k \)-mers as nodes in the assembly graph. Moreover, unlike the
majority of de Bruijn graph assemblers that are only suitable for use with short-length reads [27], Velvet can also process Sanger and mix-length sequences. The rationale for selecting Cabog [34] as representative of the Overlap-Layout-Consensus model is based on the ability of the assembler to process both Sanger and SGS sequence reads. Moreover, as the successor of the original Celera assembler [69], Cabog epitomises the original Overlap-Layout-Consensus model expounded by Kececioglu and Myers [31].

In addition to employing genomes used in previously published comparative studies [57, 192], the genomes utilised for testing, benchmarking and evaluation were selected on basis of their DDH distance [154, 155] and degree of structural fidelity to a reference genome. The Lander-Waterman model of shotgun sequencing and assembly [62] was incorporated into the test framework to facilitate the extraction of synthetic sequence reads of different lengths from the set of selected genomes. This is consistent with the method for comparing de novo assemblies applied by Salzberg et al [10] and the approach used by Medvedev et al [150] to assess the assembly characteristics of a paired de Bruijn graph.

The accurate measurement of assembly quality is limited by the complexity of the assembly process and the unique characteristics of each genome [111]. Although the de facto standard metric for assessing the contigs produced by an assembler, the limitations of the N50 value has, in recent times, resulted in the proposal of alternative measures of assembly quality. One such alternative, the Feature Response Curve [57, 178], was not included in the evaluation of the prototype, as it is designed for measuring de novo assemblies in the absence of a reference genome. The use of synthetic sequence reads, extracted from an existing complete genome, mandated the application of a method capable of accurately mapping assembled contigs to the original reference sequence. Consequently, the method described by Gurevich et al [179] and implemented in the QUAST tool was used to measure the quality of assemblies produced during the testing and benchmarking of the prototype. Furthermore, in addition to mapping assembled contigs to a reference genome, the set of metrics produced by QUAST also includes those used by Salzberg et al [10] in the GAGE study of assembly quality.
3.3 Chapter Summary

An experimental computer science methodology was employed to test, evaluate and validate the research questions addressed by this thesis. This necessitated the construction of a prototype application and its evaluation against the preeminent examples of similar k-mer centric aligners and genome assemblers. Each stage of the design, construction and testing of the prototype was informed and guided by the corpus of best practice discussed in Chapter 2. The test framework developed for the evaluation enabled an objective and accurate assessment of the fuzzy string matching approach. Similarly, the utilisation of third party software provided an independent and objective method for assessing assembly quality.
Current hash-based approaches to sequence alignment, that utilise the “seed and extend” or spaced-seed paradigms, are limited by the positional constraints of their $k$-mer models. Despite the ubiquity of $k$-mer techniques, the requirement for an exact match along some or all positions in a $k$-mer limits current approaches, necessitating a compromise between alignment speed, sensitivity and specificity.

This chapter discusses the hashing techniques that underpin current approaches to $k$-mer alignment and presents an alternative novel mechanism for $k$-mer matching. The fuzzy $k$-mer model combines the speed of hashing with the sensitivity of dynamic programming. The fuzzy $k$-mer model exploits the two-step mechanism used by hash maps to resolve collisions, allowing the two phases of the “seed and extend” approach to be unified into a single operation that executes in near constant time. This chapter also presents and discusses the design and implementation of the Ferox prototype, a sequence aligner and assembler based on the fuzzy $k$-mer model.
4.1 k-mer Matching with Hash Tables

Among the most important facets of k-mers, and one that is heavily exploited in sequence alignment, is their suitability as keys in a hash table or hash map. Hash tables and maps are dictionary data structures that use a key and a hashing function to provide rapid access to a collection of mapped values [51]. Hash maps achieve this level of rapidity by exploiting the $O(1)$ running time required to access an element at a known array index [88]. As hash maps are capable of quickly detecting exact matches between keys, they represent an ideal data structure for use in k-mer centric alignment and assembly applications. Consequently, as discussed in Chapter 2, hash-based k-mer matching techniques have been heavily exploited in bioinformatics and represent the preeminent underlying mechanism for k-mer matching employed for sequence alignment and genome assembly.

In a hash data structure, a hash key, $S$, is used to functionally determine a mapped value. Keys are mapped to a linear array, $T[0..m – 1]$, called a hash map, by a surjective hash function $h:S \rightarrow \{0..m – 1\}$ [195]. The array index position is known as a bucket and acts as a container for key-value pairs [51]. The implication of using a hashing function in this manner is that, although redundancy is permitted among the values in a hash map, the hash keys must be unique. While this uniqueness requirement provides hash structures with the underlying property to facilitate speed, it constrains access to the exact matches of keys. For k-mer centric applications, this renders conventional hash data structures intolerant of variations in sequence composition, such as the polymorphisms, insertions and deletions common in biological sequences. Furthermore, the absence of injectiveness in the hash function results in address collisions in a hash map, where different keys map to the same array index.

4.1.1 Hashing and k-mers

In the context of biological sequence data, a k-mer represents a string of characters from an alphabet of nucleotide or protein symbols. Using k-mers as
keys in a hash-based dictionary structure therefore requires some mechanism to compute a numeric hash representation from the set of characters in a string. Cormen *et al* [50] describes a good hash function as one that approximately satisfies the assumption of simple uniform hashing, i.e. that each key is equally likely to hash to any of the $n$ buckets in a hash map, independent of where any other key has hashed to. A hashing function that satisfies this property thus generates an independent and uniform distribution of keys over the set of buckets in a hash map.

**Fig 4.1.** A hash table created from a set of sequence reads. Locating a bucket index from a hash key requires computing an integer value from the key and then transforming that value into a bucket index.

For strings therefore, a good hashing function should take cognisance not only of the numeric representation of each character, but also its index position in the sequence of characters. This requirement is particularly cogent for strings comprised of nucleotide symbols, where the alphabet size is very small.

Determining a bucket index from a hash key is a two-step operation [88], involving the computation of an integer value from an arbitrary key type and then transforming that value into an index in the range $[0..m - 1]$, where $m$ is the number of buckets in the hash map. In the Java language [196], a numeric
hash value is computed from a string of length \( n \) using the 16-bit Unicode value of each character at each index position in the string:

\[
h(S) = \sum_{i=0}^{n-1} \text{char}(S_i) \cdot 31^{n-(i+1)}
\]

Eq.(4.1) returns a 32-bit signed integer that can be translated into a bucket index of a hash map \( T \) using the division method [50], i.e. \( T[S] = h(S) \mod m \).

In contrast with procedural programming approaches, object-oriented languages, such as Java, allow arbitrary types to be used as keys and values in a hash map [194]. Consequently, some mechanism must be provided to allow the computation of a hash integer that exploits the semantics of the arbitrary type used. In Java, this functionality is encapsulated in a method called \( \text{hashCode()}, \) inherited from the proto-generic type \( \text{java.lang.Object} \). While the native inherited implementation of \( \text{hashCode()} \) is sufficient for most purposes, the method can be overridden and adapted to the semantics of a particular type. In the context of strings, the default implementation of \( \text{hashCode()} \) is overridden to compute an integer representation of a string using the transformation shown in Eq.(4.1). While Eq.(4.1) provides a good approximation of simple uniform hashing, any variation in sequence composition will yield a different integer representation of the string and consequently, a different bucket index. For this reason, insertion, deletion and search operations on a hash map using a \( k \)-mer key are limited to exact matches, with no native mechanism available to allow for approximations.

### 4.1.2 Collision Resolution in Hash Maps

Given a uniform and random distribution of keys in a hash map, operations for insertion, search and deletion execute in \( O(1) \) average time [50]. In a chained hash map, a hashing function is used to compute a bucket index from a search key, with each bucket index containing a pointer to a linked list. During an insertion operation, a collision occurs if two keys hash to the same bucket index. Although a good hashing function will minimise their occurrence, collisions can be resolved by adding the new key to the head of the linked list at a bucket index. A search of a hash map operates in a similar
manner, with the hashing function providing \( O(1) \) time access to a bucket index, followed by the time it takes to search the linked list at that index for a hash key.

Constructing a hash map of \( k \)-mers, from a set of shotgun reads or a reference genome, typically involves tiling a sliding window of size \( k \) over each sequence, extracting the substring of consecutive \( k \) characters into a \( k \)-mer string and then adding each \( k \)-mer to the map. For a genome \( G \), the set of \( k \)-mers or the \( k \)-spectrum, can be defined as the set \( G^k = \{ G[i : i + k - 1] \mid 0 \leq i < |G| - k + 1 \} \), where \( G[i : i + k - 1] \) denotes, for a constant \( k \), the substring of consecutive characters in \( G \) from index \( i \) to index \( i + k - 1 \). A hash map, \( T \), can be constructed from \( G \) by inserting each \( G[i : i + k - 1] \) in \( G^k \) as a key in \( T \), along with its positional information as a value.

![Bucket 0](https://via.placeholder.com/150)

![Bucket 1](https://via.placeholder.com/150)

![Bucket 2](https://via.placeholder.com/150)

![Bucket 3](https://via.placeholder.com/150)

![Bucket 4](https://via.placeholder.com/150)

![Bucket 5](https://via.placeholder.com/150)

![Bucket 6](https://via.placeholder.com/150)

**Fig 4.2.** A hash table created from a set of 8-mers. Each bucket index contains a linked list to which \( k \)-mers are added as hash keys. Each \( k \)-mer maps to some satellite data, in this case an index position in a reference genome. At buckets 0 and 4, a hash collision has resulted in more than one key being added to the bucket’s linked list.
The average number of $k$-mers stored in the linked list at an index in $T$ is known as the load factor $\alpha = \#G^k/m$, where $m$ is the capacity or number of buckets in the hash map. In a conventional hash map, under the assumption of simple uniform hashing, a total of $\Theta(1 + \alpha)$ time is required for an insertion or search operation [50]. Clearly therefore, a search at a bucket index that is not empty will require a minimum of two tests for equality – one test based on the hash of a search key and one or more tests against the existing keys in the linked list.

While the mechanics for manipulating collision detection in hash structures vary between programming languages, collisions are resolved in Java by the semantics of object equality, as implemented in the `hashCode()` and `equals()` methods [196]. In common with `hashCode()`, the `equals()` method is inherited from the proto-generic type `java.lang.Object` and provides, through overriding, a flexible mechanism for defining the semantics of object equality for any given type. When searching a hash map for a given key, if two `hashCode()` methods return the same integer, a collision is detected if the linked list at that bucket index is not empty. The collision is then resolved by executing the `equals()` method of the search key against each key in the linked list.

The semantics of object equality depend on the context in which an object is used and forms part of the design work for a class. In general, objects that are equal according to their `equals()` method must share the same hash code. The corollary is not necessarily true. Although the default implementation of `equals()` returns true only when two objects share the same object ID, it is sometimes desirable to relax or refine the definition and allow some scope for variability. The manipulation of the contract between these two methods is the fundamental mechanism employed to implement the fuzzy $k$-mer model.

4.2 The Fuzzy $k$-mer Model

The fuzzy $k$-mer approach is based on the work of Topac [197], who showed how fuzzy operations can be performed on a hash map by manipulating the
two-step mechanism required to search a hash map with a key. Instead of attempting to design a hashing function to avoid possible collisions, controlled collisions are deliberately induced in the hash map by allowing only part of the search key to be used by the hashing function and permitting a degree of variability in the remainder of the key.

Unlike conventional maps, that provide a surjective mapping of keys to values, fuzzy hash maps relate fuzzy hash keys to fuzzy sets, the latter of which are characterised by a fuzzy set membership function $\mu()$. For a $k$-mer $S \in G^k$, a fuzzy hash map can be defined as $M:f(S) \rightarrow A$, where $f(S)$ is a fuzzy hash key operation on $S$ and $A$ is a fuzzy set with a membership function of $\mu_A(S)$. In contrast with the dichotomy implicit in Boolean operations on crisp sets, fuzzy sets are characterised by the continuum of real numbers between 0 and 1 that describe the degree of set membership [198]. For a fuzzy hash map $M$, the degree of membership of the fuzzy set $A$ is governed by the fuzzy set membership function, $\mu_A(S) \rightarrow [0..1]$.

A fuzzy $k$-mer can be defined as $f(S) = h(S) \land \mu_A(S)$, where $h(S)$ is the subsequence of $S$ used to compute a hash code and $\mu_A(S)$ the substring that permits a degree of variability. The function $h(S)$ maps a fuzzy hash key $f(S)$ to the linked list of keys at a bucket index, with the membership function
\( \mu_A(S) \) used to evaluate the degree of membership of the fuzzy set associated with each of the keys.

Using a conventional hashing algorithm, such as Eq.(4.1), in \( \text{hashCode}() \) on part of a \( k \)-mer, the degree of key similarity determined by \( \mu_A(S) \) can be encapsulated inside the \( \text{equals}() \) method, which can utilise any sequence similarity algorithm capable of returning a fuzzy value in the interval [0..1]. For example, given the Damerau-Levenshtein distance \( d \) computed using a match score of +1 and a mismatch score of 0, the membership function \( \mu_A(S) = 1 - d/k \) will return a fuzzy measure of similarity for a \( k \)-mer search key \( S \) and a fuzzy set \( A \). The manipulation of the contract between \( \text{hashCode}() \) and \( \text{equals}() \) has the effect of grouping together approximately matching \( k \)-mers into a single fuzzy set, allowing the fuzzy set of \( k \)-mers to be accessed in a single search operation. Any edit distance metric, such as Hamming distance [199], Levenshtein distance [13] or Smith-Waterman score [15] can be employed to implement the fuzzy set membership function, providing flexible alternative mechanisms for designing and configuring a fuzzy \( k \)-mer.

Consistent with the functionality of a regular hash map, a collision will occur during an insertion or search operation in a fuzzy hash map \( M \) when the number of elements in the fuzzy set belonging to a hash key > 0, i.e.

\[
\exists A \in M[h(S)] \mid \#A > 0
\]  

(4.2)

A further constraint can be applied to the fuzzy set membership function that limits membership to \( k \)-mers that match above a given threshold and eschews the possibility of a \( k \)-mer being added to the fuzzy set \( A \) with a membership value of zero. Such a constraint is analogous to the application of hedges to a fuzzy set described by Negnevitsky [200] and by Tanaka and Niimura [198]. If the fuzzy set membership function \( \mu_A(S) \) is constrained with a cut-off threshold \( \beta \), a set of matches will be detected during an insertion or a search operation where:

\[
\exists A \in M[h(S)] \mid \mu_A(S) \rightarrow [0, 1], \beta \leq \mu_A(S) \leq 1
\]  

(4.3)
Crucially, no change to the underlying implementation of the regular hash map is required to accommodate the fuzzy behaviour. This enables the fuzzy approach to allow variability in hash keys, without sacrificing the $O(1)$ average running time of the basic map operations. As an insertion at the head of the fuzzy set $A$ is a constant time operation, assuming a dynamic programming algorithm is used to compute $\mu_A(S)$, the total running time for an insertion is $\Theta(1 + \alpha(|S|^2) + 1)$ and $\Theta(1 + \alpha(|S|^2))$ for a search operation. Using this technique, the number of hash collisions is controlled by the number of indices in $S$ used by $h(S)$, with the number of fuzzy sets in the linked list at each bucket controlled by the $\beta$ cut-off threshold. Once a fuzzy hash map has been constructed from $G^k$, the running time required to align a tiling of $k$-mers from a query sequence will be proportional to the length of the query sequence and the size of $k$, but will remain constant with respect to the size of the map. Note that the hashing indices in $S$ do not have to be consecutive, in which case, $h(S)$ is logically equivalent to the spaced seeds developed by Ma et al [55] and subsequently refined and improved by Li et al [54] and Lin et al [56].

4.3 Implementing the Fuzzy $k$-mer Model

The fuzzy $k$-mer model has been implemented in a prototype sequence aligner and assembler called Ferox that exploits its support for approximate $k$-mer matching to align both Sanger and SGS sequence reads against a reference genome. Ferox is also capable of aligning whole genomes and can be used to create synteny plots. The remainder of this chapter describes the design and functionality of the alignment module of the Ferox prototype and details how the fuzzy $k$-mer model can be employed to create a sequence aligner that unifies the speed of hashing with the sensitivity of dynamic programming.

4.3.1 Seed Design and Implementation

The prototype extends established approaches to $k$-mer seeding by exploiting the variability inherent in the fuzzy $k$-mer model. At an architectural level, the
design requires the decomposition of a reference genome into a tiling of fuzzy $k$-mers that are subsequently added to a hash map. The query sequence of shotgun reads or a complete genome is similarly fragmented into a set of overlapping $k$-mers, which are then aligned to the reference sequence by searching a hash map and processing the results. This general approach to $k$-mer alignment is similar to those used by Green [126], Sutton et al [130] and Mullikin et al [129] for genome assembly and by Altschul et al [53], Kent [18] and Ma et al [55] for sequence alignment.

![UML class diagram](image)

**Fig. 4.4.** A UML class diagram, showing the structural relationships between the classes used to implement fuzzy $k$-mer matching. The application of interfaces and composition in the design promotes a high degree of extensibility and reuse.

At a class level, the prototype has been designed to be robust, flexible and extensible by applying the principles of high cohesion and loose coupling in a highly modularised design. A particular emphasis has been placed on ensuring that essential components of the design are defined and referenced
as abstractions, enabling the widespread application of polymorphism\(^1\) throughout the application.

The UML class design in Fig. 4.4 depicts the fuzzy \(k\)-mer module of classes used by the prototype. The design abstracts the minimal set of behaviours associated with a fuzzy \(k\)-mer into an interface called *FuzzySeed*. The set of behaviours includes the ability to parse and store the hash and fuzzy components of a user-defined \(k\)-mer, along with service methods for stipulating an approximate string-matching algorithm. The interface also provides methods for specifying and accessing the \(\beta\) cut-off threshold for controlling the match sensitivity of a fuzzy \(k\)-mer alignment.

Fig. 4.5. Fuzzy seeds can be defined using consecutive or spaced indices to define “must match” or “don’t care” positions. The degree of similarity in “may care” fuzzy indices is computed using an approximate string-matching algorithm.

The behaviours exposed by *FuzzySeed* are implemented in the abstract class *AbstractFuzzySeed* and in the derived type *DefaultFuzzySeed*. The former

\[^1\] Polymorphism is a salient and essential concept in the object-oriented paradigm. Polymorphism, in this context, refers to the application of the Liskov Substitution Principle [201], where a supertype can be dynamically substituted by a subtype, using either implementation or specification inheritance, without altering or compromising the robustness of a design. As polymorphism is an equally significant concept in molecular biology, its use in the context of object-object design will be denoted by rendering the term in italics throughout this thesis.
encapsulates the mechanism for computing a hash integer from a $k$-mer, but
delegates the approximate string-matching responsibility to subtypes of the
interface $KeyComparator$. The class $DefaultFuzzySeed$ is a concrete
implementation of $FuzzySeed$ that pre-processes the indices in a seed that
permit a degree of variability. The polymorphic composition relationship
between $AbstractFuzzySeed$ and $KeyComparator$ is essential, as it promotes
the loose-coupling necessary to allow instances of $FuzzySeed$ to be
configured with any approximate string-matching algorithm. Its application in
Ferox embodies the Strategy Pattern described by Gamma et al [202],
allowing the implementation of the matching algorithm used to vary
independently of the $FuzzySeed$ that uses it. It is notable that subtypes of
$KeyComparator$ are the mechanism used by the Ferox prototype to implement
the fuzzy set membership function, $\mu_A(S)$, required by the fuzzy $k$-mer model.

The $KeyComparator$ interface specifies a single method for computing an
approximate match between two strings:

$$\text{float fuzzyCompare(String s, String t);}$$

Notwithstanding the 32-bit floating point return type defined by the method,
the value returned must be a fuzzy real number in the interval $[0..1]$
Applying post-conditions to a method invocation in this manner is consistent
with the principles of Design by Contract developed by Meyer [203] and
upholds the Principle of Substitutability described by Liskov and Wing [201].
Moreover, defining the string matching behaviour in this manner allows any
edit distance metric to be used to implement the fuzzy set membership
function.

A number of derived implementations of $KeyComparator$ are available in the
prototype. The class $FuzzyHammingDistance$ provides a modified version of
Hamming Distance [199] that computes fuzzy set membership based on the
number of substitutions required to transform one string into another. Despite
having a time complexity of $O(n)$, Hamming Distance is a jejune approach for
approximate $k$-mer matching, as the algorithm applies a punitive edit distance
in the presence of insertions, deletions and transpositions. More refined
measures of edit distance, based on dynamic programming, such as
Levenshtein [13] and Damerau–Levenshtein Distance [204] can better accommodate the variability found in biological sequences, albeit with a quadratic space and time complexity. In particular, the Levenshtein algorithm is ideally suited to $k$-mer sequences as it is capable of accommodating polymorphisms, insertions and deletions. The global alignment algorithms employed by Ferox, calculate string similarity based on a measure of edit distance and compute the membership function $\mu_A(S) = 1 - d/k$, where $d$ is the edit distance between two $k$-mers.

An implementation of the Smith-Waterman [15] algorithm for local sequence alignment is also included. The implementation computes $\mu_A(S) = \text{score} / k * 2$, using a match score of +2, a mismatch score of -1 and a gap penalty of -1. As the maximum score in the Smith-Waterman dynamic programming matrix is sufficient for the computation of a fuzzy measure of string similarity, the trace-back step of the algorithm is omitted, providing a slight decrease in running time.

The rapidity of fuzzy $k$-mer matching is predicated on the successful manipulation of the two-step collision resolution mechanism employed by hash maps. While this manipulation is exemplified in a fuzzy hash map, the underlying implementation proposed by Topac [197] violates the Principle of Substitutability [201] by adding fuzzy behaviour to crisp methods of the class `java.util.HashMap` through direct inheritance. Although the fuzzy hash map implementation employed in the prototype also reuses `java.util.HashMap`, it does so through composition and delegation, thus upholding the Principle of Substitutability, allowing any valid instance of `java.util.Map` to be used by the aligner.

The salient characteristic of a fuzzy hash map is the use of a fuzzy hash key to encourage controlled collisions during insertion, deletion and search operations. The implementation used by the Ferox prototype defines a fuzzy hash map $M$ as a mapping between instances of the interface `FuzzyHashKey` and a fuzzy set $S$, implemented as a linked list, i.e. $M:FuzzyHashKey \rightarrow S$. While operations on the underlying conventional hash map utilised by the prototype are governed by a surjective mapping, an injective function is
employed by the class *FuzzyHashMap* to relate each fuzzy hash key with a unique fuzzy set.

The *FuzzyHashKey* interface abstracts the minimal set of behaviours required for fuzzy operations on a hash map. Derived types are responsible for manipulating the contract between the *hashCode()* and *equals()* methods, along with providing additional services methods for specifying the type of *FuzzySeed* to use. The design enables instances of *FuzzyHashKey* to delegate the implementation of *hashCode()* and *equals()* to any instance of *FuzzySeed*, enabling Ferox to dynamically bind the logic for controlling collisions in a fuzzy hash map at run-time using *polymorphism*. To reduce the memory overhead associated with the composition relationship between a *FuzzyHashKey* and a *FuzzySeed*, the latter is designed as a flyweight [205], separating the intrinsic state of a *FuzzyHashKey* from the extrinsic shared state of a *FuzzySeed*.

The implementation of the class *FuzzyHashMap* provides methods and constructors that allow the map to be parameterised with a single flyweight fuzzy seed. In contrast with the approach used by Topac [197], only fuzzy behaviours are exposed by the fuzzy hash map. As deletion operations are not required for sequence alignment, the following salient fuzzy methods, defined by the fuzzy hash map, relate to insert and search operations respectively:

```java
void put(K key, V value)
List<V> getFuzzy(K fuzzyKey)
```

Both methods are parameterised with generic types enabling the fuzzy hash map to be configured with any instance of *FuzzyHashKey*. The *put(K key, V value)* method creates an injective mapping between a key-value pair, by adding the generic type *V* to the fuzzy set associated with each *FuzzyHashKey*. Significantly, the fuzzy set is implemented as a linked list, providing *O(1)* running time for insertions and deletions at either end of the list [50].

As the map is being constructed from the *k*-spectrum of a reference genome, a *FuzzyHashKey* is created for each *k*-mer. The *FuzzyHashKey*, along with anchoring information about the position of its *k*-mer in the reference genome,
is passed as a parameter to the `put(K key, V value)` method. If a hash collision occurs in the map during insertion, the `equals()` method belonging to the instance of `FuzzyHashKey` will be executed, resulting in the invocation of the approximate string-matching algorithm associated with the `FuzzySeed`. If the `equals()` method returns true, the anchoring information is added to the head of the linked list implementation of the fuzzy set. This operation is guaranteed to execute in $O(1)$ time [50, 51]. If no matching `FuzzyHashKey` is found at the bucket index during insertion, a new fuzzy set, containing the anchoring information associated with the $k$-mer, is created and mapped to the key. Using this approach, all $k$-mers from the $k$-spectrum of a reference genome that share the same hash integer and similarity above the $\beta$ cut-off threshold will become elements of the same fuzzy set.

A fuzzy search capability is provided in the fuzzy hash map through the implementation of the `getFuzzy(K fuzzyKey)` method. For a given instance of `FuzzyHashKey`, passed in as a parameter, the method returns the fuzzy set associated by the injective mapping of fuzzy hash keys to fuzzy sets. The mechanics of the implementation are consistent with that employed during insertion, where a hash code collision is resolved by executing the `equals()` method against each fuzzy hash key at a bucket index. Notably, because the fuzzy hash map delegates the underlying collision resolution mechanism to `java.util.HashMap`, the average $O(1)$ running time of a search operation is retained.

To increase the degree of sensitivity in the implementation of the fuzzy $k$-mer model, Ferox can be configured with multiple fuzzy seeds. Li et al [54] have shown this approach to be highly effective in increasing the sensitivity and specificity of alignments using spaced seeds. Consistent with the techniques for supporting multiple spaced seeds reported by Rumble et al [75], Homer et al [99] and Lin et al [56], the implementation used by the prototype creates a separate fuzzy hash map for each seed. Although this approach increases the overall memory complexity of Ferox, the anchoring information associated with each $k$-mer is shared between the different fuzzy hash maps.
4.3.2 Configuring Fuzzy k-mers

To facilitate extensibility and flexibility, the salient aspects of the fuzzy k-mer model are encapsulated in the design of the Ferox prototype, by enabling user-defined fuzzy seeds and implementation classes in a configuration file. Essential properties, such as k-mer size and the implementation classes of FuzzySeed and FuzzyHashKey are defined declaratively. The prototype parses the XML declarations and uses dynamic class loading to read the defined classes into the heap space of the Java Virtual Machine at run-time. This embodies the application of the Open-Closed Principle, permitting the extension of the API exposed by the prototype through specification inheritance.

Ferox allows seeds to be defined declaratively in a configuration file by specifying a seed sequence, a string matching algorithm and the β cut-off threshold for matches. A FuzzySeed can be configured with any approximate string-matching algorithm by specifying the instance of KeyComparator to use. As illustrated in Fig. 4.6, the 24-mer seed #((((((((((((*************)#))###)), which can be described as $f(k) = 11h + 13f$ for convenience, computes a bucket index from the first 11 characters in a 24-mer and uses the remaining 13 characters to permit a degree of variability. Specifying a β cut-off threshold of 0.53 for this seed will allow up to six mismatches using the Damerau–Levenshtein algorithm.

The hash indices of seeds are not required to be consecutive. The second seed declaration of ####### in Fig. 4.5 specifies a hash integer using the 6 prefix and 5 suffix characters of a 24-mer and computes the degree of fuzzy similarity from the middle 13 characters. The prototype can be configured with multiple fuzzy seeds for a given k-mer size, but allows different algorithms and β cut-off thresholds for different seeds.

The class DefaultFuzzySeed has been designed to also accommodate the use of spaced-seeds. A spaced-seed is a binary string that represents a set of non-consecutive indices in a sequence, where 1’s denote a hash index and 0’s indicate a “don’t care” position. In a seminal publication, Ma et al. [55] noted that the employment of spaced-seeds in a homology search leads to
surprisingly higher sensitivity and speed over conventional consecutive seeds. The superiority of the spaced-seed model was proven by Li et al [207], who also showed that computing the hit probability of a spaced-seed in a uniform region is NP-hard.

Fig. 4.6. Configuration file for the Ferox prototype.
Despite providing a marked increase in sensitivity, as noted by Ilie [95], decreasing the number of hash positions in a spaced-seed also lowers the specificity of a homology search. The challenge of increasing specificity led to the development of multiple spaced-seeds by Li et al [54], who showed that increasing the number of seeds acts as a counter-balance to a reduced number of hash indices, simultaneously increasing sensitivity without reducing specificity. Further studies of the spaced-seed model by Ilie [95], Choi and Zang [104] and Rasmussen et al [90] describe methods for identifying and computing optimal and efficient seeds.

The type DefaultFuzzySeed provides orthogonal support for spaced-seeds, allowing the seed pattern to be defined declaratively in the same manner as a conventional seed. For the 18-mer spaced-seed, ###-#-##-#-##-##, used in the original PatternHunter [55] aligner, the hashCode() and equals() methods will compute an integer and a Boolean value respectively from an exact match of the specified hash indices. The pattern-parsing mechanism implemented in the prototype requires the use of a hash symbol (#) to denote an exact match position and an asterisk to indicate a fuzzy index. Indices in a seed pattern containing any other symbol will be ignored and are equivalent to “don’t care” positions in contemporary spaced-seeds. Using a spaced-seed in this manner implies that all k-mers that share the same characters at the set of hash indices will also share membership of the same fuzzy set.

The fuzzy k-mer approach can also be used to extend the spaced-seed model. Ferox permits fuzzy spaced-seeds to be declared, allowing part of a spaced-seed to be used to compute the bucket index in a hash map, allowing a degree of variability in the remainder of the seed. For example, the 33-mer fuzzy spaced-seed -###-##-##-##-##-##-#************* contains 11 hash positions in the first 20 indices and a further 13 positions used to compute $\mu_A(S)$. In this formulation of a fuzzy k-mer, the spaced-seed contains both “don’t care” and “may care” indices, the latter offering the potential to redress the problem of low specificity in a single spaced-seed [54]. For larger values of $k$, this is effectively a “spaced-seed and extend” approach, combining the two phases of alignment into a single unified operation.
4.3.3 Aligning $k$-mers and Reifying Matches

The prototype aligns two sequences by first generating a tiling of $k$-mers from a reference genome and then inserting each $k$-mer into a fuzzy hash map, along with its positional information, using the specified fuzzy seeds.

1) **Sequence:** AAGTGGATACGCACCAATTACGGCGATGT

2) **Decompose into Overlapping $k$-mers ($8$-mers):**

   - AAGTGGAT
   - AGTGATAT
   - GTTGGATA
   - CACCGAGG
   - GATACTAC
   - TTACGGCA
   - GATAAGCT

3) **Align $k$-mer in +/- Orientations:** Set results = getFuzzy("AAGTGGAT") = \{a_1, a_2, a_3, a_{10}, a_{11}\}

   ![Diagram](image)

   - getFuzzy("AAGTGGAT") = \{a_1, a_2, a_3\}
   - getFuzzy("AAGTGGAT") = \{a_{10}, a_{11}\}

Fuzzy Seed: #ÿ##**, β = 0.75

<table>
<thead>
<tr>
<th>FuzzyHashKey</th>
<th>Set&lt;k-mer&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAGT****</td>
<td>a_1[1], a_2[1], a_3[1]</td>
</tr>
<tr>
<td>TTAC*****</td>
<td>a_4[1], a_5[1]</td>
</tr>
</tbody>
</table>

Fuzzy Seed: ##****#, β = 0.5

<table>
<thead>
<tr>
<th>FuzzyHashKey</th>
<th>Set&lt;k-mer&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>CT****CA</td>
<td>a_2[1]</td>
</tr>
<tr>
<td>AA****AT</td>
<td>a_3[1]</td>
</tr>
</tbody>
</table>

4) **Group $k$-mer Alignments:** Filter overlapping or adjacent alignments into groups.

5) **Score Alignments:** For each group $S$ in the set of results compute \[ \text{score} = \frac{\sum_{i=1}^{n} S_i}{n} \leq 1 \]

   - ![Alignment Window](image)

   \[ \text{score} = \frac{27}{30} = 0.9 \]

6) **Report Alignments:** Report alignments where score ≥ alignment threshold.

Fig. 4.7. Overview of fuzzy $k$-mer alignment in the Ferox prototype. A separate fuzzy hash map is created for each fuzzy seed. Each hash map relates a fuzzy $k$-mer to a fuzzy set of anchor points in a reference genome.
As $k$-mers are added to the hash map, collisions are resolved using the
`equals()` method, resulting in approximately matching $k$-mers being added
to the fuzzy set associated with a matching seed. Consequently, as illustrated in
Fig. 4.7, the constructed fuzzy hash map will consist of a set of fuzzy hash
keys that map to fuzzy sets of index positions in a reference genome.

After the $k$-spectrum of anchor points have been hashed into a map, the
alignment process commences by parsing the set of query sequences from a
FASTA file and offering each sequence to a blocking queue [194]. A separate
thread polls each sequence from the head of queue and creates its dyadic
complement using a modified version of object cloning. Each read is then
aligned by decomposing it into a tiling of $k$-mers and then searching the hash
map of fuzzy $k$-mers for approximate matches, using each $k$-mer in a read as a
search key. Regardless of the multiplicity of fuzzy seeds used, the fuzzy $k$-
mer alignment process returns a set of unique matches, with duplicate
positional information eliminated by manipulating the semantics of object
equality in matching anchor points.

Once the result set of a read alignment has been created, the set of matching
anchor points are further processed to remove spurious or weak matches. An
alignment window is applied to the set of matches for each read with an
anchor sequence. The alignment window imposes distance constraints,
computed from the alignment indices, the read length and the length of the
anchor sequence. This has the effect of grouping together both overlapping
and non-overlapping $k$-mer matches that fall within the alignment window.
As depicted in Fig. 4.7, each of these groups is represented as a binary array,
with 1’s corresponding to matching positions and 0’s to mismatches. For each
alignment group, $S$, an alignment score is computed as:

$$\text{score} = \frac{\sum_{i=1}^{n} S_i}{n} \leq 1$$

(4.4)

After an alignment group has been scored, a user-defined minimum alignment
threshold constraint is then applied, resulting in only those alignment groups
at or above the threshold being reported.
The minimum alignment threshold is a fuzzy value in the interval \([0..1]\) and can be specified in the configuration file as the attribute `minimum-alignment-match`. This threshold value applies a percentage identity constraint to the entire sequence read and controls the number of reported matches that remain after filtering and scoring. If the `-genome` switch is specified when running the aligner, this filtering process is omitted, as whole genome alignment mandates the reporting of all \(k\)-mer matches. In the context of anchoring a sequence against a reference genome, applying the `-best` switch will result in only the top scoring alignment being reported.

As each sequence read is parsed, aligned, processed and scored, matches are reported by formatting the output in BED format. Where the `-genome` switch is specified, an optional `-mums` switch can also be used to specify an output
format compatible with MUMmer [16, 80, 81]. This enables synteny plots to be created from whole genome alignments using the \textit{mummerplot} tool.

\textbf{4.4 Chapter Summary}

The fuzzy $k$-mer model combines the speed of hashing with the sensitivity of dynamic programming, by exploiting the two-step collision resolution mechanism employed by hash maps. The model is highly flexible, permitting multiple user-defined seeds and configuration with any approximate string-matching algorithm. The model has been implemented in a prototype aligner that unifies the more salient aspects of both the “seed and extend” and spaced-seed models.
Chapter 5

Integrated Assembly with Fuzzy $k$-mers

The de novo models of genome assembly, discussed in Chapter 2, are characterised by the identification of repetitive sequences and the merging of unambiguous nodes in a graph to the edges of repeat boundaries [69, 143]. In both the Overlap-Layout-Consensus and de Bruijn graph models, read length constrains contig assembly by placing limits on the ability of an assembler to resolve repeat-induced conflicts by circumnavigating repetitive regions during graph transversal [4]. Consequently, a separate scaffolding phase, involving the application of paired-read and other constraints, is typically used to resolve the ambiguity induced by repeats and to direct the merging of unitigs.

Comparative assembly models attempt to resolve the conflicts effectuated by repeats by utilising the nucleotide or protein sequences of a related species to guide the contig assembly process. The Alignment-Layout-Consensus model, employed in AMOS [68] and Mosaik [102], eschews the overlap phase of assembly completely, relying on alignment and anchoring techniques to provide positional information to the layout phase. While this approach
directly integrates anchoring information into the contig assembly process, it requires a high degree of structural fidelity between the genome being assembled and the reference sequence.

The alternative *post hoc* comparative assembly techniques described by Gnerre *et al* [63], Reinhardt *et al* [166] and others [167, 169, 170] use *de novo* assemblers to generate contigs and scaffolds, which are subsequently reified by their alignment against a reference genome. Such approaches to comparative assembly do not integrate anchoring or alignment information directly into the *de novo* contig assembly process and have no native mechanism for identifying and circumventing repeat boundaries.

This chapter describes how fuzzy $k$-mer alignment can be applied to the de Bruijn graph model of genome assembly, enabling the assembly of contigs up to and through the boundaries of repeats using anchoring information. In contrast with existing techniques, the anchoring information provided by the fuzzy $k$-mer model is directly integrated into the contig assembly process.

The techniques presented in this chapter have been implemented as additional modules in the Ferox prototype and complement the fuzzy $k$-mer aligner discussed in Chapter 4. The structure of this chapter follows the assembly pipeline of the implementation, reflecting the discrete and sequential phases of the overall assembly process.

**5.1 Guided Assembly Pipeline**

The assembly of a set of shotgun sequence reads into a collection of contigs is both a multifaceted and complex process. The implementations of the principal phases of the assembly pipeline, depicted in Fig. 5.1, have been heavily influenced by the published contributions of a number of preeminent authors of $k$-mer models of genome assembly and ancillary techniques. In contrast with existing models of *de novo* assembly, the comparative approach described in this chapter is native to the assembler and is heavily exploited as an integral part of the overall assembly process.
Fig 5.1. Assembly Pipeline. Parsed sequences are aligned and added to the graph in parallel. After compressing the de Bruijn graph into a sequence graph, contigs are assembled and then scaffolded. Anchoring information, given by a fuzzy k-mer alignment, is incorporated into both the contig assembly and scaffolding phases.

At a general level, the set of parsed sequence reads are aligned and added to the de Bruijn graph in parallel. Parallelising these discrete sub-processes is both intuitive and warranted by the shared k-mer nature of both the assembly graph and the alignment model. The node-centric implementation of the de Bruijn graph is based on the models described by Zerbino and Birney [49] and Simpson et al [48], with the transformation to a sequence graph heavily influenced by the original formulation of a k-mer graph described by Idury and Waterman [143].

Although the processes of generating a collection of contigs from a sequence graph employs the concepts of read threading described by Chaisson and Pevzner [29], the anchoring information yielded by the fuzzy k-mer alignment is integrated directly into the graph transversal algorithm and is a salient and novel aspect of genome assembly in the prototype. The set of assembled contigs is scaffolded using techniques adapted from both Gnerre et al [63]
and Saltzberg et al [156], enabling the assembler to output information on the proximity relationships between contigs to supplement the conventional output provided as a FASTA file. While the comparative approach to assembly, described in this chapter, shares certain characteristics with the Alignment-Layout-Consensus model expounded by Pop et al [68] and Strömberg [102], the totality of the assembly process is dominated more by de novo models of genome assembly and by k-mer centric alignment and assembly techniques.

The remainder of this chapter presents and discusses the techniques used to implement each of the assembly phases in more detail and places an emphasis on the centrality of both the fuzzy k-mer model and k-mer techniques and methods.

### 5.2 de Bruijn Graph Model

A k-mer de Bruijn graph, \( G = (V, E) \), on an alphabet \( \Sigma \) can be defined as \( V = \Sigma^k, E = \{(u, v) \mid u, v \in V \text{ and } u_{i+1} = v_i, \forall \ i \mid 0 \leq i < k - 1\} \). A k-mer de Bruijn graph has \(| \Sigma|^k\) nodes and \(| \Sigma|^{k+1}\) edges [29]. Vertices in the graph with indegree(\(V\)) > 1 are called junctions and vertices with indegree(\(V\)) = 0 are referred to as source nodes. A vertex in \( G \) with outdegree(\(V\)) > 1 is known as a bifurcation and those vertices with outdegree(\(V\)) = 0 are sink nodes. As the size of a k-mer de Bruijn graph is significantly greater than the k-spectrum of a genome, i.e. \(| \Sigma|^k >> G^k\), the de Bruijn graph constructed from a reference genome or a set of sequence reads is implicitly sparse [144]. Notwithstanding the reduced number of nodes required to represent \( G^k \) in a k-mer graph, the multiplicity of nodes still remains vast, necessitating the employment of an efficient data structure to model the vertices and edges.

The de Bruijn graph data structure employed in the prototype is based on a hash map implementation of an adjacency list [51, 208], where a map \( G \) is defined as \( G: \text{String} \rightarrow \text{Node} \). In this formulation, k-mers are used as string keys in a hash map, with the default implementations of the hashCode() and equals() methods retained. In contrast with the de Bruijn graph structures
described by Pevzner et al [78] and Idury and Waterman [143], the graph implementation in Ferox maps $k$-mers to nodes. This formulation is similar to the approaches used by both Simpson et al [48] and Zerbino and Birney [49], where nodes represent sequences and paths represent possible assemblies. However, the node design used in the prototype eschews the twin node structures described by the latter authors, as these models hinder the optimisation of node compression when transforming a de Bruijn data structure into a sequence graph. By modelling the adjacency list as a mapping of $k$-mers to nodes, complementary nodes are already implicit in the graph as the reverse complement of a $k$-mer key. In addition, the Ferox prototype does not directly incorporate paired reads into the graph, as suggested by Medvedev et al [150], but relies instead on object-oriented techniques to apply paired read constraints during contig assembly. As a consequence of modelling $k$-mers as nodes in a de Bruijn graph, the Eulerian Superpath formulation of contig assembly described by Pevzner et al [78] is not viable.

Consistent with the approach used by Zerbino and Birney [49], the prototype employs heuristics and constraints to restrict node expansion when transversing the graph during contig assembly.

A UML representation of the essential components in the design of the prototype graph module is depicted in Fig. 5.2. The singleton Graph class is responsible for managing the node adjacency list and exposes methods for adding, modifying and querying the graph. The most salient methods include mechanisms for adding and removing nodes, adding sequence reads, transforming to a sequence graph and querying for a set of candidate source nodes.

The basic behaviours of a $k$-mer node are abstracted into the Node interface. The implementing class DefaultNode exploits the composite pattern described by Gamma et al [209], enabling the edges emanating from a node to be represented implicitly as object references in an array list. The stateful information encapsulated in DefaultNode is the nucleotide sequence that the node represents, along with an integer instance variable used to track the number of times a node is visited.
As noted by Miller et al [27], when adding a sequence read to a de Bruijn graph, its decomposition into a tiling of $k$-mers results in the loss of long-range continuity information. Despite the critique by Myers [65], that this reduction renders de Bruijn graphs read incoherent, the prototype implementation requires that paths in the graph must be supported by their underlying reads. This is accomplished using instances of the class ReadIndex, by associating the index position of a read with a node when the read is first added to the graph. The utilisation of read indices, shown in Fig. 5.3, enables distance constraints to be applied to putative paths in the graph, enabling reliable transversal through bifurcated nodes and cycles. This approach is analogous to the read threading technique described by Chaisson and Pevzner [29] for resolving paths through repetitive regions that are spanned by one or more reads. In addition, the type ReadIndex maintains an object reference to its parent Read class, enabling access to the various facets of a read, such as orientation and length, from any node in the graph that a read aligns to.

The class Read provides an abstraction of the state and behaviour of a shotgun sequence read. Characteristics, such as name, sequence, and orientation are associated with instances of this class, as reads are parsed from
a FASTA input file. Each read also retains a pointer to its reverse complement and a Boolean flag that indicates whether or not the read has already been processed. The prototype supports paired reads, allowing the delimiting characters in a read name, along with the distance between paired reads, to be specified declaratively in a configuration file. This information is exploited during contig assembly to provide a mechanism for mate threading [27], enabling the establishment of joins spanning collapsed repeats that are shorter than the paired-end distance.

As early as 1995, Idury and Waterman [143] had identified the topological features of $k$-mer graphs as key properties for facilitating genome assembly. In particular, they showed how the elimination of singletons not only transforms and simplifies a $k$-mer graph, but also provides a mechanism for identifying and merging unique consecutive $k$-mer sequences. The merged nodes in the resultant sequence graph are logically equivalent to the unitigs described by Myers et al [69], with the remaining nodes of size $k$ corresponding to repetitive sequences. Merging uncontested nodes in a de Bruijn graph corresponds to the first phase of assembly and is a salient characteristic of $k$-mer graph assemblers [47, 49, 78].

Node compression in the prototype is implemented by instances of the interface $NodeMerger$. Once the set of sequence reads have been added to the de Bruijn graph, a depth-first search is employed to transverse the graph in linear time [152], merging consecutive nodes where:

$$\text{indegree}(V) = 0 \lor \text{indegree}(V) = 1 \land \text{outdegree}(V) = 1 \quad (5.1)$$

As the nucleotide sequence of a node is extended, its constituent read indices are modified to reflect the new sequence length. In addition to modifying the index position of a read in a merged node, instances of $ReadIndex$ also modify a merged-index variable, used to denote the starting position of a read in a unitig. The elimination of singletons further results in superfluous read indices being discarded from the graph, as the highest read index in a merged node provides sufficient information to enforce distance constraints when transversing the graph. As a consequence of merging nodes and read indices,
the transformation from a $k$-mer to a sequence graph significantly simplifies
the data structure, with an accompanying large reduction in space complexity.

Fig 5.3. Read coherence in the Ferox prototype. (a) A 4-mer de Bruijn graph for
the overlapping sequences GTTACTTTCTCTTA and TCTTACCGGG. In
practice, $k$-mer sizes of at least 21 are used. As the $k$-mers of each read are
added to the graph, the index position of each $k$-mer in the read, relative to the
node sequence, is recorded. This information enables reliable transversal
through highly repetitive nodes, by following the indices of reads in increasing
order. (b) Merging unambiguous nodes transforms the data structure into a
sequence graph. The starting index of each read, relative to a merged node, is
altered to reflect the length of the newly merged sequence and superfluous read
indices are removed. This transformation greatly simplifies the graph and
significantly reduces memory complexity.
5.3 Anchoring Reads with Fuzzy k-mers

Comparative approaches to genome assembly require the anchoring of a set of shotgun sequence reads to a reference genome before generating or extending contigs using the positional information provided by the anchors. The Alignment-Layout-Consensus model, employed by Pop et al [68] and Strömberg [102] relies entirely on the correct placement of reads aligned by MUMmer [16, 81] and Mosaik respectively and does not perform well where a weak homology exists between the target and reference genomes. Although paired reads can be used to resolve the anchoring ambiguity induced by repeats or segmental duplications, contig assembly is confounded by the presence of major structural variations such as insertions and inversions. The anchoring mechanism described by Gnerre et al [63] for “assisted” assembly is predicated on the accurate alignment of sequence reads using BLASTZ [165]. This approach groups together anchored reads into proto-contigs and uses the anchoring information to extend the set of contigs generated by a concurrent de novo assembly of reads.

In contrast with these approaches, anchoring is integrated directly into the prototype and is used to provide heuristic information to the assembler during graph transversal. As the anchoring mechanism is used in tandem with conventional techniques when expanding graph nodes, structural differences between the target and reference genomes are less likely to stymie contig processing. A notable facet of the approach utilised in the prototype is the pre-processing of the reference genome to identify and remove repetitive sequences that may induce alignment ambiguities. This is achieved by constructing a de Bruijn graph from the reference genome and then extracting unitig sequences after compressing the graph. Using anchors in this manner is analogous to the long-established strategy of employing bacterial artificial chromosomes (BACs) and jump libraries to provide long-range positional information to the contig assembly process.
5.3.1 Identifying and Extracting Anchor Sequences

The anchor detection and extraction process consists of three main phases; the construction of a de Bruijn graph to represent overlapping \( k \)-mers, the transformation of the de Bruijn graph into a sequence graph and the extraction of unitig sequences from the transformed graph. As these unitig sequences are constructed from sets of unique overlapping \( k \)-mers, they represent unambiguous substrings of a reference genome and are thus ideal candidate anchors sequences.

![De Bruijn graph and sequence graph](image)

**Fig 5.4.** Anchor identification and extraction. After constructing a de Bruijn graph, unambiguous nodes are merged, transforming the data structure into a sequence graph. Merged nodes, shown in yellow, correspond to unique anchoring sequences.

In the context of anchor identification, a de Bruijn graph creates a perfect tiling path through a complete reference genome. Commensurate with the technique describe in Section 5.2, instances of the class ReadIndex are used to label graph nodes. Transformation of the de Bruijin into a sequence graph requires the application of the following predicate when compressing nodes:

\[
\text{indegree}(V) = 0 \vee \text{indegree}(V) = 1 \land \text{outdegree}(V) = 1 \land \#V.\text{indices} = 1
\]  

\(5.2\)
The addition of a constraint on the number of read indices is required to prevent node compression, where a cycle exists in the graph that includes a sequence of nodes with an in-degree of one. Consistent with the technique described in Section 5.2, the elimination of singletons both compresses graph nodes and discards redundant instances of the type *ReadIndex*.

Unitig nodes are extracted from the sequence graph in linear time, using a depth-first search, and are saved in both FASTA format and as a serialized map. As anchors constructed from short unitig sequences are of little utility, the depth-first search can be parameterised to exclude anchors below a minimum length threshold. Critically, the exact position of each anchor in the reference genome, given by the unary *ReadIndex* at each compressed node, is recorded during the extraction process. This enables the extracted set of anchors to be sorted in increasing order and provides a heuristic for identifying starting nodes during contig assembly. It is noteworthy that anchor identification and extraction is both independent of the overall assembly process and is a single event for a given k-mer size of a reference genome.

### 5.3.2 Aligning Sequence Reads to Anchors

The fuzzy $k$-mer model, described in Chapter 4, is utilised by the prototype to align and anchor shotgun sequence reads. The anchoring process executes in parallel with the construction of the assembly graph, as reads are parsed from a FASTA input file. Before the set of sequence reads can be aligned, the extracted anchors must first be inserted into a fuzzy hash map.Instances of *FuzzySeed* are created and initialised with the pattern, alignment algorithm, $\beta$ cut-off threshold and minimum alignment constraint specified in a configuration file.

An instance of the class *Read* is created and initialised from each read parsed in the FASTA input file and added to a blocking queue. A separate thread of execution iteratively polls from the head of the queue and dispatches the *Read* for further processing. Reads are processed and anchored in both orientations,
but each orientation is maximally constrained to a single anchoring position along the reference genome. The $k$-mer content of each read is simultaneously added to the de Bruijn graph and aligned using the fuzzy $k$-mer model in separate threads of execution. For each fuzzy $k$-mer match, the name and index of the anchor alignment is recorded. A minimal alignment threshold is applied to the set of anchors aligned to a sequence read, with the remaining anchors ranked by alignment score. The instance of $Read$ is then assigned the highest scoring anchor from the sorted set of candidate unitigs.

It is noteworthy that each anchor maintains a list of the name, orientation and starting position of each aligned read. Sorting this list yields the order in which the set of reads align to an anchor. As each anchor knows its own starting position with respect to the reference genome, this process creates a syntenic ordering of the full set of aligned reads.

### 5.4 Contig Assembly

As noted by Simpson et al [48], contig assembly may be viewed as the merging together of nodes in a de Bruijn graph that are unambiguously connected. Advocates of this opinion, which includes both Butler et al [47] and Li et al [33], eschew further processing of graph nodes after the elimination of singletons. As an alternative, the initial set of contigs generated by node compression is scaffolded, with paired read information used to link and extend the contigs.

Commensurate with the original formulation of a $k$-mer graph by Idury and Waterman [143], other assemblers attempt to extend contigs by computing paths through the compressed graph [4, 6]. The heuristic approach described by Zerbino and Birney [149] uses a bounded breadth-first search, along with read and paired read threading, to expand nodes and extend contigs. Similarly, the node expansion method described by Chaisson and Pevzner [29] also exploits read and paired read threading, in conjunction with coverage heuristics and graph simplification techniques. Although contig assembly in the Ferox prototype also applies read and paired read threading to resolve
ambiguous paths through the sequence graph, the anchoring information provided by the fuzzy $k$-mer alignment is heavily exploited as a path selection heuristic.

The graph transversal process implemented in the prototype requires the identification of all potential starting nodes from the compressed sequence graph. The set of starting nodes, each of which is a potential contig, corresponds to all of the source and compressed nodes in the graph. A strategy pattern [202] is implemented in the $Graph$ class that returns a queue of candidate starting nodes, sorted by their anchoring index relative to the reference genome. Where a merged node is aligned to more than one anchor, either by spanning separate anchors or though the alignment of read complements, its anchor index is computed from a majority count of the lowest indexed anchor supported by the underlying reads.

The process of contig extension involves the expansion of nodes along a path through the sequence graph and the addition to a contig of the unique sequence information encapsulated by each node. It is noteworthy that, as a consequence of node compression, the remaining nodes in the graph of size $k$ are either junctions or bifurcations, representing alternative assembly paths. Resolving the ambiguity induced by these nodes is imperative for the correct assembly of a genome.

Contig assembly in the prototype is based on the premise that anchoring information, in conjunction with read and paired read threading, provides sufficient information to resolve the ambiguities effectuated by junctions and bifurcations in the merged graph. The contig assembly module employs a priority queue when determining the next node to expand during graph transversal. The priority queue exploits read indices, along with paired read constraints and anchoring information to determine queue membership and priority respectively. At an abstract level, this general approach exhibits the behaviours associated with both a constraint-satisfaction search and a best-first search.

For each child node expanded, its constituent instances of $ReadIndex$ are offered to the queue, together with the current index of the contig being
Membership of the priority queue is constrained to those nodes containing read indices that are consistent with the underlying set of reads already added to a contig.

**Procedure:** offer

**Input:** Node node, ReadIndex \( r_i \), int contigIndex, Queue \( Q \)

**Output:** void

1. IF \( \text{read}(r_i).\text{processed} \) OR \( \text{read}(r_i).\text{complement}.\text{processed} \) THEN RETURN

2. IF \( \text{read}(r_j) \in Q \) THEN

3. IF \( \exists r_j \in Q \mid \text{read}(r_j).\text{complete} \) THEN RETURN

4. IF \( \exists r_j \in Q \mid r_i = r_j + \text{len}(r_j.\text{node}) - k - 1 \) OR
   \( \exists r_j \in Q \mid r_i = (\text{len}(r_j.\text{node}) - k - 1) - r_j^{mi} + 2 \) THEN

5. IF \( \exists r_j \in Q \mid r_i < r_j \) THEN RETURN

6. \( \text{read}(r_i).\text{validated} \leftarrow \text{TRUE} \)

7. IF \( r_i^{ci} + \text{len}(\text{read}(r_i)) - 1 \leq \text{contigIndex} + \text{len}(\text{node}) \) THEN
   \( \text{read}(r_i).\text{complete} \leftarrow \text{TRUE} \)

9. END IF

10. END IF

11. ELSE

12. IF \( r_i = 1 \) THEN

13. \( \text{read}(r_i).\text{validated} \leftarrow \text{TRUE} \)

14. IF \( r_i^{ci} + \text{len}(\text{read}(r_i)) - 1 \leq \text{len}(\text{node}) \) THEN \( \text{read}(r_i).\text{complete} \leftarrow \text{TRUE} \)

15. add \( \text{read}(r_i) \) to \( Q \)

16. END IF

17. END IF

Fig 5.5. Constraints employed when expanding nodes in the sequence graph.
The index of an existing read in the queue is denoted by \( r_j \), with \( r_j^{mi} \) and \( r_j^{ci} \) denoting the merged index of an existing read and the contig index of the current read respectively.
As shown on lines 12-15 of Fig. 5.5, newly encountered reads, with a starting index of one, are labelled as validated and are added to the queue. Those reads that are entirely contained by a merged node are marked as complete, triggering the reverse complement of a read to be also flagged as processed. This prevents the subsequent processing of uncontested sub-paths of the graph that correspond to the reverse complement of a processed sequence and forestalls segments of a genome from being assembled twice.

For reads that have already been added to the priority queue, their continuing membership is predicated on the satisfaction of the disjunction of the two constraints described on line 4 of Fig 5.5. This first of these predicates is only true where the value of a read index is equivalent to the sum of the previous read index and the length of the nucleotide sequence of the last processed node. The second predicate is necessary to handle the condition where the starting index of a read aligns to the middle of a compressed node. A further constraint is applied to read indices that comply with these predicates, ensuring that the lowest read index is selected from a node where the multiplicity of indices for a given read is greater than one. As this condition is only true where a cycle exists in the graph, selecting read indices in increasing order enables the assembler to reliably transverse loops. Consistent with the technique described for processing newly encountered contained reads, both completed reads and their reverse complements are labelled as processed.

The semantics for computing the comparability of elements in the priority queue are encapsulated in an instance of the interface Comparator [196] and are shown in Fig 5.6. The Comparator interface mandates that the order of elements in collections be denoted by integer values of -1, 0, or +1, corresponding to less than, equal to and greater than respectively. After the set of adjacent nodes are added to the priority queue, an optimised merge sort [194], encapsulated in the class java.util.Collections, modifies the order of elements by applying the semantics of the comparator.
Procedure: compare

Input: NodeQueueElement current, NodeQueueElement next, Queue Q

Output: -1, 1, 0 denoting less than, greater than, or equal to.

1. IF current.validated AND NOT next.validated THEN RETURN -1;
2. IF NOT current.validated AND next.validated THEN RETURN 1;
3. IF hasValidAnchor(current) AND NOT hasValidAnchor(next) THEN RETURN -1
4. ELSE IF NOT hasValidAnchor(current) AND hasValidAnchor(next) THEN RETURN 1
5. ELSE IF hasValidAnchor(current) AND hasValidAnchor(next) THEN
6. IF anchorScore(current) > anchorScore(next) THEN RETURN -1
7. ELSE RETURN 1
8. ELSE
9. IF current.mate ∈ Q AND next.mate ∉ Q THEN RETURN -1
10. ELSE IF current.mate ∉ Q AND next.mate ∈ Q THEN RETURN 1
11. ELSE
12. IF lookAhead(current) ≤ lookAhead(next) THEN RETURN -1
13. ELSE RETURN 1
14. END IF

Fig 5.6. Priority heuristics used in the node queue. The semantics of comparability exploit the anchoring information, provided by the fuzzy k-mer alignment, to guide the assembler when expanding nodes. Mate threading and a bounded look-ahead are also used to determine node priority.

A salient feature of the comparator heuristics, as shown in Fig. 5.6, is the exploitation of anchoring information when determining the priority of nodes in the queue. The method hasValidAnchor() checks if an anchored read aligns to either the current anchor or its two unprocessed preceding or succeeding anchors. This technique enables the prototype to apply anchoring heuristics to both genomes that exhibit a strong synteny and those that contain segmental inversions. In the event of a tie between two reads, the original anchor score computed from the fuzzy k-mer alignment is used to determine priority.
**Priority Queue**

**At Node 1: Expand Node 2**  
R1, R2, R3  
R1 is aligned to Anchor 1. R2 and R3 added to queue. Look-ahead favours R2 over R3.

**At Node 2: Expand Nodes 3 & 9**  
R1, R2, R3  
Queue order retained.

**At Node 3: Expand Nodes 4 & 5**  
R1, R2  
R3 is now invalid and removed. Loop twice through cycle 2-3-4-2 using increasing read indices.

**At Node 5: Expand Nodes 6 & 10**  
R1, R4, R5, R2  
R4 and R5 align to Anchor 1 and have a higher priority than R2.

**At Node 10: Expand Nodes 11 & 12**  
R4, R5  
R1 is complete. R2 is now invalid and removed from queue. As R4 and R5 share the same starting position and anchor, a bounded-lookahead will determine the next node to visit.

**Fig 5.7.** Expanding nodes during graph transversal. Compressed nodes are depicted in yellow. Repetitive nodes of size $k$ are shown in white. A bounded look-ahead from Node 2, with a depth of 3, is depicted by the green edges.

In the absence of anchoring information, the comparator favours nodes connected by paired reads that are consistent with their distance constraints and orientation. Failing this, a bounded look-ahead is performed, that searches for succeeding merged nodes in the graph containing reads aligned to either the current anchor or its adjacent anchors. The bounded look-ahead is implemented as a depth-limited depth-first search using the algorithm described by Coppin [133], with a default look-ahead depth of four.
Restricting the look-ahead to merged nodes is necessary, as existing nodes of size \( k \) represent junctions and bifurcations and, even if anchored, only provide ambiguous positional information to the assembler.

The contig extension process and the expansion of graph nodes continues while validated nodes remain in the priority queue. If the priority queue returns a null object reference, the next unprocessed candidate source node is selected and a new contig is started. This process of contig assembly iterates until the remaining list of unprocessed candidate source nodes has been exhausted.

### 5.5 Contig Scaffolding

The post-processing of contigs in a subsequent scaffolding phase is a well-established feature of genome assembly [116]. In its original formulation by Myers et al [69], as an essential component of the Overlap-Layout-Consensus model, scaffolding is viewed as the process of ordering, orientating and positioning contigs relative to one another, by judiciously enforcing the distance and orientation constraints provided by paired reads. Moreover, Koren et al [138] make the observation that scaffolding can also be used to extend and fill gaps in an assembly, where an insufficient overlap exists between adjacent contigs. The application of scaffolding to de novo assembly can also be supplemented with post hoc comparative techniques [159, 162, 163], incorporating the use of anchoring and alignment information to order and orientate contigs. Such comparative scaffolding techniques are useful where contigs are separated by gaps in an assembly, induced by regions of low sequence coverage.

Scaffolding is implemented in the prototype as a module that exploits both anchoring and paired read information to group together contigs assembled using the graph transversal mechanism described in Section 5.4. Based on the Contig Adjacency Graph described by Husemann and Stoye [158], a weighted directed graph is constructed, in which vertices represent contigs and weighted edges connect adjacent vertices that share paired reads or align
to the same region of a reference genome. The objective of the scaffolding phase in Ferox is to order and orientate the set of contigs into groups and to provide an estimate of the distance between grouped contigs, using either anchoring or paired read information. This general approach of grouping contigs, based on anchoring and paired read constraints, is analogous to the “assisted assembly” and gene-boosting techniques for improving de novo assembly described by Gnerre at al [63] and Salzberg at al [156] respectively.

The Contig Adjacency Graph is constructed by polling from the priority queue of aligned reads maintained by each anchor. As reads are polled from the head of the queue, edges are added to the graph where two adjacent contigs contain reads that align to the same anchor. Where an anchor spans two contigs, a minimum threshold of anchored reads, specified in a configuration file, must be present in both contigs to establish a join. Applying an anchor threshold is consistent with the technique described by Gnerre at al [63], where a minimum threshold of three anchor links is required to establish a join between adjacent contigs. In addition to establishing proximity relationships, anchor alignments also facilitate the computation of an approximate distance between contigs. In contrast with the gene-boosted approach of Salzberg at al [156], where distance information and sequence content can be accurately inferred from highly conserved protein sequences, the propensity for variation in nucleotide sequences reduces proximity calculations in the prototype to rough estimates.

Paired read information is also used to establish joins between adjacent contigs. Using a variation of the edge-bundling technique employed by Pop et al [110] in the Bambus scaffolder, the prototype bundles together parallel edges linking adjacent contigs into a single weighted edge. Existing edge weights, computed from anchor alignments, are incremented for each paired read that spans adjacent contigs. As paired reads provide a more accurate mechanism for calculating proximity distance, an estimated distance computed solely from anchor joins will be supplanted by the distance inferred from any present paired read joins.
Fig 5.8. Contig adjacency graph, showing a group of connected contigs. Anchor alignment and paired read information are used to connect adjacent contigs with a single bundled edge. Edge weights, depicted in red, denote the multiplicity of anchored and paired reads. Proximity distance, shown in black, can be computed using either anchoring or paired read constraints. A linear ordering of the vertices in the Contig Adjacency Graph can be achieved using a heuristically informed topological sort.

A linear ordering of the contigs in each connected subgraph of the Contig Adjacency Graph is achieved by performing a topological sort [208] that maximises the heaviest sequence path to each vertex from a source node. For each group of connected contigs, each contig is assigned a group position index computed from the maximum of the indices and sequence length of preceding vertices, along with the estimated gap between contigs. A comparator is then employed to sort each group of connected contigs using the group position index. As a supplement to outputting the set of assembled contigs in FASTA format, Ferox uses this information to generate an XML document containing the contig groups, along with the sequence, size and distance between grouped contigs. Ancillary information, such as read composition, contig starting position, read order and orientation are also given in the XML document. The set of unassembled singleton sequences is also provided by the assembler, as a FASTA output file.
5.6 Chapter Summary

Current models of de novo assembly are characterised by the merging of uncontested graph nodes up to the boundary of repetitive sequences, followed by the extension of contigs using scaffolding and read threading techniques. Despite their incorporation of anchoring and orthologous alignments, existing comparative assemblers either omit the overlap phase completely or use reference alignments in a post-assembly contig extension operation.

This chapter presented and described a novel mechanism that directly integrates anchoring information into the contig assembly process. Using the de Bruijn graph model of assembly, anchoring and read threading are used as path selection heuristics to guide the assembly of contigs through repeat boundaries in a compressed sequence graph. Anchoring and paired-read information is also employed in a scaffolding phase to group together and provide an estimate of the distance between adjacent contigs.
Chapter 6

Evaluation of the Fuzzy $k$-mer Model

The ubiquity and longevity of hash-based approaches to sequence alignment and genome assembly arises from the average $O(1)$ running time of basic hash map operations [12]. As noted by Li et al [54] however, while using smaller sizes of $k$ increases the sensitivity of a $k$-mer alignment, this has the concomitant effect of decreasing both alignment speed and specificity. The evaluation and discussion presented in this chapter addresses these key features of $k$-mer matching and explores those aspects of the fuzzy $k$-mer model that impact on the speed, sensitivity and specificity of alignments.

Benchmarking the fuzzy $k$-mer model involved the development of a testing framework to facilitate an objective and accurate analysis of the key features of the approach. The effect of fuzzy $k$-mer characteristics, such as seed type, hash size, $\beta$ cut-off threshold and $k$-mer size on the speed, sensitivity and specificity of alignments, was evaluated using real, synthetic and random DNA sequences.
The genomes used in the study were selected on the basis of their relative size and degree of homology, with the latter determined using the DNA-to-DNA Hybridization (DDH) distance metric described by Auch et al [154, 155]. The DDH distance is calculated by aligning two genomes and then inferring the genetic distance between the two sequences, from the set of high-scoring pairs produced. The metric is a real number between 0 and 1, with lower values indicating a strong homology between two genomes. The DDH metric used in this thesis is based on Formula 1 described by Auch et al [154], i.e. \[ DDH = \frac{\text{HSP length}}{\text{total length}} \], where the HSP length was computed using MUMmer [16, 80, 81].

The fuzzy \( k \)-mer model was tested using real sequence reads acquired from GenBank, including sets of Illumina, 454 and Sanger-length reads. Synthetic DNA sequences, extracted from completed genomes, were also used in the study to provide a mechanism for measuring sensitivity and specificity. In addition to real and synthetic genome sequences, a set of randomly generated sequences was employed to eliminate the effect of homological bias on the speed of the approach. The testing framework also included a comparative analysis of the speed of the underlying fuzzy hash map used by the Ferox prototype with that of a conventional hash map and a tree map.

The testing framework further necessitated a comparison of the results obtained using the fuzzy \( k \)-mer approach with those produced by established \( k \)-mer based aligners. BLAT [18] and Mosaik [102] were selected to evaluate the fuzzy \( k \)-mer model against the “seed and extend” and “spaced-seed and extend” approaches respectively. The implementation of the “seed and extend” model used in BLAT aligns \( k \)-mers extracted from sequence reads against a set of non-overlapping \( k \)-mers from a reference genome. BLAT can accommodate a single polymorphism at the end of a \( k \)-mer by making separate lookups of a hash table for each terminating base. The Mosaik aligner clusters \( k \)-mer matches of reads against a hash table of reference positions using spaced seeds, before evaluating each cluster with a full Smith-Waterman [15] alignment. The different approaches were compared and evaluated using both synthetic and real sequences of different read lengths.

All results presented in this chapter were compiled from executing the suite
of aligners on an OSX 10.6.8 platform, with a single 3.2 GHz Intel Core i3 processor and 16GB of RAM. In addition, an instance of the Java HotSpot 1.6 64-bit virtual machine was used as the runtime environment to test the Ferox prototype.

6.1 Speed Benchmarks

A key requirement of the testing framework was to establish those components of the fuzzy k-mer model that impact on the running time of alignments. The results described in this section are derived from experiments with the hash size, k-mer size and the β cut-off threshold of fuzzy seeds. In addition to these tests, the load and alignment times of the fuzzy hash map implementation was compared with the running times obtained from alignments using a conventional hash map and a tree map. In contrast with the average $O(1)$ access time of a hash map, a tree map employs a binary search tree to sort its constituent hash keys and guarantees $O(\log(n))$ running time for search, insertion and deletion operations [194].

6.1.1 Effect of Hash Size on Running Time

In the original formulation of the “seed and extend” approach, developed by Pearson and Lipman [19], seeds represent exact matching k-mers, where the size of $k$ also denotes the hash size of a search key. The trade-off between speed and sensitivity is controlled by the k-mer size, with smaller values of $k$ increasing the possibility of detecting a local alignment, at the expense of an increased running time. Larger values of $k$ decrease running time and increase specificity, but also decrease the sensitivity of an alignment. In the context of spaced seeds, hash size is not controlled by k-mer size, but by the seed weight [55, 56], with heavier weights increasing both speed and specificity.

Fig. 6.1 shows the results of aligning the genome of B. suis 1330 against a 1Mbp sequence of randomly generated nucleotide symbols, with a DDH distance of 0.9999. A set of 15 different hash sizes ranging from 6-20 indices
was used to compare the speed of a fuzzy $k$-mer alignment with that achieved using a hash map and a tree map. All fuzzy seeds used in the experiment were 24-mers, with the number of hash positions increased incrementally from 6 to 20 indices, i.e. with seed patterns ranging from #fffffff to #fffffff####. The $\beta$ cut-off threshold was fixed at 0.5 for all fuzzy seeds. The seeds employed in the hash map and tree map ranged in size from 6-mers to 20-mers, reflecting the number of hash positions to use. This discrepancy in seed size explains the larger space complexity of a fuzzy seed at lower hash sizes and the accordance in memory complexity as the hash size increases.

![Fig 6.1](image.png)

**Fig 6.1.** Running times and space complexity for the alignment of *B. suis* 1330 against a 1Mbp random genome. The green lines correspond to fuzzy $k$-mer alignments, with the blue and maroon lines denoting alignments using a hash map and a tree map respectively. An implementation of the Levenshtein algorithm was used to compute the fuzzy set membership function $\mu_A(S)$. 

104
Notwithstanding the discrepancy in $k$-mer size, the fuzzy $k$-mer approach is clearly less memory efficient than either a hash map or a tree map. This is due to the overhead associated with mapping instances of FuzzyHashKey to fuzzy sets of reference indices, as opposed to the use of string keys in the conventional maps. Because they utilised the same hash indices, the hashCode() method was guaranteed to return the same integer value and subsequent bucket index for all three maps. The implication of this behaviour is that the graph is really depicting the relative cost of the collision detection mechanism used in each map, i.e. the cost of executing the implementation of the equals() method in the search key used by each type of map. For the string search keys, used by the hash map and tree map, the default implementation of equals() returns true only where the $k$-mer content of a search key exactly matches the $k$-mer content of an existing key in the linked list at a bucket index. The fuzzy $k$-mer approach supplants this definition of key equality with the result returned by a string similarity algorithm constrained by a $\%$ cut-off threshold.

Consistent with the speed of a hash map and a tree map, the running time of an alignment begins to rapidly increase once the number of hash indices in a fuzzy $k$-mer falls below 11. Given the randomised content of the reference genome, for values of $k \geq 11$, its constituent $k$-mers are well distributed throughout the buckets in all three maps. As the set of hash indices in a search key is reduced below 11, the number of hash collisions in the map begins to rapidly escalate. This has the consequence of flattening the hash map into a small collection of buckets, each of which contains an increasingly large linked list of hash keys and their satellite data. Compacting a large number of hash keys into a relatively small number of occupied buckets degrades the running time for insertion and search operations from an average $O(1)$ to $O(n^2)$, with the latter time arising in the extreme condition where a constant hash code is computed for all instances of FuzzyHashKey, effectively flattening the fuzzy hash map into a linked list.

In contrast with the hash and tree maps, the alignment time begins to level off once the hash size falls below 8 indices. This levelling off in time complexity illustrates an important feature of the fuzzy $k$-mer approach. As the hash size
is reduced, the number of key collisions in the map increases, forcing the invocation of the `equals()` method against the set of existing keys at a bucket index. Notwithstanding the randomly generated genome used in this test, the \( \beta \) cut-off threshold of 0.5 results in an increasing number of `FuzzyHashKey` instances being added to the linked list attached to each bucket in the map, with some fuzzy sets containing more than one fuzzy \( k \)-mer. This contrasts with the use of an exact-match \( k \)-mer, where a large number of \( k \)-mers are compacted into a relatively small number of buckets. The levelling off in running time for values of \( k \leq 8 \) is due to fuzzy approach producing a larger number of hash keys than an exact matching \( k \)-mer.

This extreme case illustrates the essence of the fuzzy \( k \)-mer approach – the grouping together of approximately matching \( k \)-mers into a single fuzzy set in the linked list at a bucket index. An increase in the number of hash keys in a map is logically equivalent to a reduction in the overall load factor of a map, \( \alpha = |G_k|/m \), where \( m \) is the number of keys. Although \( \alpha \) is assigned a constant value, typically 0.75, when a map is first instantiated, measuring the total number of keys produced when a reference genome is loaded into a map allows for the measurement of the effective load factor \( (\alpha') \) of a map. The effective load factor \( (\alpha') \) represents the average number of \( k \)-mers in the linked list at a non-empty bucket index and is most significant factor that impacts on the running time of \( k \)-mer operations on a map. Any increase in \( \alpha' \) will lead to an increase in the number of hash collisions in a map and degrade the average \( O(1) \) running time of map operations. Conversely, any reduction in \( \alpha' \) will result in a corresponding reduction in running time.

The comparison of alignment times shown in Fig 6.1 demonstrates that, for a reasonable number of hash indices, the running time of a fuzzy \( k \)-mer alignment is more efficient than that of a tree map and broadly similar to that of a hash map. The comparison also indicates that, for a randomly generated reference genome and a hash size > 11, configuring a fuzzy seed to use a string similarity algorithm with a time complexity of \( O(n^2) \) has no significant impact on the overall running time of an alignment.
The load times for the alignment of \textit{B.suis} 1330 against the randomly generated 1Mbp genome are shown in Fig 6.2 and further illustrate the nature of the fuzzy \textit{k}-mer approach. For hash sizes > 11, the fuzzy \textit{k}-mer model loads the reference genome in constant time. As the number of hash indices is decreased, the volume of collisions in the fuzzy hash map escalates rapidly, resulting in the \textit{k}-spectrum of the reference genome being loaded into a smaller number of buckets, each of which contains a large linked list. The small hash size forces the map to execute the \textit{equals()} method of an instance of \textit{FuzzyHashKey} against all the keys at a bucket index, each time a new key is inserted into the hash map.

\textbf{Fig 6.2.} Load times for the alignment of \textit{B.suis} 1330 against a 1Mbp random genome. The area shaded in green reflects the cost of using a dynamic programming algorithm, with a time complexity of \(O(n^2)\), in a fuzzy \textit{k}-mer versus the \(O(n)\) implementation of \textit{equals()} used by the String class.
In contrast with a search operation on a hash map, which returns the satellite data related by the first matching hash key, an insertion operation requires a key comparison against all the existing hash keys in the linked list at a bucket index [50, 51]. This \( O(n) \) time search of the linked list is necessary to ensure that the map maintains a unique collection of keys that relate to values. In a fuzzy hash map, an insertion operation will thus require the invocation of a string similarity algorithm against all of the fuzzy hash keys at a bucket index. If the number of string indices used to compute a hash code is reduced to a very small number, the multiplicity of key comparisons required for an insertion operation will rise significantly, resulting in a subsequent increase in the overall load time. In the context of running time, this situation approaches the worst possible condition where a dynamic programming algorithm, with a quadratic time complexity, is executed each time a new fuzzy \( k \)-mer is inserted into the map.

Table 6.1 shows the results of an alignment of \( B. suis \) 1330 against the genome of \( B. suis \) ATCC. For this comparison, the same parameters were applied as were used in the alignment of \( B. suis \) 1330 against a randomly generated genome. The genome sequences are highly homologous, with a DDH distance of 0.0861. Due to the high level of sequence similarity, a fuzzy \( k \)-mer alignment will cause a large number of hash collisions, with each collision resulting in the execution of the approximate string-matching algorithm in the \emph{equals()} method of the fuzzy seed. As depicted graphically in Fig 6.3, despite the strong homology between the two sequences, for hash sizes > 11 indices, the running time of the fuzzy \( k \)-mer approach is consistent with that of the constant and sub-linear times of a hash and tree map respectively. This accordance in running time occurs despite the fact that the fuzzy approach allows for an approximate \( k \)-mer match, in contrast with the exact match constraints imposed by conventional \( k \)-mer seeds.
<table>
<thead>
<tr>
<th>Memory (MB)</th>
<th>Load Time (s)</th>
<th>Align Time (s)</th>
<th>$\beta$ Cut-off threshold</th>
<th>$k$</th>
<th>Hash Size</th>
<th>Fuzzy Seed</th>
<th>Memory (MB)</th>
<th>Load Time (s)</th>
<th>Align Time (s)</th>
<th>Tree Map</th>
</tr>
</thead>
<tbody>
<tr>
<td>369</td>
<td>4</td>
<td>6</td>
<td>0.5</td>
<td>24</td>
<td>20</td>
<td></td>
<td>177</td>
<td>1</td>
<td>1</td>
<td>170</td>
</tr>
<tr>
<td>369</td>
<td>5</td>
<td>7</td>
<td>0.5</td>
<td>24</td>
<td>19</td>
<td></td>
<td>177</td>
<td>1</td>
<td>1</td>
<td>170</td>
</tr>
<tr>
<td>373</td>
<td>4</td>
<td>7</td>
<td>0.5</td>
<td>24</td>
<td>18</td>
<td></td>
<td>176</td>
<td>1</td>
<td>1</td>
<td>170</td>
</tr>
<tr>
<td>373</td>
<td>4</td>
<td>8</td>
<td>0.5</td>
<td>24</td>
<td>17</td>
<td></td>
<td>177</td>
<td>1</td>
<td>1</td>
<td>170</td>
</tr>
<tr>
<td>373</td>
<td>5</td>
<td>8</td>
<td>0.5</td>
<td>24</td>
<td>16</td>
<td></td>
<td>177</td>
<td>1</td>
<td>1</td>
<td>170</td>
</tr>
<tr>
<td>373</td>
<td>5</td>
<td>9</td>
<td>0.5</td>
<td>24</td>
<td>15</td>
<td></td>
<td>177</td>
<td>1</td>
<td>1</td>
<td>170</td>
</tr>
<tr>
<td>370</td>
<td>4</td>
<td>10</td>
<td>0.5</td>
<td>24</td>
<td>14</td>
<td></td>
<td>177</td>
<td>1</td>
<td>1</td>
<td>170</td>
</tr>
<tr>
<td>368</td>
<td>5</td>
<td>11</td>
<td>0.5</td>
<td>24</td>
<td>13</td>
<td></td>
<td>176</td>
<td>1</td>
<td>2</td>
<td>169</td>
</tr>
<tr>
<td>371</td>
<td>5</td>
<td>13</td>
<td>0.5</td>
<td>24</td>
<td>12</td>
<td></td>
<td>173</td>
<td>1</td>
<td>2</td>
<td>166</td>
</tr>
<tr>
<td>360</td>
<td>7</td>
<td>17</td>
<td>0.5</td>
<td>24</td>
<td>11</td>
<td></td>
<td>163</td>
<td>1</td>
<td>2</td>
<td>154</td>
</tr>
<tr>
<td>357</td>
<td>12</td>
<td>30</td>
<td>0.5</td>
<td>24</td>
<td>10</td>
<td></td>
<td>122</td>
<td>1</td>
<td>5</td>
<td>119</td>
</tr>
<tr>
<td>288</td>
<td>22</td>
<td>53</td>
<td>0.5</td>
<td>24</td>
<td>9</td>
<td></td>
<td>70</td>
<td>1</td>
<td>10</td>
<td>68</td>
</tr>
<tr>
<td>260</td>
<td>49</td>
<td>121</td>
<td>0.5</td>
<td>24</td>
<td>8</td>
<td></td>
<td>47</td>
<td>1</td>
<td>29</td>
<td>47</td>
</tr>
<tr>
<td>176</td>
<td>60</td>
<td>154</td>
<td>0.5</td>
<td>24</td>
<td>7</td>
<td></td>
<td>43</td>
<td>1</td>
<td>117</td>
<td>43</td>
</tr>
<tr>
<td>149</td>
<td>122</td>
<td>332</td>
<td>0.5</td>
<td>24</td>
<td>6</td>
<td></td>
<td>41</td>
<td>1</td>
<td>400</td>
<td>41</td>
</tr>
</tbody>
</table>

Table 6.1. The effect of hash size on the running time of aligning *B. suis* 1330 against the highly homologous genome of *B. suis* ATCC. The same parameters used in the alignment against a random genome were applied.
The same collision dynamics impact the alignment running time as the number of hash indices is progressively reduced below 11. Smaller hash sizes result in an increasing number of hash collisions, requiring the invocation of the approximate string-matching algorithm to evaluate the equality between hash keys. Commensurate with the hash sizes used by established \( k \)-mer based aligners, such as BLAST [52, 53], BLAT [18] and PatternHunter [54, 55], a lower bound threshold of approximately 11 indices is required to maintain the constant running time of a fuzzy \( k \)-mer alignment.

![Fig 6.3](image_url). Running times for the alignment of \( B. suis \) 1330 against the genome of \( B. suis \) ATCC. The genomes are highly homologous, with a DDH distance of 0.0861.

The kernel of the fuzzy \( k \)-mer model is best illustrated at low hash sizes, where a high number of collisions are generated in a map. For a given \( k \)-mer size, the fuzzy approach should result in fewer keys in a map than an exact matching seed of size \( k \), as approximately matching \( k \)-mers will be added to
the fuzzy set of a matching fuzzy hash key. Significantly however, the number of keys generated by the fuzzy \( k \)-mer will be larger than the number produced by an exact matching seed with the same number of hash indices, resulting in a reduction in the effective load factor \((\alpha')\) of a map. This is illustrated in Fig 6.4, where the numbers of fuzzy and exact-matching keys generated from the genome of \textit{B. suis} ATCC are compared. The tabulated results for this test are shown in Table 6.2.

\textbf{Fig 6.4.} The number of hash keys generated when the genome of \textit{B. suis} ATCC is loaded into a map at decreasing hash sizes. The area in yellow corresponds to the increased sensitivity of fuzzy \( k \)-mers over exact seeds for a 24-mer key. The area in green represents an increase in specificity over smaller exact-matching \( k \)-mers with the same number of hash indices. The blue line denotes the number of hash keys generated in both a hash map and a tree map.
<table>
<thead>
<tr>
<th>Hash Size</th>
<th>Fuzzy Map</th>
<th>Hash Map</th>
<th>Tree Map</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Fuzzy Seed</td>
<td>Keys</td>
<td>Time (s)</td>
</tr>
<tr>
<td>20</td>
<td>11211513</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>1121248</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>1910430</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>1909981</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>1908198</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>1906894</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>1898160</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>1890294</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>1823537</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1763727</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1400260</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>1183447</td>
<td>50</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>562460</td>
<td>62</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>378694</td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>105895</td>
<td>107</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.2. Comparison of load times and key number when the genome of *B. suis* ATCC is loaded into a map. The Levenshtein distance algorithm, with a $\beta$ cut-off threshold of 0.5, was used to compute $\mu_\alpha(S)$ for the fuzzy alignments.
Using an exact matching 24-mer seed resulted in the generation of $1.91 \times 10^6$ keys in both a hash map and a tree map. This figure represents an approximation of the $k$-spectrum set of the B. suis ATCC genome, as hash collisions will result in the bucket address of a hash key containing more than one $k$-mer. As illustrated in the Fig 6.4, a 24-mer fuzzy seed generates fewer keys, with the number decreasing rapidly as the hash size is reduced.

The area coloured in yellow in Fig 6.4 represents the increased sensitivity of a fuzzy 24-mer seed over an exact 24-mer seed, in the window of hash sizes from 10 to 13 bases. The increased sensitivity is due to approximately matching $k$-mers from the reference genome being added to the fuzzy set of an existing fuzzy hash key. Consistent with an exact matching seed, fuzzy $k$-mer operations are performed on discrete chunks of $k$-sized sequences, with the fuzzy approach permitting some variability in $k$-mer content. The area shaded in green corresponds to the level of increased specificity over an exact match seed with the same number of hash indices. As the complexity of a $k$-mer increases in the order of $4^k$ [79], an exact-matching $k$-mer at small hash sizes will generate a high number of collisions and an accompanying high number of candidate matches. It is precisely because it is more specific, that the fuzzy approach produces more keys than an exact-matching seed with the same number of hash indices.

It is important to note that the implementation of the fuzzy hash map in the Ferox prototype delegates the map operations to an underlying instance of the class `java.util.HashMap`. As a consequence of this delegation, the collision resolution mechanism used in the fuzzy $k$-mer approach is identical to that of a conventional hash map. Given the known $\Theta(1 + \alpha)$ average running time for insertion and search operations on a hash map [50], once a reasonable number of hash indices are used, the speed of an alignment will be determined primarily by the effective load factor ($\alpha'$). Aside from $\alpha'$, the only remaining variables that can influence the speed of insertion or search operations are the time complexity of the approximate string matching algorithm and the $\beta$ cut-off threshold. For a string similarity algorithm with a running time of $O(n^2)$,
the time complexity for a fuzzy $k$-mer insertion or search is $\Theta(1 + \alpha'(k^2))$, accounting for the constant time required to convert a hash key into a bucket index, followed by the average time required to perform a dynamic programming alignment against the set of fuzzy hash keys in the linked list at that index.

6.1.2 Effect of Reference Genome Size on Running Time

Arguably the most important feature of hash maps is the preservation of $O(1)$ average running time for a search operation, regardless of the number of entries in the map [50, 51]. Thus, an essential requirement of the fuzzy $k$-mer approach is to retain the speed of a search against an increasingly large map of fuzzy $k$-mers, while simultaneously permitting variability in the $k$-mer content of the search and insertion keys.

The effect of the map size on the running time of a fuzzy $k$-mer alignment is depicted in Fig 6.5 and tabulated in Table 6.3, where the genome of B.suis 1330 was aligned against a set of randomly generated genome sequences with sizes ranging from 1-20Mbps. A single 24-mer fuzzy seed of ###############************, with a $\beta$ cut-off threshold of 0.7 was configured with an instance of the FuzzyLevenshtein class for the comparison.

The results demonstrate that the running time for the fuzzy $k$-mer alignment remains relatively constant as the map size is increased from 1-20Mbps, with no major deviation from the $O(1)$ average running time of the hash map. Moreover, the alignment times for both the fuzzy and conventional hash maps are superior to the sub-linear search time of the tree map. Given that the same 24-mer was used for all three maps, the support for $k$-mer variability in a fuzzy seed has no significant impact on the running time of a search operation against an increasingly larger genome. For the results compiled in Table 6.3, the fuzzy $k$-mer alignment was on average 2.8 times slower than that of an exact-matching hash alignment. An exact-matching alignment using a tree map, with a guaranteed search time of $O(\log(n))$ [51], was on average 13.55 times slower than that achieved with a hash map.
Fig 6.5 also depicts a significant difference in the load times between all three approaches. As the size of the reference genome is increased, aside from the overhead associated with creating and increasingly larger $k$-spectrum, the tendency for a collision to occur rises, requiring both the fuzzy and conventional hash maps to engage their collision resolution mechanism. The sharp increase in the load time for the hash map at 12Mbps is probably due to the rehashing of the map as the number of occupied buckets in the data structure exceeds the product of the load factor and the current capacity the map. This technique is used by the `HashMap` class in the Java Collections Framework to incrementally expand the capacity of a map, from an initial $2^4 = 16$ buckets [194]. As rehashing does not affect the binary search tree used to store keys in a tree map, the load times shown for the tree map reflect the cost of inserting $k$-mers as nodes in a binary tree.

![Graph showing load and alignment times for different genome sizes](image)

**Fig 6.5.** Alignment of the 2.1Mbp genome of *B. suis* 1330 against a set of randomly generated reference sequences, ranging in size from 1-20Mbps. The load and alignment times for fuzzy, hash and tree maps are shown for $k = 24$. A fuzzy seed of $f(k) = 13h+11f$, with a $\beta$ cut-off threshold of 0.7, was used for the fuzzy $k$-mer alignments.
<table>
<thead>
<tr>
<th>Genome Size (MB)</th>
<th>Fuzzy Hash Map</th>
<th>Hash Map</th>
<th>Tree Map</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Memory (MB)</td>
<td>Load Time (s)</td>
<td>Align Time (s)</td>
</tr>
<tr>
<td>1</td>
<td>195</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>389</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>564</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>773</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>945</td>
<td>13</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>1122</td>
<td>16</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>1343</td>
<td>19</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>1521</td>
<td>24</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>1717</td>
<td>32</td>
<td>3</td>
</tr>
<tr>
<td>10</td>
<td>1889</td>
<td>31</td>
<td>4</td>
</tr>
<tr>
<td>11</td>
<td>2027</td>
<td>32</td>
<td>4</td>
</tr>
<tr>
<td>12</td>
<td>2206</td>
<td>37</td>
<td>5</td>
</tr>
<tr>
<td>13</td>
<td>2516</td>
<td>40</td>
<td>4</td>
</tr>
<tr>
<td>14</td>
<td>2709</td>
<td>45</td>
<td>4</td>
</tr>
<tr>
<td>15</td>
<td>2862</td>
<td>48</td>
<td>6</td>
</tr>
<tr>
<td>16</td>
<td>3038</td>
<td>52</td>
<td>6</td>
</tr>
<tr>
<td>17</td>
<td>3217</td>
<td>59</td>
<td>5</td>
</tr>
<tr>
<td>18</td>
<td>3334</td>
<td>62</td>
<td>6</td>
</tr>
<tr>
<td>19</td>
<td>3513</td>
<td>66</td>
<td>6</td>
</tr>
<tr>
<td>20</td>
<td>3689</td>
<td>71</td>
<td>7</td>
</tr>
</tbody>
</table>

Table 6.3. The effect of map size on the running time of aligning B. suis 1330 against a set of randomly generated genomes of sizes ranging from 1-20Mbp. An instance of the class FuzzyLevenshtein was used to compute the fuzzy set membership function.
6.1.3 Computational Overhead of KeyComparactor

Given its centrality to the fuzzy k-mer approach, an evaluation of the computational overhead associated with the use of an approximate string-matching algorithm for a k-mer search is essential. As discussed in Chapter 4, the collision resolution mechanism employed by a chained hash map is a two-step process, consisting of an initial hashing operation on a search key to determine a bucket index, followed by a comparison of the search key against the collection of existing keys in the linked list at that index. In the fuzzy k-mer model, the latter component of collision resolution is encapsulated in the proto-generic equals() method of FuzzyHashKey. Through the application of the strategy pattern [202], an instance of FuzzySeed can be configured with any implementation of the interface KeyComparactor, allowing a variety of string-similarity algorithms to be used with fuzzy hash keys.

To assess this aspect of fuzzy k-mers, a set of whole genomes was aligned and the number of generated keys computed, along with the load and alignment times achieved using different string similarity algorithms. The prototype was configured with different approximate string-matching algorithms and a 24-mer seed, for a range of hash sizes decreasing from 20 to 6 hash indices. The β cut-off threshold was held constant at 0.5 for all alignments. For the test with an exact-matching seed, the implementation of the equals() method of FuzzyHashKey was held constant, returning a Boolean false for every invocation, regardless of the content of the fuzzy component of a seed. This had the effect of forcing the collision detection mechanism of the hash map to compare each k-mer search key against all of the existing keys in the linked list at a bucket index, allowing a measure of the relative running time cost of using different similarity algorithms against a guaranteed \(O(1)\) operation. The results of tests for the alignment of B.suis 1330 against the genome of B.suis ATCC are shown Fig 6.6 and tabulated in Table 6.4.
Fig 6.6. Running times for different types of KeyComparator used to align B. suis 1330 against the genome of B. suis ATCC. The broken lines denote the load times for each approximate string-matching algorithm, with the full lines showing the running times for alignments.

As the two B. suis genomes are highly homologous, the strong sequence similarity should induce a high number of hash collisions for a $k$-mer search, with an accompanying escalation in the invocation of the string similarity algorithm. The tests demonstrate that, regardless of the string similarity algorithm used, the fuzzy approach generates more hash keys than an exact-matching seed with the same number of hash indices, resulting in a reduction in the effective load factor of the map. This effect is accentuated when the number of hash indices in a seed is reduced, as approximately matching $k$-mers will be grouped together as satellite data of a fuzzy hash key. The reduction in effective load factor has a direct impact on the running time of map operations, as an increase in the number of hash keys reduces the overall number of hash collisions. The reduced effective load factor also contributes to mitigating the computational overhead of executing a string similarity algorithm for each $k$-mer search operation on the map, as the $\Theta(1 + \alpha(|S|^{\alpha}))$
running time is controlled by the effective load factor ($\alpha'$), with both the fuzzy subsequence ($S$) of a k-mer and the running time ($n$) of the algorithm remaining constant for a given seed.

For hash sizes > 13, there is no significant variation in the running times of any of the dynamic programming algorithms used in the test. Although the alignment running times converge as the hash size increases, the likelihood of locating an exact or approximate k-mer match rapidly decreases. The opposite effect can be observed as the hash size is reduced below 11, indicating that, consistent with established best practice [18, 23, 53, 54, 56], the window from 10 to 13 indices contains the optimal hash size for a k-mer seed. The superior alignment time for Hamming distance is not surprising, as the algorithm has a time complexity of $O(n)$ [199], in contrast with the $O(n^2)$ running time of the other measures of string similarity used. This difference is less prominent for the alignment of B. suis 1330 against the randomly generated genome (Table V, Appendix B), as the k-mers from the reference sequence are well distributed throughout the buckets in the hash map, resulting in less collisions and thus fewer invocations of the string similarity algorithm.

Notwithstanding the large difference in time complexity between the $O(1)$, $O(n)$ and $O(n^2)$ implementation of an exact match, Hamming Distance and the remaining algorithms respectively, the difference in load and alignment times is not significant above 10 hash indices. The overhead associated with using the Damerau-Levenshtein algorithm over the Levenshtein implementation relates to the additional step required by the former to check for the transposition of adjacent nucleotides between two k-mers [204].
<table>
<thead>
<tr>
<th>Hash</th>
<th>Seed</th>
<th>Hamming</th>
<th>Levenshtein</th>
<th>Damerau Levenshtein</th>
<th>Smith Waterman</th>
<th>Exact Seed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Keys</td>
<td>$\alpha'$</td>
<td>$T$ (s)</td>
<td>Keys</td>
<td>$\alpha'$</td>
</tr>
<tr>
<td>20</td>
<td></td>
<td>1911557</td>
<td>1.01</td>
<td>4</td>
<td>1911513</td>
<td>1.01</td>
</tr>
<tr>
<td>19</td>
<td></td>
<td>1911298</td>
<td>1.01</td>
<td>4</td>
<td>1911248</td>
<td>1.01</td>
</tr>
<tr>
<td>18</td>
<td></td>
<td>1910559</td>
<td>1.01</td>
<td>4</td>
<td>1910430</td>
<td>1.01</td>
</tr>
<tr>
<td>17</td>
<td></td>
<td>1910157</td>
<td>1.01</td>
<td>4</td>
<td>1909981</td>
<td>1.01</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td>1908783</td>
<td>1.01</td>
<td>4</td>
<td>1908198</td>
<td>1.01</td>
</tr>
<tr>
<td>15</td>
<td></td>
<td>1907896</td>
<td>1.01</td>
<td>4</td>
<td>1906894</td>
<td>1.01</td>
</tr>
<tr>
<td>14</td>
<td></td>
<td>1902996</td>
<td>1.01</td>
<td>4</td>
<td>1898160</td>
<td>1.01</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td>1899385</td>
<td>1.01</td>
<td>5</td>
<td>1890294</td>
<td>1.02</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>1870331</td>
<td>1.03</td>
<td>5</td>
<td>1823537</td>
<td>1.05</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>1847924</td>
<td>1.04</td>
<td>7</td>
<td>1763727</td>
<td>1.09</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>1678528</td>
<td>1.15</td>
<td>12</td>
<td>1400260</td>
<td>1.37</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>1570251</td>
<td>1.23</td>
<td>25</td>
<td>1183447</td>
<td>1.63</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>1055447</td>
<td>1.82</td>
<td>38</td>
<td>562460</td>
<td>3.42</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>852880</td>
<td>2.26</td>
<td>106</td>
<td>376694</td>
<td>5.08</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>344437</td>
<td>5.59</td>
<td>104</td>
<td>105895</td>
<td>18.17</td>
</tr>
</tbody>
</table>

Table 6.4. Comparison of string similarity algorithms for *B. suis* ATCC. $\alpha'$ = effective load factor, $T$ = time. A $\beta$ cut-off threshold of 0.5 was used for all fuzzy alignments.
Similarly, the increase in running time of the Smith-Waterman at small hash sizes relates to the overhead associated with constructing a dynamic programming matrix to compute a maximum alignment score. The divergence in running time from that achieved by an exact-matching seed, although noteworthy, demonstrates that the use of approximate string matching in the fuzzy $k$-mer model does not place an excessive or punitive computation burden on sequence alignment.

6.1.4 Effect of $\beta$ cut-off Threshold on Running Time

The $\beta$ cut-off threshold acts as a constraint on the membership function of a fuzzy $k$-mer and, in addition to the size of the fuzzy component of a $k$-mer, effects both the sensitivity and specificity of an alignment. To test the overall effect of the $\beta$ cut-off threshold on the speed of a $k$-mer alignment, the genomes of $B. suis$ 1330 and $B. suis$ ATCC were aligned with the threshold scaled from $1 \times 10^{-7}$ to 0.99. In addition to measuring the speed of load and alignment times, the tests also computed the number of hash keys generated, the ratio of generated keys to the $k$-spectrum of the $B. suis$ ATCC genome and the effective load factor of the map. The results of these tests are compiled in Tables 6.5 and 6.6 and are also depicted graphically in Fig. 6.7.

When the reference genome is first parsed, the fuzzy hash map will initially contain a set of empty buckets. As keys are added to the map, collisions are induced, resulting in approximately matching $k$-mers being added to the fuzzy set of a matching fuzzy hash key. For the 24-mer fuzzy seed, with 11 hash indices, a $\beta$ cut-off threshold of $1 \times 10^{-7}$ will result in the creation of a single fuzzy set at each occupied bucket in the map, with the fuzzy set containing all $k$-mers that hash to that bucket index. A subsequent $k$-mer search during an alignment will therefore require a single execution of the `equals()` method at a bucket address to return all approximately-matching $k$-mers.
<table>
<thead>
<tr>
<th>β Cut-off threshold</th>
<th>Hamming Number of Keys</th>
<th>Load Time (s)</th>
<th>Align Time (s)</th>
<th>Levenshtein Number of Keys</th>
<th>Load Time (s)</th>
<th>Align Time (s)</th>
<th>Damerau Levenshtein Number of Keys</th>
<th>Load Time (s)</th>
<th>Align Time (s)</th>
<th>Smith Waterman Number of Keys</th>
<th>Load Time (s)</th>
<th>Align Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.99</td>
<td>1913040</td>
<td>6</td>
<td>7</td>
<td>1913040</td>
<td>13</td>
<td>28</td>
<td>1913040</td>
<td>14</td>
<td>33</td>
<td>1913040</td>
<td>13</td>
<td>29</td>
</tr>
<tr>
<td>0.9</td>
<td>1910802</td>
<td>6</td>
<td>7</td>
<td>1910802</td>
<td>13</td>
<td>28</td>
<td>1910757</td>
<td>14</td>
<td>32</td>
<td>1912684</td>
<td>14</td>
<td>30</td>
</tr>
<tr>
<td>0.8</td>
<td>1908921</td>
<td>6</td>
<td>7</td>
<td>1908809</td>
<td>12</td>
<td>27</td>
<td>1908723</td>
<td>14</td>
<td>32</td>
<td>1909868</td>
<td>13</td>
<td>29</td>
</tr>
<tr>
<td>0.7</td>
<td>1906944</td>
<td>6</td>
<td>7</td>
<td>1906210</td>
<td>12</td>
<td>28</td>
<td>1905820</td>
<td>14</td>
<td>32</td>
<td>1906555</td>
<td>14</td>
<td>31</td>
</tr>
<tr>
<td>0.6</td>
<td>1888109</td>
<td>6</td>
<td>7</td>
<td>1862579</td>
<td>12</td>
<td>27</td>
<td>1854496</td>
<td>14</td>
<td>32</td>
<td>1882710</td>
<td>13</td>
<td>30</td>
</tr>
<tr>
<td>0.5</td>
<td>1847924</td>
<td>6</td>
<td>7</td>
<td>1763727</td>
<td>12</td>
<td>27</td>
<td>1745850</td>
<td>14</td>
<td>32</td>
<td>1747961</td>
<td>14</td>
<td>30</td>
</tr>
<tr>
<td>0.4</td>
<td>1762860</td>
<td>6</td>
<td>7</td>
<td>1591415</td>
<td>10</td>
<td>24</td>
<td>1569512</td>
<td>14</td>
<td>32</td>
<td>1560641</td>
<td>14</td>
<td>30</td>
</tr>
<tr>
<td>0.3</td>
<td>1477751</td>
<td>5</td>
<td>8</td>
<td>1254417</td>
<td>9</td>
<td>23</td>
<td>1245608</td>
<td>10</td>
<td>27</td>
<td>1286396</td>
<td>9</td>
<td>25</td>
</tr>
<tr>
<td>0.2</td>
<td>1335752</td>
<td>5</td>
<td>8</td>
<td>1179869</td>
<td>9</td>
<td>24</td>
<td>1177622</td>
<td>10</td>
<td>27</td>
<td>1190557</td>
<td>9</td>
<td>25</td>
</tr>
<tr>
<td>0.1</td>
<td>1229128</td>
<td>5</td>
<td>9</td>
<td>1159819</td>
<td>9</td>
<td>25</td>
<td>1159380</td>
<td>10</td>
<td>27</td>
<td>1158288</td>
<td>9</td>
<td>26</td>
</tr>
<tr>
<td>0.0000001</td>
<td>1171812</td>
<td>5</td>
<td>9</td>
<td>1157463</td>
<td>9</td>
<td>24</td>
<td>1157456</td>
<td>10</td>
<td>27</td>
<td>1157357</td>
<td>10</td>
<td>26</td>
</tr>
</tbody>
</table>

Table 6.5. The effect of the β cut-off threshold on load and alignment times, for various types of KeyComparator. A single fuzzy seed of $f(k) = 11h + 13f$ was used to align the B.suis 1330 against the genome of B.suis ATCC.
<table>
<thead>
<tr>
<th>β Cut-off threshold</th>
<th>Hamming Keys (n)</th>
<th>n/G^k</th>
<th>α’</th>
<th>Levenshtein Keys (n)</th>
<th>n/G^k</th>
<th>α’</th>
<th>Damerau Levenshtein Keys (n)</th>
<th>n/G^k</th>
<th>α’</th>
<th>Smith Waterman Number of Keys (n)</th>
<th>n/G^k</th>
<th>α’</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.99</td>
<td>1913040</td>
<td>0.980</td>
<td>1.007</td>
<td>1913040</td>
<td>0.980</td>
<td>1.007</td>
<td>1913040</td>
<td>0.980</td>
<td>1.007</td>
<td>1913040</td>
<td>0.980</td>
<td>1.007</td>
</tr>
<tr>
<td>0.9</td>
<td>1910802</td>
<td>0.979</td>
<td>1.008</td>
<td>1910802</td>
<td>0.979</td>
<td>1.008</td>
<td>1910802</td>
<td>0.979</td>
<td>1.008</td>
<td>1912684</td>
<td>0.980</td>
<td>1.007</td>
</tr>
<tr>
<td>0.8</td>
<td>1908921</td>
<td>0.978</td>
<td>1.009</td>
<td>1908809</td>
<td>0.978</td>
<td>1.009</td>
<td>1908723</td>
<td>0.978</td>
<td>1.009</td>
<td>1909868</td>
<td>0.978</td>
<td>1.009</td>
</tr>
<tr>
<td>0.7</td>
<td>1906944</td>
<td>0.977</td>
<td>1.010</td>
<td>1906210</td>
<td>0.976</td>
<td>1.011</td>
<td>1905820</td>
<td>0.976</td>
<td>1.011</td>
<td>1906555</td>
<td>0.977</td>
<td>1.010</td>
</tr>
<tr>
<td>0.6</td>
<td>1888109</td>
<td>0.967</td>
<td>1.021</td>
<td>1862579</td>
<td>0.954</td>
<td>1.037</td>
<td>1854496</td>
<td>0.950</td>
<td>1.042</td>
<td>1882710</td>
<td>0.964</td>
<td>1.025</td>
</tr>
<tr>
<td>0.5</td>
<td>1847924</td>
<td>0.947</td>
<td>1.046</td>
<td>1763727</td>
<td>0.903</td>
<td>1.101</td>
<td>1745850</td>
<td>0.894</td>
<td>1.113</td>
<td>1747961</td>
<td>0.895</td>
<td>1.111</td>
</tr>
<tr>
<td>0.4</td>
<td>1762860</td>
<td>0.903</td>
<td>1.109</td>
<td>1591415</td>
<td>0.815</td>
<td>1.230</td>
<td>1569512</td>
<td>0.804</td>
<td>1.248</td>
<td>1560641</td>
<td>0.799</td>
<td>1.256</td>
</tr>
<tr>
<td>0.3</td>
<td>1477751</td>
<td>0.757</td>
<td>1.331</td>
<td>1254417</td>
<td>0.642</td>
<td>1.585</td>
<td>1245608</td>
<td>0.638</td>
<td>1.597</td>
<td>1286396</td>
<td>0.659</td>
<td>1.543</td>
</tr>
<tr>
<td>0.2</td>
<td>1335752</td>
<td>0.684</td>
<td>1.482</td>
<td>1179869</td>
<td>0.604</td>
<td>1.693</td>
<td>1177622</td>
<td>0.603</td>
<td>1.697</td>
<td>1190557</td>
<td>0.610</td>
<td>1.676</td>
</tr>
<tr>
<td>0.1</td>
<td>1229128</td>
<td>0.629</td>
<td>1.620</td>
<td>1159819</td>
<td>0.594</td>
<td>1.725</td>
<td>1159580</td>
<td>0.594</td>
<td>1.725</td>
<td>1158288</td>
<td>0.593</td>
<td>1.727</td>
</tr>
<tr>
<td>0.0000001</td>
<td>1171812</td>
<td>0.600</td>
<td>1.706</td>
<td>1157463</td>
<td>0.593</td>
<td>1.729</td>
<td>1157456</td>
<td>0.593</td>
<td>1.729</td>
<td>1157357</td>
<td>0.593</td>
<td>1.729</td>
</tr>
</tbody>
</table>

Table 6.6. The effect of the β cut-off threshold on the average size of a fuzzy set. α’ = effective load factor. The column heading n/G^k denotes the fraction of keys generated for the full k-spectrum of the B. suis ATCC reference genome.
While this decreases the running time for loading and alignment, it comes at the cost of specificity, as a 24-mer fuzzy seed with 11 hash indices and a β cut-off threshold of $1 \times 10^{-7}$ will have the same degree of specificity as an 11-mer exact seed. The converse applies when the β cut-off threshold is set too high. For the same 24-mer fuzzy seed, a β cut-off threshold of 0.99 reduces the sensitivity of an alignment to that of a full 24-mer exact seed. However, as the bucket index will be only computed from the same 11 hash indices, the high β cut-off threshold will result in the creation of more fuzzy sets. A subsequent $k$-mer insertion will thus require a comparison against all fuzzy hash keys at a given bucket index, with an accompanying invocation of the string-similarity algorithm. For this reason, the running time for an alignment increases as the β cut-off threshold is raised.

Fig 6.7. The multiplicity of fuzzy hash keys generated for the genome of B. suis ATCC at different β cut-off thresholds using alternative instances of KeyComparator.
This process is further illustrated in Table 6.6, where the fraction of hash keys to the $k$-spectrum of a genome falls from 0.980 to 0.593 as the $\beta$ cut-off threshold is reduced towards zero. Accompanying the reduction in the number of hash keys is an escalation in the number of collisions in the map, as the effective load factor increases 1.007 to 1.792. Once the reference genome has been loaded into a fuzzy hash map, a successful search will return the full fuzzy set of elements for the matching fuzzy hash key. For this reason, the load and alignment times improve as the value of $\beta$ is reduced.

A noticeable feature of the growth curves in Fig 6.7 is the degree of homogeneity for the family of dynamic programming algorithms used. In particular, the overall accordance in the number of keys generated by the various dynamic programming algorithms suggests that they broadly achieve the same degree of sensitivity and specificity. The linear-time implementation of Hamming Distance, while capable of accommodating polymorphisms in $k$-mer content, is intolerant of insertions and deletions. This reduced sensitivity of the Hamming Distance algorithm has the consequence of generating a larger number of hash keys, resulting in more fuzzy sets with a lower effective load factor. Clearly, applying the $\beta$ cut-off threshold to the fuzzy set membership function, $\mu_A(S)$, has a significant influence on the effective load factor of a map and consequently impacts on both the sensitivity and specificity of a fuzzy seed.

### 6.1.5 Impact of $k$-mer Size on Running Time

A key characteristic of the fuzzy $k$-mer model is its ability to process sequence data in $k$-sized chunks while simultaneously allowing a degree of variability in $k$-mer content. Although the primacy of hash size in determining the speed of an alignment has already been discussed, an evaluation of the limits of the fuzzy $k$-mer approach at larger sizes of $k$ is worthy of investigation. Although Kingsford et al [115] showed that sizes of $k > 35$ are unwarranted, testing the fuzzy $k$-mer model on large sized $k$-mers allows further investigation into the impact of the string similarity algorithm on
running time. For this study, $k$-mer sizes ranging from 24-mers to 104-mers were used to align \textit{B. suis} 1330 against \textit{B. suis} ATCC, using a set of different string similarity algorithms.

In addition to a consecutive fuzzy seed, the Ferox prototype supports both conventional spaced-seed and fuzzy spaced seeds, through the declaration of seeds using characters other than hash symbols and asterisks. For example, a 24-mer conventional spaced seed, with a weight of 11, can be created with the pattern \texttt{-##-##-##-##-##}, where hash symbols denote hash positions and dashes indicate “don’t care” indices. The 24-mer fuzzy spaced seed, \texttt{-##-##-##-##-##-##-##-##}, has a weight of 10 and contains five “don’t care” positions and nine “may care” fuzzy indices used by a \textit{KeyComparator}. For this test,uzzy seeds and fuzzy spaced seeds had prefixes of \texttt{##########-##-##-##-##} and \texttt{-##-##-##-##-##-##-##-##} respectively, with the number of fuzzy indices increased incrementally by 10 from 24-mer to 104-mer seeds. The $\beta$ cut-off threshold was fixed at 0.5 for all fuzzy seeds in this test. The initial spaced-seed, with a pattern of \texttt{-##-##-##-##-##-##}, was extended incrementally by repeating the pattern \texttt{-##}. The results of the tests are depicted in Fig 6.8 and are tabulated below in Table 6.7.

<table>
<thead>
<tr>
<th>$k$</th>
<th>Fuzzy $k$-mer</th>
<th>Fuzzy Spaced-Seed</th>
<th>Spaced-Seed</th>
<th>Hash Map</th>
<th>Tree Map</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
<td>13</td>
<td>7</td>
<td>4</td>
<td>3</td>
<td>15</td>
</tr>
<tr>
<td>34</td>
<td>22</td>
<td>14</td>
<td>4</td>
<td>3</td>
<td>15</td>
</tr>
<tr>
<td>44</td>
<td>36</td>
<td>24</td>
<td>5</td>
<td>3</td>
<td>15</td>
</tr>
<tr>
<td>54</td>
<td>54</td>
<td>40</td>
<td>5</td>
<td>3</td>
<td>15</td>
</tr>
<tr>
<td>64</td>
<td>76</td>
<td>59</td>
<td>5</td>
<td>4</td>
<td>17</td>
</tr>
<tr>
<td>74</td>
<td>105</td>
<td>80</td>
<td>5</td>
<td>4</td>
<td>18</td>
</tr>
<tr>
<td>84</td>
<td>136</td>
<td>110</td>
<td>5</td>
<td>4</td>
<td>20</td>
</tr>
<tr>
<td>94</td>
<td>171</td>
<td>173</td>
<td>5</td>
<td>4</td>
<td>17</td>
</tr>
<tr>
<td>104</td>
<td>212</td>
<td>215</td>
<td>5</td>
<td>4</td>
<td>17</td>
</tr>
</tbody>
</table>

\textbf{Table 6.7.} Running times (s) for different seed types at increasing $k$-mer size.
As the size of $k$ is increased, the running time of the fuzzy $k$-mer alignments escalates rapidly. This degradation in performance, shown in Fig. 6.8, is caused by the increasing time complexity of the Levenshtein algorithm, as the fuzzy component of the $k$-mer is extended. This contrasts with the linear running time, allowing for the computation of larger $k$-mer substrings, provided by the Hamming Distance algorithm. Indeed, the running times for both Hamming Distance and a conventional spaced seed are almost identical. This accordance reflects the similar time complexity required to generate a hash code from a spaced seed, by iterating over the length of a seed pattern and comparing the characters along the length of two $k$-mer strings.

The superior running time of the fuzzy spaced seed over the fuzzy seed containing consecutive hash indices indicates that the former seed type provides a greater degree of sensitivity. This is unsurprising; if we consider
fuzzy seeds as a superset of exact-matching seeds, then the proven sensitivity of exact-matching space seeds over consecutive seeds [54] must also hold. Although the running time for the hash map and tree map are vastly superior to the times obtained from the fuzzy approaches, this is counter-balanced by an almost complete loss of sensitivity as the size of $k$ is increased.

6.2 Sensitivity and Specificity

To evaluate the sensitivity and specificity of the fuzzy $k$-mer model, 800bp sequences at 10X coverage were extracted from a set of bacterial genomes and the index position of each sequence recorded. Random indels were then induced into the synthetic sequences at a rate of 10% and each set of sequences then aligned against its original genome. The a priori knowledge of the correct index of each sequence enabled the accurate evaluation of the fuzzy $k$-mer approach by comparing alignments with their correct position in the original genome. For these tests, only the highest scoring alignment for each sequence was analysed. A sequence alignment was considered a true positive (TP) if its alignment indices mapped exactly to their original position and a false positive (FP) if not. The level of accuracy was computed using the same approach described by Brundo and Morgenstein [72], i.e., sensitivity = $TP/(TP + FN)$ and specificity = $TP/(TP + FP)$.

6.2.1 Effect of $\beta$ cut-off Threshold on Sensitivity & Specificity

The $\beta$ cut-off threshold is a crucial parameter for controlling the admission of approximately matching $k$-mers to fuzzy sets, following the invocation of the fuzzy set membership function, $\mu_A(S)$. Although the sensitivity of a $k$-mer match is predicated, in the first instance, on matching hash codes, the $\beta$ cut-off threshold also affects both the sensitivity and the specificity of a fuzzy seed.

Table 6.8 shows the results of tests with synthetic sequences from the B.suis 1313 genome, using the single 24-mer fuzzy seed,
with 12 hash positions and a requirement for an overall percentage identity match of 60%. The fuzzy set membership function, $\mu_A(S)$, was computed using a variation of the Levenshtein distance algorithm. For this comparison, both BLAT [18] and Mosaik [102] were configured to use 12 hash positions per seed and a similar requirement for a 60% identity match along a query sequence.

<table>
<thead>
<tr>
<th>$\beta$</th>
<th>Align Time (s)</th>
<th>Sensitivity %</th>
<th>Specificity %</th>
<th>Keys</th>
<th>$\alpha^*$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.9</td>
<td>176</td>
<td>0.03</td>
<td>100.00</td>
<td>2089955</td>
<td>1.01</td>
</tr>
<tr>
<td>0.8</td>
<td>173</td>
<td>99.74</td>
<td>99.76</td>
<td>2087890</td>
<td>1.01</td>
</tr>
<tr>
<td>0.7</td>
<td>180</td>
<td>99.76</td>
<td>99.76</td>
<td>2085383</td>
<td>1.01</td>
</tr>
<tr>
<td>0.6</td>
<td>175</td>
<td>99.76</td>
<td>99.76</td>
<td>2077917</td>
<td>1.01</td>
</tr>
<tr>
<td>0.5</td>
<td>172</td>
<td>99.72</td>
<td>99.72</td>
<td>1984896</td>
<td>1.06</td>
</tr>
<tr>
<td>0.4</td>
<td>212</td>
<td>99.71</td>
<td>99.72</td>
<td>1880955</td>
<td>1.12</td>
</tr>
<tr>
<td>0.3</td>
<td>223</td>
<td>99.66</td>
<td>99.67</td>
<td>1778993</td>
<td>1.18</td>
</tr>
<tr>
<td>0.2</td>
<td>852</td>
<td>99.63</td>
<td>99.63</td>
<td>1717524</td>
<td>1.23</td>
</tr>
<tr>
<td>0.1</td>
<td>911</td>
<td>99.45</td>
<td>99.45</td>
<td>1698438</td>
<td>1.24</td>
</tr>
<tr>
<td>Exact 12-mer</td>
<td>90</td>
<td>92.73</td>
<td>92.74</td>
<td>1696065</td>
<td>1.24</td>
</tr>
<tr>
<td>Exact 24-mer</td>
<td>20</td>
<td>0.003</td>
<td>100.00</td>
<td>2092274</td>
<td>1.01</td>
</tr>
<tr>
<td>BLAT</td>
<td>302</td>
<td>99.38</td>
<td>98.67</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mosaik</td>
<td>342</td>
<td>99.96</td>
<td>100.00</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.8. The effect of the $\beta$ cut-off threshold sensitivity and specificity for an alignment of B. suis 1313. $\alpha^*$ = effective load factor. The fuzzy seed #\#\#\#\#\#\#\#\#\#\#\#\#\#\#\# with 12 hash positions and an overall sequence percentage identity of 60% was used. BLAT was configured with the parameters -tileSize=12 -oneOff=1 -minIdentity=60. Mosaik was executed with the switches -hs 12 -mm 80 -mmp 0.40 -mhp 100 -act 24 -bw 51.

In general, an increase in the $\beta$ cut-off threshold results in a corresponding increase in the level of specificity, with no loss of sensitivity below 0.8. Because it permits a full 24-mer approximate match, the fuzzy $k$-mer seed is significantly more sensitive and more specific than an exact matching 12-mer consecutive seed. Once the $\beta$ cut-off threshold is increased above 0.8, the sensitivity of the fuzzy approach is reduced to that of a full 24-mer exact-matching seed. Despite using a single 24-mer seed, the fuzzy approach is both more sensitive and more specific than the traditional “seed and extend”
approach used by BLAT, but less specific than the multiple spaced-seed strategy used by Mosaik.

The increased sensitivity and specificity over an exact-matching consecutive seed can be explained by the number of generated keys and by the effective load factor at different $\beta$ cut-off thresholds. At a threshold value of 0.1, an exact matching 12-mer seed and a fuzzy seed have approximately the same number of hash keys and a similar effective load factor ($\alpha'$). As $\alpha'$ represents the average number of approximately matching $k$-mers in a fuzzy set, the shared $\alpha'$ value of 1.24 implies a similar level of sensitivity and specificity. As the threshold is raised, the number of keys generated expands, reducing $\alpha'$ and increasing the sensitivity and specificity of a fuzzy match. Once the threshold increases beyond 0.8, the number of generated keys approaches the $k$-spectrum of the genome for an exact 24-mer seed, resulting in a sharp decline in sensitivity and an accompanying increase in specificity.

The same dynamics affect the sensitivity and specificity of fuzzy spaced seed alignments. Table 6.9 shows the results of a spaced seed alignment with the same synthetic sequences from the B. suis 1313 genome. For these tests, the original spaced seed used in PatternHunter [55], ###-##-##-##-###, with a seed weight of 12, was compared against the fuzzy spaced seed ###-##-##-##-##-##-###. The same fuzzy set membership function and overall percentage identity match used in the tests with a consecutive fuzzy seed were again applied. While the conventional spaced seed aligns more sequences than the fuzzy spaced seed with a $\beta$ cut-off threshold of 0.1, the fuzzy alignment contains a higher number of true positive matches and consequently a superior sensitivity and specificity. As the threshold is increased, due to the small number of fuzzy indices available, the sensitivity and specificity begin to taper off, before the former falls rapidly once a threshold of 0.6 is reached. Consistent with a consecutive fuzzy seed, the overall alignment running time is reduced significantly as the threshold is raised. This is further evidence of the primacy of key multiplicity and effective load factor on the running time of map operations.
Although the synthetic sequences used in the test allowed for an optimal estimation of a percentage identity match, in general, an optimal value will depend on a number of different factors. These include the degree of sequence similarity, the level of indels, repeats, structural re-arrangements and the error characteristics of the sequences.

### 6.2.2 Sensitivity & Specificity of Fuzzy Seeds

A further comparison of the running times, sensitivity and specificity of fuzzy k-mer alignments with those produced by BLAT and Mosaik is shown in Tables 6.10 and 6.11 and in Tables VII to IX in Appendix B. To test the specificity of a fuzzy seed with repeat-rich sequences, the genomes of Y. pestis CO92, N. meningitidis and M. flagellatus KT were included in the test.

<table>
<thead>
<tr>
<th>( \beta )</th>
<th>Number Aligned</th>
<th>TP</th>
<th>FN</th>
<th>FP</th>
<th>Align Time (s)</th>
<th>Sensitivity %</th>
<th>Specificity %</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.9</td>
<td>1317</td>
<td>1311</td>
<td>25037</td>
<td>6</td>
<td>36</td>
<td>4.98</td>
<td>99.54</td>
</tr>
<tr>
<td>0.8</td>
<td>3836</td>
<td>3820</td>
<td>22528</td>
<td>16</td>
<td>36</td>
<td>14.50</td>
<td>99.58</td>
</tr>
<tr>
<td>0.7</td>
<td>3837</td>
<td>3821</td>
<td>22527</td>
<td>16</td>
<td>37</td>
<td>14.50</td>
<td>99.58</td>
</tr>
<tr>
<td>0.6</td>
<td>21681</td>
<td>21619</td>
<td>4729</td>
<td>62</td>
<td>40</td>
<td>82.05</td>
<td>99.71</td>
</tr>
<tr>
<td>0.5</td>
<td>23514</td>
<td>23441</td>
<td>2907</td>
<td>73</td>
<td>44</td>
<td>88.97</td>
<td>99.69</td>
</tr>
<tr>
<td>0.4</td>
<td>23514</td>
<td>23441</td>
<td>2907</td>
<td>73</td>
<td>45</td>
<td>88.97</td>
<td>99.69</td>
</tr>
<tr>
<td>0.3</td>
<td>24190</td>
<td>24107</td>
<td>2241</td>
<td>83</td>
<td>64</td>
<td>91.49</td>
<td>99.66</td>
</tr>
<tr>
<td>0.2</td>
<td>24190</td>
<td>24107</td>
<td>2241</td>
<td>83</td>
<td>66</td>
<td>91.49</td>
<td>99.66</td>
</tr>
<tr>
<td>0.1</td>
<td>23602</td>
<td>23527</td>
<td>2821</td>
<td>75</td>
<td>71</td>
<td>89.29</td>
<td>99.68</td>
</tr>
<tr>
<td>Spaced Seed</td>
<td>24546</td>
<td>23054</td>
<td>3294</td>
<td>1492</td>
<td>71</td>
<td>87.50</td>
<td>93.92</td>
</tr>
<tr>
<td>Mosaik</td>
<td>26339</td>
<td>26339</td>
<td>9</td>
<td>0</td>
<td>342</td>
<td>99.97</td>
<td>100.00</td>
</tr>
</tbody>
</table>

**Table 6.9.** The effect of the \( \beta \) cut-off threshold on the sensitivity and specificity of a fuzzy spaced-seed. The fuzzy spaced-seed ####-###-###-###-## was used, with a percentage alignment match of 60%.
<table>
<thead>
<tr>
<th>Genome</th>
<th>Size (Mbp)</th>
<th>Number Sequences</th>
<th>12-mer Keys</th>
<th>α'</th>
<th>Fuzzy Keys</th>
<th>α'</th>
<th>Time (s)</th>
<th>Sn %</th>
<th>Sp %</th>
<th>BLAT</th>
<th>Time (s)</th>
<th>Sn %</th>
<th>Sp %</th>
<th>Mosaik</th>
<th>Time (s)</th>
<th>Sn %</th>
<th>Sp %</th>
</tr>
</thead>
<tbody>
<tr>
<td>S.epidermidis RP62A</td>
<td>2.61</td>
<td>32706</td>
<td>1889928</td>
<td>1.38</td>
<td>2324358</td>
<td>1.13</td>
<td>151</td>
<td>98.84</td>
<td>98.97</td>
<td>431</td>
<td>85.62</td>
<td>95.04</td>
<td>442</td>
<td>99.86</td>
<td>100.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E.coli 536</td>
<td>4.93</td>
<td>61736</td>
<td>3678092</td>
<td>1.34</td>
<td>4583586</td>
<td>1.08</td>
<td>254</td>
<td>99.29</td>
<td>99.55</td>
<td>546</td>
<td>84.56</td>
<td>97.11</td>
<td>697</td>
<td>99.82</td>
<td>100.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S.aureus COL</td>
<td>2.81</td>
<td>35118</td>
<td>2021176</td>
<td>1.39</td>
<td>2514925</td>
<td>1.12</td>
<td>162</td>
<td>99.40</td>
<td>99.51</td>
<td>412</td>
<td>86.43</td>
<td>96.58</td>
<td>412</td>
<td>99.88</td>
<td>100.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Synecococcus sp. WH8109</td>
<td>2.11</td>
<td>26864</td>
<td>1732851</td>
<td>1.22</td>
<td>2001675</td>
<td>1.06</td>
<td>86</td>
<td>99.16</td>
<td>99.72</td>
<td>211</td>
<td>83.94</td>
<td>97.90</td>
<td>310</td>
<td>99.67</td>
<td>100.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y.pestis CO92</td>
<td>4.65</td>
<td>58172</td>
<td>3542946</td>
<td>1.31</td>
<td>4250475</td>
<td>1.09</td>
<td>212</td>
<td>97.57</td>
<td>98.10</td>
<td>1123</td>
<td>84.79</td>
<td>79.78</td>
<td>1634</td>
<td>99.86</td>
<td>100.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>K.pneumoniae MGH-78578</td>
<td>5.31</td>
<td>66440</td>
<td>3532226</td>
<td>1.50</td>
<td>4678125</td>
<td>1.14</td>
<td>317</td>
<td>99.52</td>
<td>99.63</td>
<td>1147</td>
<td>84.10</td>
<td>98.20</td>
<td>810</td>
<td>99.82</td>
<td>100.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M.flagellatus KT</td>
<td>2.97</td>
<td>37144</td>
<td>2316507</td>
<td>1.28</td>
<td>2699868</td>
<td>1.10</td>
<td>120</td>
<td>90.32</td>
<td>90.66</td>
<td>344</td>
<td>71.00</td>
<td>76.83</td>
<td>431</td>
<td>99.82</td>
<td>100.00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N.meningitidis MC58</td>
<td>2.27</td>
<td>28404</td>
<td>1670515</td>
<td>1.36</td>
<td>1975995</td>
<td>1.15</td>
<td>138</td>
<td>91.46</td>
<td>91.88</td>
<td>1827</td>
<td>73.12</td>
<td>77.32</td>
<td>641</td>
<td>99.87</td>
<td>100.00</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.10. Comparison of alignments with BLAT and Mosaik. Sn = sensitivity, Sp = specificity. The fuzzy seed # repeating 12 hash positions, a β cut-off threshold of 0.5 and an overall sequence percentage identity of 90% was used for all fuzzy alignments. BLAT was configured with -tileSize=12 -oneOff=1 -minIdentity=90 and Mosaik with the parameters -hs 12 -mm 80 -mmp 0.10 -mhp 100 -act 24 -bw 51.
<table>
<thead>
<tr>
<th>Genome</th>
<th>Size (Mbp)</th>
<th>Number Sequences</th>
<th>Spaced Seed</th>
<th>Fuzzy Spaced Seed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Keys</td>
<td>α'</td>
</tr>
<tr>
<td>S.epidermidis RP62A</td>
<td>2.61</td>
<td>32706</td>
<td>1924875</td>
<td>1.36</td>
</tr>
<tr>
<td>E.coli 536</td>
<td>4.93</td>
<td>61736</td>
<td>3967701</td>
<td>1.24</td>
</tr>
<tr>
<td>S.aureus COL</td>
<td>2.81</td>
<td>35118</td>
<td>2078470</td>
<td>1.35</td>
</tr>
<tr>
<td>Synechococcus sp. WH8109</td>
<td>2.11</td>
<td>26864</td>
<td>1807800</td>
<td>1.17</td>
</tr>
<tr>
<td>Y.pestis CO92</td>
<td>4.65</td>
<td>58172</td>
<td>3722073</td>
<td>1.25</td>
</tr>
<tr>
<td>K.pneumoniae MGH1-78578</td>
<td>5.31</td>
<td>66440</td>
<td>3874102</td>
<td>1.37</td>
</tr>
<tr>
<td>M.flagellatus KT</td>
<td>2.97</td>
<td>37144</td>
<td>2428393</td>
<td>1.22</td>
</tr>
<tr>
<td>N.meningitidis MC58</td>
<td>2.27</td>
<td>28404</td>
<td>1831222</td>
<td>1.24</td>
</tr>
</tbody>
</table>

Table 6.11. Comparison of conventional and fuzzy spaced-seeds. The single spaced-seed, `###-###-###-###-###-###-###-###` and the fuzzy seed `###-###-###-###-###-###-###-###` were used for all alignments, with an overall percentage identity match of 90%. The Levenshtein distance algorithm, with a β cut-off threshold of 0.5 was used for the fuzzy k-mer alignments.
An analysis by Darling et al [192] of 9 genomes from the genus Yersinia attributed a substantial variation in genome arrangement, both within and across species, to the complex repeat structure of the genomes. In a separate study of the genomes of 58 bacteria, Achaz et al [193] singled out N. meningitidis as having a highly repetitive genome. A further analysis by Haubold and Wiehe [60] identified M. flagellatus KT as an extreme case of repetitive bacterial DNA, due to the presence of a large 143,034 bp tandem repeat sequence in the genome.

The results support the superiority of fuzzy k-mer seeds over the exact-matching consecutive seeds used by BLAT and the greater specificity of the multiple spaced-seed approach used by Mosaik. The fuzzy approach consistently generates a larger number of map keys than an exact-matching k-mer with the same number of hash indices. It is this increase in the multiplicity of keys and the accompanying reduction in effective load factor that provides fuzzy k-mers with greater sensitivity and specificity than exact-matching seeds.

The alignment times shown in Table 6.10 demonstrate that the execution of a string-similarity algorithm for every fuzzy k-mer search does not adversely impact overall running time. In addition to increased specificity, the reduction in effective load factor contributes to faster fuzzy search and insertion operations on a hash map. This is evidenced by the tests with Y. pestis, where the effective load factor was reduced from 1.31 to 1.09, enabling the overall alignment using fuzzy k-mers to be completed in 212 seconds, against the 1123 and 1634 seconds required by BLAT and Mosaik respectively. A similar effect can be observed for the alignment of N. meningitidis, where a decrease in the effective load factor from 1.36 to 1.15 allowed for an overall alignment time of 120 seconds. While both Ferox and BLAT show degradations in sensitivity and specificity for the N. meningitidis and M. flagellatus genomes, both aligners employed a single k-mer seed for the alignment, in contrast with the multiple spaced-seeds used by Mosaik.

The running times in Table 6.10 confirm that the computational burden required for an approximate k-mer match is not excessive or punitive, as the
difference in running times of the various fuzzy alignments is primarily due to the overhead of processing a larger number of query sequences. This is borne out by a comparison of the results for K. *pneumoniae* and S. *epidermidis*, where a similar effective load factor enabled more than twice the number of query sequences in the former to be aligned in approximately double the alignment time for the latter.

A comparison between a conventional and a fuzzy spaced seed using the same bacterial genome set is displayed in Table 6.11. Consistent with the results for consecutive fuzzy seeds, fuzzy spaced seeds generate a larger number of hash keys with an accompanying reduction in effective load factor compared to a conventional spaced seed. The results further demonstrate that the reduction in load factor improves the speed, sensitivity and specificity of alignments. Moreover, the results illustrate how the augmentation of conventional spaced seeds with fuzzy “may care” positions partially redresses the problem of low specificity in a single spaced seed identified by Li *et al* [54]. Although this issue can be addressed through the employment of multiple spaced seeds, as shown by Lin *et al* [56], this approach increases the overall space complexity for alignments, as a separate hash table is typically required for each seed to store matching *k*-mers in either the query or subject sequence.

### 6.3 Results with Real Data Sets

While the use of synthetic sequences enables an accurate assessment of the sensitivity and specificity of the fuzzy *k*-mer model, the use of randomly induced indels does not reflect the true error characteristics of real biological sequence data. This section evaluates the fuzzy *k*-mer approach using SGS and Sanger sequences that were aligned against a set of reference genomes. The results of these experiments, including a comparison with BLAT and Mosaik, are presented in Tables 6.12 and 6.13. To ensure an equitable comparison, each aligner was configured to use a seed with 12 hash positions and a requirement for an overall sequence percentage identity of 70%.
<table>
<thead>
<tr>
<th>Query</th>
<th>Reference</th>
<th>Size (Mbp)</th>
<th>DDH Distance</th>
<th>Sequence Length</th>
<th>Sequence Number</th>
<th>Coverage</th>
<th>Fuzzy k-mers</th>
<th>BLAT</th>
<th>Mosaik</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.suis 1330</td>
<td>B.suis 1330</td>
<td>2.1</td>
<td>0.0000</td>
<td>900</td>
<td>37948</td>
<td>16</td>
<td>1984896</td>
<td>1.06</td>
<td>176</td>
</tr>
<tr>
<td>E.coli 536</td>
<td>E.coli K-12 MG1655</td>
<td>4.63</td>
<td>0.2812</td>
<td>1009</td>
<td>64581</td>
<td>14</td>
<td>4307136</td>
<td>1.08</td>
<td>427</td>
</tr>
<tr>
<td>E.coli O157:H7</td>
<td>E.coli K-12 MG1655</td>
<td>4.63</td>
<td>0.2447</td>
<td>1064</td>
<td>56663</td>
<td>13</td>
<td>4307136</td>
<td>1.08</td>
<td>364</td>
</tr>
<tr>
<td>S.aureus COL</td>
<td>S.aureus COL</td>
<td>2.8</td>
<td>0.0000</td>
<td>200</td>
<td>483205</td>
<td>34</td>
<td>2514925</td>
<td>1.12</td>
<td>661</td>
</tr>
<tr>
<td>S.aureus JH1</td>
<td>S.aureus COL</td>
<td>2.8</td>
<td>0.1095</td>
<td>1050</td>
<td>40257</td>
<td>15</td>
<td>2514925</td>
<td>1.12</td>
<td>278</td>
</tr>
<tr>
<td>Shewanella sp.</td>
<td>S.oneidensis MR-1</td>
<td>4.96</td>
<td>0.7819</td>
<td>1016</td>
<td>60437</td>
<td>12</td>
<td>4524260</td>
<td>1.10</td>
<td>254</td>
</tr>
<tr>
<td>Synchococcus sp. CC9605</td>
<td>Synchococcus sp. CC9902</td>
<td>2.23</td>
<td>0.9646</td>
<td>969</td>
<td>44022</td>
<td>19</td>
<td>2173361</td>
<td>1.03</td>
<td>108</td>
</tr>
<tr>
<td>Synchococcus sp. WH8109</td>
<td>Synchococcus sp. CC9902</td>
<td>2.23</td>
<td>0.9626</td>
<td>990</td>
<td>24841</td>
<td>11</td>
<td>2173361</td>
<td>1.03</td>
<td>65</td>
</tr>
</tbody>
</table>

Table 6.12. Results with real data from Sanger sequencing instruments, showing the running time (T) and the percentage of reads aligned. The sequence length shown denotes the average read length in a sequence set. The column “DDH Aligned %” shows the percentage of sequences aligned when the percentage alignment match was altered to reflect the DDH distance. The percentage of sequences aligned by Mosaik was adjusted to compensate for the deletion of reads by the MosaikBuild utility. A hash size of 12 was used with all aligners for this comparison. A single 24-mer fuzzy seed, with a β cut-off threshold of 0.5 and a percentage alignment match of 70% was used the fuzzy k-mer alignments. BLAT was configured with the parameters -tileSize=12 -oneOff=1 -minIdentity=70 and Mosaik with the switches -hs 12 -mm 80 -minP 0.70 -mhp 100 -act 24 -bw 51.

All sequence sets were downloaded from the NCBI Trace Archive. The Trace Identifiers are as follows: B.suis 1330 (TI Range: 185503887-332240859), E.coli 536 (TI Range: 987733769-987798349), E.coli O157:H7 (TI Range: 1749497076-1749553738), S.aureus COL (TI Range: 1261839613-1261995093), S.aureus JH1 (TI Range: 515171320-1800402094), Shewanella sp. ANA-3 (TI Range: 2222700921-2222725761), Synchococcus sp. CC9605 (TI Range: 774963613-775015276), Synchococcus sp. WH8109 (TI Range: 1055305923-1055478098).
The fuzzy $k$-mer seed was configured to use the Levenshtein distance algorithm to compute $\mu_A(S)$, with the $\beta$ cut-off threshold held constant at 0.5.

Table 6.12 presents the alignment results for 8 sets of Sanger sequences, with sequence lengths ranging from 200 to 1064bps and varying levels of sequence coverage. The running times indicate that the fuzzy $k$-mer approach outperforms the speed of BLAT and is broadly similar to the alignment speed of Mosaik. The slow alignment time of BLAT is, in all likelihood, caused by the computational burden of performing a full Smith-Waterman alignment on the sequence substring between each high-scoring pair of $k$-mer seeds. The difference in running time between the various fuzzy alignments is primarily due to the overhead associated with processing a greater multiplicity of sequence data. This is evidenced by the alignment times of B. suis 1330 and E. coli 536 where, with approximately similar effective load factors, 37,948 B. suis sequences, with an average length of 900bps, were aligned in 176 seconds, compared to the 427 seconds required to align the 64,581 sequences, from E. coli 536, with an average length of 1064bps.

In the context of the percentage of sequences aligned, BLAT consistently outperformed both the fuzzy approach and the multiple spaced-seed strategy used by Mosaik. In particular, the high percentage of sequences aligned for the tests with Shewanella sp. ANA-3, Synechococcus sp. CC9605 and Synechococcus sp. WH8109 are at considerable variance with the very poor results for both the Ferox prototype and Mosaik. While it may be argued that the high performance of BLAT is due to a wider application of the Smith-Waterman algorithm and a robust accommodation of sequence read errors, a more cogent explanation can be gleaned from an examination of the DDH distances between the query and reference genomes. In all three cases the DDH distances and overall percentage identity indicate that the query and reference sequences are highly dissimilar, ranging from 0.7819 (21.81%) for Shewanella to 0.9646 (3.54%) and 0.9626 (3.74%) for Synechococcus sp. CC9605 and WH8109 respectively. With a requirement for an overall percentage identity match of 70%, the degree of dissimilarity between the three query and reference sequences explains the poor alignment results for the fuzzy approach and Mosaik and cast doubt on the accuracy of the
alignments reported by BLAT. This is born out by the alignment times depicted under the column “Aligned (DDH) %” in Table 6.12 that show the percentage of sequences aligned when the requirement for an overall percentage identity was modified to reflect the DDH distance. These results are a significant improvement on the original alignments and are consistent with the expected results for the given DDH distance.

The fuzzy $k$-mer approach was also tested with real SGS data extracted from the Sequence Read Archive. A total of 7 sets of SGS sequence data were aligned, consisting of 5 different strains of $S.\ aureus$ and sequences from $B.\ suis$ and $S.\ elongatus$. As noted by Henson et al [25], SGS sequences vary significantly from those produced by Sanger instruments, with the former having a much higher coverage depth, a shorter read length and different error characteristics. The results of the tests with SGS data, from Illumina and 454 sequencing platforms, are presented in Table 6.13. The same seeding parameters used for the alignment of Sanger sequences were again applied for these tests.

Consistent with the results obtained utilising Sanger sequences, the fuzzy approach is significantly faster than the “seed and extend” alignments computed by BLAT, but has a slower alignment time than that of Mosaik. The percentage of $B.\ suis$ 019 and $S.\ aureus$ USA-300 sequences aligned is broadly similar for all three aligners. This is not surprising as the genomes of $B.\ suis$ 019 and $B.\ suis$ 1330 have a DDH distance 0.2399, with those of $S.\ aureus$ even closer at 0.0688, both well within the overall percentage identity constraint. The results for $S.\ elongatus$ are less clear. Although the fuzzy approach outperformed both BLAT and Mosaik for the $S.\ elongatus$ alignment, with a DDH distance of just 0.0216, the strong homology between the two genomes should give rise to a very high percentage of aligned sequences. In the absence of any other evidence, the relatively poor alignment results for $S.\ elongatus$ may be attributed to a higher sequencing error rate, reducing the number of putative $k$-mer matches along the short 32bp length of each Illumina sequence. A similar explanation can be proffered for the alignment of $S.\ aureus$ ATCC BAA-39 which has a DDH distance of just 0.0426 from $S.\ aureus$ COL.
<table>
<thead>
<tr>
<th>Query</th>
<th>Reference</th>
<th>Size (Mbp)</th>
<th>Sequencing Platform</th>
<th>Sequence Length</th>
<th>Sequence Number</th>
<th>Coverage</th>
<th>Keys</th>
<th>α'</th>
<th>Time (s)</th>
<th>Aligned %</th>
<th>BLAT</th>
<th>Time (s)</th>
<th>Aligned %</th>
<th>Mosaik</th>
<th>Time (s)</th>
<th>Aligned %</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>B. suis</em> 019</td>
<td><em>B. suis</em> 1330</td>
<td>2.1</td>
<td>Illumina</td>
<td>80</td>
<td>4125000</td>
<td>157</td>
<td>1984896</td>
<td>1.06</td>
<td>1640</td>
<td>65.13</td>
<td>2595</td>
<td>64.96</td>
<td>359</td>
<td>64.95</td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>S. aureus</em> USA300</td>
<td><em>S. aureus</em> COL</td>
<td>2.8</td>
<td>Illumina</td>
<td>36</td>
<td>2739566</td>
<td>35</td>
<td>2514925</td>
<td>1.12</td>
<td>225</td>
<td>77.53</td>
<td>683</td>
<td>72.63</td>
<td>125</td>
<td>74.78</td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>S. elongatus</em> PCC 7942</td>
<td><em>S. elongatus</em> PCC 6301</td>
<td>2.7</td>
<td>Illumina</td>
<td>36</td>
<td>22932111</td>
<td>302</td>
<td>2551498</td>
<td>1.06</td>
<td>1978</td>
<td>54.62</td>
<td>4315</td>
<td>40.70</td>
<td>688</td>
<td>42.82</td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>S. aureus</em> 1185</td>
<td><em>S. aureus</em> COL</td>
<td>2.8</td>
<td>454</td>
<td>563</td>
<td>273744</td>
<td>55</td>
<td>2514925</td>
<td>1.12</td>
<td>825</td>
<td>83.87</td>
<td>3065</td>
<td>93.46</td>
<td>404</td>
<td>65.94</td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>S. aureus</em> 7211</td>
<td><em>S. aureus</em> COL</td>
<td>2.8</td>
<td>454</td>
<td>551</td>
<td>168648</td>
<td>33</td>
<td>2514925</td>
<td>1.12</td>
<td>441</td>
<td>79.49</td>
<td>1497</td>
<td>87.16</td>
<td>231</td>
<td>63.25</td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>S. aureus</em> ATCC BAA-39</td>
<td><em>S. aureus</em> COL</td>
<td>2.8</td>
<td>454</td>
<td>469</td>
<td>242521</td>
<td>40</td>
<td>2514925</td>
<td>1.12</td>
<td>539</td>
<td>57.45</td>
<td>1463</td>
<td>93.36</td>
<td>250</td>
<td>29.17</td>
<td></td>
<td></td>
</tr>
<tr>
<td><em>S. aureus</em> MRSA-TCH60</td>
<td><em>S. aureus</em> COL</td>
<td>2.8</td>
<td>454</td>
<td>276</td>
<td>395462</td>
<td>39</td>
<td>2514925</td>
<td>1.12</td>
<td>571</td>
<td>88.21</td>
<td>1133</td>
<td>93.52</td>
<td>226</td>
<td>89.14</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.13. Results with real data from Illumina and 454 sequencing instruments, showing the running time (T) and the percentage of reads aligned. The percentage of sequences aligned by Mosaik was adjusted for compensate to the deletion of reads by the *MosaikBuild* utility. A hash size of 12 was used with all aligners for this comparison. A single 24-mer fuzzy seed, #FFFFFFFFFFFFFFFFF, with a β cut-off threshold of 0.5 and a percentage alignment match of 70% was used for the fuzzy k-mer alignments. BLAT was configured with the parameters -tileSize=12 -oneOff=1 -minIdentity=70 and Mosaik with the switches -hs 12 -mm 80 -minp 0.70 -mhp 100 -act 24 -bw 13.

All sequence sets were downloaded from the NCBI Short Read Archive. The Sequence Identifiers are as follows: *B. suis* 019 (SRX134780), *S. aureus* USA300 (SRX007668), *S. elongatus* PCC 7942 (SRP003368), *S. aureus* 1185 (SRX025925), *S. aureus* 7211 (SRR074251.1), *S. aureus* ATCC BAA-39 (SRX030107), *S. aureus* MRSA-TCH60 (SRX003764)
The results with both Sanger and SGS sequences are broadly consistent with those obtained from synthetic data and demonstrate that the alignment speed of the fuzzy k-mer model is significantly faster than the alternative “seed and extend” approach used by BLAT, but slower for short length reads than the multiple spaced-seed strategy employed by Mosaik. Moreover, the sensitivity of the fuzzy k-mer model is corroborated by the use of a single 24-mer fuzzy seed to align all the read sets used in the tests with real sequence data. Furthermore, provided a suitable overall percentage identity match is specified, the fuzzy approach works well for both short and long sequences, as the tests included real sequence data with lengths ranging from 32 to 1064bps.

6.4 Chapter Summary

The fuzzy k-mer approach enables approximate k-mer matching, without significantly impacting on the overall running time of sequence alignment. By manipulating the collision detection mechanism and reducing the effective load factor of a hash map, fuzzy k-mers can combine the sensitivity of small k-mer seeds with the speed and specificity of larger seeds. The fuzzy k-mer approach is highly flexible and can be extended to include established models of consecutive and spaced seeds.

The results presented in this chapter demonstrate that the fuzzy k-mer approach is both fast and accurate with the running time of alignments matching those obtained using established exact k-mer aligners. Moreover, the results illustrate that the fuzzy approach is both more sensitive and more specific than an exact-matching consecutive seed alignment and can achieve, using a single fuzzy seed, almost the same specificity and sensitivity as a multiple spaced-seed alignment.
Chapter 7

Evaluation of Integrated Comparative Assembly

Current models of de novo genome assembly are characterised by the merging of unambiguous nodes in an assembly graph up to the boundaries of repeats [4]. In both the Overlap-Layout-Consensus [118] and de Bruijn graph [78, 142] models, contigs are then generated from the set of merged nodes. Assembled contigs are typically processed further, in a subsequent scaffolding phase, through the judicious application of paired-read constraints and long-distance positional information [69]. In contrast with these models, the comparative approaches to genome assembly, described in Chapter 2, are dominated by the Alignment-Layout-Consensus approach [68], where the overlap phase of assembly is supplanted completely by a faster local alignment of sequence reads against a reference genome. These comparative approaches however, assume both a strong homology and a high degree of synteny between the genome being assembled and the reference sequence. Consequently, current comparative assembly approaches are adversely affected by factors such as weak homology, repetitive regions and structural variations in the reference genome.
This chapter presents the results of benchmarks and tests on the assembly module of the Ferox prototype. In contrast with existing approaches to comparative assembly, alignment and anchoring information is directly incorporated into the assembly model of the prototype, enabling the extension of contigs through repeat boundaries. The prototype was benchmarked using synthetic error-free paired-end reads, extracted from a set of candidate genomes at 10X coverage, corresponding to a 99.995% sampling of each genome [62]. Sequence reads of 800bps, 400bps and 80bps were used to test the prototype, with the varying read lengths corresponding to Sanger, 454 and Illumina sequencing platforms.

The tests also included the benchmarking of the prototype against the preeminent implementations of de novo and comparative assembly, capable of utilising both Sanger and SGS sequence reads. Cabog 7.0 [34], Velvet 1.2.06 [49] and AMOS 3.0.1 [68, 210] were selected as reference implementations of the de Bruijn graph, Overlap-Layout-Consensus and Alignment-Layout-Consensus assembly models respectively. While also capable of comparative assembly, Mosaik 1.1 [102] is limited to genome re-sequencing, as reference sequences that align with sequence reads are also assembled into contigs. Consequently, Mosaik was not included in the evaluation. An alternative comparative assembler described by Peng et al [146, 211] was also excluded from the comparison. Despite its similarity with the prototype, in combining de Bruijn graph and comparative assembly, the IDBA-Hybrid assembler consistently exited with fatal errors, even when using the sample data sets provided with the assembler.

The N50 contig metric [131] was used as a standard to compare the output of the different assemblers. The assemblies produced were also subjected to a quality analysis, using the QUAST tool described by Gurevich et al [179]. In addition, MUMmer [16, 80] was used to generate dot-plots of assembled contigs against the original genome, enabling a visual inspection of the quality of the contig sets generated by the different assemblers.

The results presented in this chapter were compiled from executing the suite of assemblers on an OSX 10.6.8 platform, with a single 3.2 GHz Intel Core i3
processor and 16GB of RAM. The assembly module of the Ferox prototype required the additional provisioning of a Java HotSpot 1.6 64-bit virtual machine.

7.1 Genome Selection and Anchor Extraction

The 11 bacterial genomes used in the evaluation are shown in Table 7.1. The genomes of B. suis, E. coli, S. aureus and S. epidermidis were also used by Narzisi and Misra [57] in a recent comparison of de novo assemblers. The remaining genomes were selected on the basis of the DDH distance [154, 155] and variation in structural arrangement between the genome being assembled and the reference sequence. While the DDH metric indicates that the genomes of C. pneumoniae CWL029 and J138 are highly homologous, K. pneumonia and S. typhimurium have a DDH distance of at least 0.89 from their reference sequences, with both genomes also containing significant inversions.

<table>
<thead>
<tr>
<th>Query Genome</th>
<th>Query Size (Mbp)</th>
<th>Reference Genome</th>
<th>Reference Size (Mbp)</th>
<th>Anchors Extracted</th>
<th>Anchored (%)</th>
<th>DDH Distance</th>
<th>DDH % ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>B. suis 1330</td>
<td>2.10</td>
<td>B. suis ATCC 25445</td>
<td>1.92</td>
<td>164</td>
<td>97.51</td>
<td>0.0861</td>
<td>91.39</td>
</tr>
<tr>
<td>C. pneumoniae CWL029</td>
<td>1.23</td>
<td>C. pneumoniae J138</td>
<td>1.22</td>
<td>21</td>
<td>97.47</td>
<td>0.0260</td>
<td>97.40</td>
</tr>
<tr>
<td>E. coli 536</td>
<td>4.93</td>
<td>E. coli K-12 MG1655</td>
<td>4.63</td>
<td>354</td>
<td>95.62</td>
<td>0.2812</td>
<td>71.88</td>
</tr>
<tr>
<td>K. pneumoniae</td>
<td>5.31</td>
<td>E. coli K-12 MG1655</td>
<td>4.63</td>
<td>354</td>
<td>95.62</td>
<td>0.9265</td>
<td>7.35</td>
</tr>
<tr>
<td>M. genitalium</td>
<td>0.58</td>
<td>M. genitalium</td>
<td>0.81</td>
<td>67</td>
<td>83.89</td>
<td>0.9795</td>
<td>2.05</td>
</tr>
<tr>
<td>S. aureus COL</td>
<td>2.80</td>
<td>S. aureus COL</td>
<td>2.90</td>
<td>169</td>
<td>92.88</td>
<td>0.1095</td>
<td>89.05</td>
</tr>
<tr>
<td>S. epidermidis RP62A</td>
<td>2.61</td>
<td>S. epidermidis ATCC 12228</td>
<td>2.49</td>
<td>109</td>
<td>94.49</td>
<td>0.1335</td>
<td>86.63</td>
</tr>
<tr>
<td>S. maltophilia</td>
<td>4.96</td>
<td>S. maltophilia</td>
<td>4.97</td>
<td>171</td>
<td>96.00</td>
<td>0.7819</td>
<td>21.81</td>
</tr>
<tr>
<td>S. typhimurium</td>
<td>4.85</td>
<td>E. coli K-12 MG1655</td>
<td>4.63</td>
<td>354</td>
<td>95.62</td>
<td>0.8946</td>
<td>10.54</td>
</tr>
<tr>
<td>Synechococcus sp. WH-8109</td>
<td>2.11</td>
<td>Synechococcus sp. CC902</td>
<td>2.23</td>
<td>34</td>
<td>98.02</td>
<td>0.9626</td>
<td>03.74</td>
</tr>
<tr>
<td>Y. pestis CO92</td>
<td>4.65</td>
<td>Y. pestis CO92</td>
<td>4.60</td>
<td>421</td>
<td>92.12</td>
<td>0.1763</td>
<td>82.37</td>
</tr>
</tbody>
</table>

Table 7.1. Query and reference genomes using for benchmarking assembly.
Despite having one of the smallest bacterial genomes, \textit{M. genitalium} was also included in the evaluation, as the genome size facilitated a manual debug of an assembly graph. Both Synechococcus WH8109 and \textit{S. oneidensis} have a high DDH distance from their reference sequence and also contain significant inversions. \textit{Y. pestis} COL was selected due to its complex repeat structure and the substantial variation in genome arrangement between it and \textit{Y. pestis} KIM [192].

Unique 24-mer merged anchoring sequences, with a minimum length of 1000bps, were extracted from the set of reference genomes using the technique described in Chapter 5 (Section 5.3.2). An even value of $k$ was chosen to ensure that the set of anchors produced excluded palindromic sequences. As an anchor sequence is logically equivalent to a unitig, the percentage of the reference genome extracted as anchors, shown in Table 7.1, is indicative of the level of repetitive $k$-mers that the genome contains.

### 7.2 Comparison of Assembly Results

The results presented in this section were produced from the assembly of synthetic paired reads, generated from each of the query genomes listed in Table 7.1. Sequence read pairs were extracted from simulated insert libraries of 2000, 1200 and 200bps for read lengths of 800, 400 and 80 bps respectively. Synthetic shorter 36bp Illumina lengths were not used in the evaluation, as Cabog imposes a minimum read length constraint of 64bps.

A single 25-mer fuzzy seed, "###############************", with 12 hash positions was used for all the assemblies produced by the Ferox prototype. The fuzzy seed was configured with an implementation of the Levenshtein distance algorithm and a $\beta$ cut-off threshold of 0.8. The DDH distance metric was used to compute the percentage identity match required to anchor a sequence read against the reference genome.

In addition to a set of sequence reads, Cabog [34] requires an input file providing a quality score for each sequence base and another file describing
how to link paired-reads. The former was generated for each set of sequence reads, where a Phred quality score of 40, corresponding to 99.99% accuracy, was given to each base [212]. For each Cabog assembly, the FASTA file of sequence reads was converted into Celera format using the `fastaToCA` utility, with the `–mean` and `–stddev` parameters set to the sequence read length and 0.1 respectively. After executing each Cabog assembly with the `runCA` command, the `parsecas`m and `scaff2fasta` utilities, provided with AMOS, were used to extract the contigs and scaffolds from the ASM assembly file. The N50 metric was computed for both the assembled and scaffolded contigs produced by Cabog.

The sequence reads used by Velvet [49] were prepared for assembly using the `velveth` tool and a k-mer size of 25. The `velveth` utility was parameterised with `–longPaired` and `–long` to pre-process Sanger length sequence reads, with the `–shortPaired` switch used for the 80bps sequences. The core `velvetg` assembler was configured with the `–exp_cov` parameter set to 10.0 and an `–ins_length` reflecting the size of the synthetic insert library used.

The AMOS comparative assembler uses MUMmer [16, 80] to compute alignments between sequence reads and a reference genome. It then employs the Minumus [140] assembler to perform a layout of contigs before scaffolding with Bambus [110, 138]. The `AMOScmp` tool was used for a comparative assembly of the 400 and 800bp sequence reads. The shorter 80bp reads were assembled with the `AMOScmp-shortReads` utility, which uses a smaller Nucmer alignment cluster size of 20bp, as opposed to the default size of 65bp used for longer reads. While the modules used in each phase of the `Alignment–Layout–Consensus` approach are configurable, the recommended default parameters were used for the AMOS assemblies, after providing the necessary information required to link paired reads.

In addition to computing the N50 contig size, the contig sets produced by the different assemblers were subject to a quality analysis using the QUAST tool [179]. This was achieved by comparing each set of contigs assembled from synthetic reads against the original genome that they were extracted from. As noted by Narzisi and Mishra [57], the N50 metric emphasises contig size and
is a poor metric for capturing contig quality. An aggressive or greedy assembly strategy may produce a large N50 value, but may also increase the number of misassemblies in contigs. Conversely, a conservative approach to contig extension may produce a higher quality assembly, but with a lower N50 value. Alternative assembly metrics such as those described by Vezzi et al [59], Salzberg et al [10] and, more recently by Gurevich et al [179] emphasis both the size and quality of contigs. In addition to the N50 contig size, QUAST computes an NG50 metric using a reference genome, where the lengths of aligned blocks are counted instead of contig lengths [179]. If a contig contains a misassembly with respect to the reference, the contig is broken into smaller sequences and the N50 value recomputed. QUAST detects a relocation misassembly if the left flanking sequence of a contig overlaps or aligns more than 1Kbps away from the right flanking sequence on a reference genome. An inversion misassembly is detected where flanking sequences align on different strands of each genome.

A summary of the assembly results is presented in Tables 7.2 – 7.7. The expected number of contigs was computed using the formulae provided by Lander and Waterman [62] and assume an overlap of 24 bases. The number of misassemblies, M, the percentage of the genome assembled and the NG50 metric were computed by QUAST. Note that the results for the Cabog assemblies contain both an N50 and an S-N50 value, where the latter refers to the size of scaffolded contigs.

### 7.2.1 Assembly Running Time

In the context of running time, Velvet out-performed the other assemblers used at all read lengths. As Velvet was originally designed to assemble high volumes of short-length Illumina reads, processing sequence reads at 10X coverage posed little difficulty. The fast assembly time for the AMOS assembly of long reads is not surprising, as the Alignment-Layout-Consensus approach does not require a computationally expensive overlap phase. The larger running times for the 80bp reads are due to the smaller cluster size used by the AMOScmp-shortReads utility to increase the sensitivity of alignments.
<table>
<thead>
<tr>
<th>Organism</th>
<th>Size (Mbp)</th>
<th>Reads</th>
<th>Expected Contigs</th>
<th>Ferox Time (h)</th>
<th>Ferox Aligned (%)</th>
<th>Ferox Contigs</th>
<th>Ferox N50</th>
<th>Amos Time (h)</th>
<th>Amos Contigs</th>
<th>Amos N50</th>
<th>Velvet Time (h)</th>
<th>Velvet Contigs</th>
<th>Velvet N50</th>
<th>Cabog Time (h)</th>
<th>Cabog Contigs</th>
<th>Cabog N50</th>
<th>Cabog S-Contigs</th>
<th>Cabog S-N50</th>
</tr>
</thead>
<tbody>
<tr>
<td>B. subtilis 1330</td>
<td>2.10</td>
<td>26347</td>
<td>2</td>
<td>217</td>
<td>84.53</td>
<td>15</td>
<td>243423</td>
<td>49</td>
<td>12</td>
<td>318321</td>
<td>17</td>
<td>88</td>
<td>229505</td>
<td>1848</td>
<td>26</td>
<td>208800</td>
<td>22</td>
<td>267763</td>
</tr>
<tr>
<td>C. pneumoniae CWL029</td>
<td>1.23</td>
<td>15378</td>
<td>1</td>
<td>124</td>
<td>97.44</td>
<td>2</td>
<td>1211878</td>
<td>48</td>
<td>8</td>
<td>619668</td>
<td>9</td>
<td>32</td>
<td>1053368</td>
<td>2171</td>
<td>13</td>
<td>208621</td>
<td>9</td>
<td>274210</td>
</tr>
<tr>
<td>E. coli 536</td>
<td>4.93</td>
<td>61737</td>
<td>4</td>
<td>624</td>
<td>75.92</td>
<td>20</td>
<td>468454</td>
<td>138</td>
<td>174</td>
<td>38441</td>
<td>41</td>
<td>416</td>
<td>227379</td>
<td>4993</td>
<td>67</td>
<td>174415</td>
<td>59</td>
<td>201727</td>
</tr>
<tr>
<td>K. pneumoniae</td>
<td>5.31</td>
<td>66439</td>
<td>4</td>
<td>486</td>
<td>46.65</td>
<td>29</td>
<td>355443</td>
<td>163</td>
<td>401</td>
<td>1710</td>
<td>42</td>
<td>287</td>
<td>229804</td>
<td>4668</td>
<td>84</td>
<td>150194</td>
<td>68</td>
<td>222595</td>
</tr>
<tr>
<td>M. genitalium</td>
<td>0.58</td>
<td>7251</td>
<td>1</td>
<td>27</td>
<td>64.10</td>
<td>5</td>
<td>435990</td>
<td>9</td>
<td>1</td>
<td>4765</td>
<td>4</td>
<td>164</td>
<td>82003</td>
<td>820</td>
<td>5</td>
<td>422483</td>
<td>3</td>
<td>431935</td>
</tr>
<tr>
<td>S. aureus COL</td>
<td>2.80</td>
<td>35118</td>
<td>2</td>
<td>361</td>
<td>86.45</td>
<td>18</td>
<td>196999</td>
<td>81</td>
<td>91</td>
<td>83676</td>
<td>24</td>
<td>329</td>
<td>224286</td>
<td>3637</td>
<td>36</td>
<td>314664</td>
<td>34</td>
<td>354059</td>
</tr>
<tr>
<td>S. epidermidis RP62A</td>
<td>2.61</td>
<td>32707</td>
<td>2</td>
<td>340</td>
<td>83.27</td>
<td>17</td>
<td>364345</td>
<td>82</td>
<td>50</td>
<td>97992</td>
<td>21</td>
<td>278</td>
<td>97476</td>
<td>1765</td>
<td>51</td>
<td>122181</td>
<td>45</td>
<td>142505</td>
</tr>
<tr>
<td>S. cerevisiae</td>
<td>4.96</td>
<td>62123</td>
<td>4</td>
<td>489</td>
<td>77.94</td>
<td>29</td>
<td>287088</td>
<td>274</td>
<td>945</td>
<td>3505</td>
<td>41</td>
<td>596</td>
<td>59798</td>
<td>2794</td>
<td>110</td>
<td>103819</td>
<td>99</td>
<td>126569</td>
</tr>
<tr>
<td>S. typhimurium</td>
<td>4.85</td>
<td>60718</td>
<td>4</td>
<td>518</td>
<td>56.62</td>
<td>15</td>
<td>2334317</td>
<td>180</td>
<td>560</td>
<td>2025</td>
<td>39</td>
<td>318</td>
<td>277049</td>
<td>5242</td>
<td>81</td>
<td>151210</td>
<td>75</td>
<td>220140</td>
</tr>
<tr>
<td>Synechococcus sp.WH-8109</td>
<td>2.11</td>
<td>26486</td>
<td>2</td>
<td>126</td>
<td>55.01</td>
<td>5</td>
<td>596528</td>
<td>49</td>
<td>55</td>
<td>1350</td>
<td>17</td>
<td>120</td>
<td>483410</td>
<td>2415</td>
<td>30</td>
<td>168777</td>
<td>26</td>
<td>201082</td>
</tr>
<tr>
<td>Y. pestis CO92</td>
<td>4.65</td>
<td>58172</td>
<td>4</td>
<td>590</td>
<td>87.64</td>
<td>65</td>
<td>82702</td>
<td>243</td>
<td>101</td>
<td>65647</td>
<td>37</td>
<td>446</td>
<td>46770</td>
<td>2345</td>
<td>134</td>
<td>64859</td>
<td>125</td>
<td>67256</td>
</tr>
</tbody>
</table>

Table 7.2. Comparison of assembly approaches using 800bp reads. The column Aligned (%) shows the percentage of sequence reads anchored to a reference genome using fuzzy k-mers. The columns S-Contigs and S-N50 show the contig number and N50 value after scaffolding.
<table>
<thead>
<tr>
<th>Organism</th>
<th>Ferox</th>
<th>Amos</th>
<th>Velvet</th>
<th>Cabog</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Contigs N50</td>
<td>G (%)</td>
<td>Ml</td>
<td>NGA50</td>
</tr>
<tr>
<td>B. suis 1330</td>
<td>15</td>
<td>243423</td>
<td>99.69</td>
<td>2</td>
</tr>
<tr>
<td>C. pneumoniae CWL029</td>
<td>2</td>
<td>1211878</td>
<td>98.74</td>
<td>3</td>
</tr>
<tr>
<td>E. coli 536</td>
<td>20</td>
<td>468454</td>
<td>90.29</td>
<td>12</td>
</tr>
<tr>
<td>K. pneumoniae</td>
<td>29</td>
<td>355443</td>
<td>97.73</td>
<td>8</td>
</tr>
<tr>
<td>M. genitalium</td>
<td>5</td>
<td>435990</td>
<td>99.89</td>
<td>0</td>
</tr>
<tr>
<td>S. aureus COL</td>
<td>18</td>
<td>196999</td>
<td>94.23</td>
<td>8</td>
</tr>
<tr>
<td>S. epidermidis RP22A</td>
<td>17</td>
<td>364345</td>
<td>81.36</td>
<td>39</td>
</tr>
<tr>
<td>S. sonelensis</td>
<td>29</td>
<td>287088</td>
<td>87.76</td>
<td>28</td>
</tr>
<tr>
<td>S. typhimurium</td>
<td>15</td>
<td>2334217</td>
<td>94.27</td>
<td>3</td>
</tr>
<tr>
<td>Synchochoccus sp WH-8109</td>
<td>5</td>
<td>596528</td>
<td>91.38</td>
<td>2</td>
</tr>
<tr>
<td>Y. pestis CO92</td>
<td>65</td>
<td>82702</td>
<td>67.22</td>
<td>19</td>
</tr>
</tbody>
</table>

Table 7.3. QUAST analysis of assemblies using 800bp reads. The column G (%) refers to the percentage of the target genome assembled. Ml denotes the number of misassemblies. NGA50 represents the re-computed N50 value after fracturing misassembled contigs at breakpoints.
<table>
<thead>
<tr>
<th>Organism</th>
<th>Size (Mbp)</th>
<th>Reads</th>
<th>Expected Contigs</th>
<th>Ferox</th>
<th>Amos</th>
<th>Velvet</th>
<th>Cabog</th>
</tr>
</thead>
<tbody>
<tr>
<td>B. suis 1330</td>
<td>2.10</td>
<td>53574</td>
<td>4</td>
<td>228</td>
<td>50</td>
<td>17</td>
<td>1876</td>
</tr>
<tr>
<td>C. pneumoniae CWL029</td>
<td>1.23</td>
<td>30758</td>
<td>3</td>
<td>124</td>
<td>73</td>
<td>9</td>
<td>1698</td>
</tr>
<tr>
<td>E. coli 536</td>
<td>4.93</td>
<td>123474</td>
<td>10</td>
<td>546</td>
<td>152</td>
<td>40</td>
<td>2961</td>
</tr>
<tr>
<td>K. pneumoniae</td>
<td>5.31</td>
<td>132878</td>
<td>11</td>
<td>583</td>
<td>139</td>
<td>43</td>
<td>3436</td>
</tr>
<tr>
<td>M. genitalium</td>
<td>0.58</td>
<td>14502</td>
<td>1</td>
<td>36</td>
<td>9</td>
<td>4</td>
<td>497</td>
</tr>
<tr>
<td>S. aureus COL</td>
<td>2.80</td>
<td>70346</td>
<td>6</td>
<td>406</td>
<td>82</td>
<td>23</td>
<td>2420</td>
</tr>
<tr>
<td>S. epidermidis RP62A</td>
<td>2.61</td>
<td>65413</td>
<td>5</td>
<td>344</td>
<td>79</td>
<td>20</td>
<td>1744</td>
</tr>
<tr>
<td>S. oneidensis</td>
<td>4.96</td>
<td>124246</td>
<td>10</td>
<td>509</td>
<td>221</td>
<td>40</td>
<td>2000</td>
</tr>
<tr>
<td>S. typhimurium</td>
<td>4.85</td>
<td>123170</td>
<td>10</td>
<td>494</td>
<td>145</td>
<td>40</td>
<td>3871</td>
</tr>
<tr>
<td>Synchococcus sp. WH-8109</td>
<td>2.11</td>
<td>52972</td>
<td>4</td>
<td>148</td>
<td>45</td>
<td>15</td>
<td>1864</td>
</tr>
<tr>
<td>Y. pestis CO92</td>
<td>4.65</td>
<td>116343</td>
<td>10</td>
<td>597</td>
<td>238</td>
<td>36</td>
<td>1692</td>
</tr>
</tbody>
</table>

Table 7.4. Comparison of assembly approaches using 400bp reads. The column Aligned (%) shows the percentage of sequence reads anchored to a reference genome using fuzzy k-mers. The columns S-Contigs and S-N50 show the contig number and N50 value after scaffolding.
<table>
<thead>
<tr>
<th>Organism</th>
<th>Ferox</th>
<th></th>
<th></th>
<th></th>
<th>Amos</th>
<th></th>
<th></th>
<th></th>
<th>Velvet</th>
<th></th>
<th></th>
<th>Cabog</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Contigs</td>
<td>N50</td>
<td>G (%)</td>
<td>M_e</td>
<td>Contigs</td>
<td>N50</td>
<td>G (%)</td>
<td>M_e</td>
<td>Contigs</td>
<td>N50</td>
<td>G (%)</td>
<td>M_e</td>
<td>Contigs</td>
<td>N50</td>
</tr>
<tr>
<td>B. suis 1330</td>
<td>16</td>
<td>660417</td>
<td>92.27</td>
<td>4</td>
<td>321132</td>
<td>32</td>
<td>104725</td>
<td>89.60</td>
<td>0</td>
<td>96994</td>
<td>91</td>
<td>162835</td>
<td>99.15</td>
<td>25</td>
</tr>
<tr>
<td>C. pneumoniae CWL029</td>
<td>9</td>
<td>321444</td>
<td>99.11</td>
<td>2</td>
<td>321444</td>
<td>10</td>
<td>396714</td>
<td>99.42</td>
<td>0</td>
<td>396714</td>
<td>39</td>
<td>693042</td>
<td>99.53</td>
<td>15</td>
</tr>
<tr>
<td>E. coli 536</td>
<td>68</td>
<td>192211</td>
<td>94.77</td>
<td>23</td>
<td>134712</td>
<td>275</td>
<td>24727</td>
<td>78.52</td>
<td>0</td>
<td>18180</td>
<td>493</td>
<td>131897</td>
<td>97.72</td>
<td>72</td>
</tr>
<tr>
<td>K. pneumoniae</td>
<td>49</td>
<td>225457</td>
<td>95.03</td>
<td>9</td>
<td>201750</td>
<td>634</td>
<td>838</td>
<td>8.79</td>
<td>0</td>
<td>–</td>
<td>325</td>
<td>139662</td>
<td>98.41</td>
<td>63</td>
</tr>
<tr>
<td>M. genitalium</td>
<td>1</td>
<td>579804</td>
<td>100.00</td>
<td>0</td>
<td>579804</td>
<td>6</td>
<td>4515</td>
<td>1.02</td>
<td>0</td>
<td>–</td>
<td>168</td>
<td>79890</td>
<td>97.34</td>
<td>7</td>
</tr>
<tr>
<td>S. aureus COL</td>
<td>35</td>
<td>183073</td>
<td>94.50</td>
<td>10</td>
<td>157673</td>
<td>156</td>
<td>37239</td>
<td>92.80</td>
<td>1</td>
<td>34028</td>
<td>359</td>
<td>109626</td>
<td>97.92</td>
<td>36</td>
</tr>
<tr>
<td>S. epidermidis RP22A</td>
<td>24</td>
<td>160044</td>
<td>84.55</td>
<td>35</td>
<td>82460</td>
<td>73</td>
<td>73396</td>
<td>91.58</td>
<td>5</td>
<td>56439</td>
<td>302</td>
<td>68651</td>
<td>91.03</td>
<td>47</td>
</tr>
<tr>
<td>S. oneidensis</td>
<td>90</td>
<td>74943</td>
<td>69.31</td>
<td>27</td>
<td>39423</td>
<td>1751</td>
<td>1598</td>
<td>41.99</td>
<td>0</td>
<td>–</td>
<td>623</td>
<td>48543</td>
<td>95.15</td>
<td>171</td>
</tr>
<tr>
<td>S. typhimurium</td>
<td>36</td>
<td>276886</td>
<td>94.21</td>
<td>4</td>
<td>273114</td>
<td>949</td>
<td>967</td>
<td>15.84</td>
<td>0</td>
<td>–</td>
<td>349</td>
<td>170554</td>
<td>98.34</td>
<td>47</td>
</tr>
<tr>
<td>Synechococcus sp.WH-8109</td>
<td>9</td>
<td>185530</td>
<td>96.54</td>
<td>9</td>
<td>139216</td>
<td>94</td>
<td>673</td>
<td>2.42</td>
<td>0</td>
<td>–</td>
<td>138</td>
<td>161405</td>
<td>98.49</td>
<td>26</td>
</tr>
<tr>
<td>Y. pestis CO92</td>
<td>81</td>
<td>63177</td>
<td>65.04</td>
<td>28</td>
<td>31651</td>
<td>170</td>
<td>45527</td>
<td>95.83</td>
<td>2</td>
<td>44360</td>
<td>511</td>
<td>39311</td>
<td>94.98</td>
<td>145</td>
</tr>
</tbody>
</table>

Table 7.5. QUAST analysis of assemblies using 400bp reads. The column G (%) refers to the percentage of the target genome assembled. M_e denotes the number of misassemblies. NGA50 represents the re-computed N50 value after fracturing misassembled contigs at breakpoints.
<table>
<thead>
<tr>
<th>Organism</th>
<th>Size (Mb)</th>
<th>Reads</th>
<th>Expected Contigs</th>
<th>Contigs</th>
<th>Time (T)</th>
<th>Aligned (%)</th>
<th>N50</th>
<th>Time (T)</th>
<th>Contigs</th>
<th>N50</th>
<th>Time (T)</th>
<th>Contigs</th>
<th>N50</th>
<th>Time (T)</th>
<th>Contigs</th>
<th>N50</th>
<th>S-Contigs</th>
<th>S-N50</th>
</tr>
</thead>
<tbody>
<tr>
<td>B. suis 1330</td>
<td>2.10</td>
<td>263474</td>
<td>240</td>
<td>264</td>
<td>198</td>
<td>111</td>
<td>29651</td>
<td>20</td>
<td>220</td>
<td>22272</td>
<td>429</td>
<td>192</td>
<td>2712</td>
<td>192</td>
<td>2856</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C. pneumoniae CWL029</td>
<td>1.23</td>
<td>153779</td>
<td>140</td>
<td>136</td>
<td>78</td>
<td>1</td>
<td>122732</td>
<td>11</td>
<td>110</td>
<td>27176</td>
<td>238</td>
<td>105</td>
<td>2505</td>
<td>105</td>
<td>2662</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E. coli 536</td>
<td>4.93</td>
<td>617365</td>
<td>563</td>
<td>809</td>
<td>551</td>
<td>1</td>
<td>4789415</td>
<td>49</td>
<td>798</td>
<td>19728</td>
<td>964</td>
<td>452</td>
<td>2570</td>
<td>452</td>
<td>2750</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>K. pneumoniae</td>
<td>5.31</td>
<td>664390</td>
<td>606</td>
<td>777</td>
<td>421</td>
<td>167</td>
<td>863</td>
<td>64</td>
<td>698</td>
<td>22066</td>
<td>1041</td>
<td>457</td>
<td>2571</td>
<td>457</td>
<td>2763</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M. genitalium</td>
<td>0.58</td>
<td>72510</td>
<td>66</td>
<td>89</td>
<td>26</td>
<td>4</td>
<td>3199</td>
<td>5</td>
<td>216</td>
<td>17243</td>
<td>143</td>
<td>58</td>
<td>2678</td>
<td>58</td>
<td>2678</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S. aureus COL</td>
<td>2.80</td>
<td>351178</td>
<td>320</td>
<td>422</td>
<td>323</td>
<td>1</td>
<td>3178906</td>
<td>28</td>
<td>608</td>
<td>16459</td>
<td>548</td>
<td>265</td>
<td>2585</td>
<td>265</td>
<td>2771</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S. epidermidis RP62A</td>
<td>2.61</td>
<td>327066</td>
<td>298</td>
<td>319</td>
<td>298</td>
<td>1</td>
<td>2550354</td>
<td>26</td>
<td>484</td>
<td>18621</td>
<td>503</td>
<td>229</td>
<td>2623</td>
<td>229</td>
<td>2791</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S. oneidensis</td>
<td>4.96</td>
<td>621225</td>
<td>566</td>
<td>882</td>
<td>456</td>
<td>1505</td>
<td>1114</td>
<td>49</td>
<td>1041</td>
<td>13838</td>
<td>978</td>
<td>395</td>
<td>2508</td>
<td>395</td>
<td>2642</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S. typhimurium</td>
<td>4.85</td>
<td>607179</td>
<td>554</td>
<td>632</td>
<td>481</td>
<td>568</td>
<td>868</td>
<td>47</td>
<td>711</td>
<td>21517</td>
<td>944</td>
<td>431</td>
<td>2448</td>
<td>431</td>
<td>2614</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Synchococcus sp. WW-8109</td>
<td>2.11</td>
<td>264863</td>
<td>242</td>
<td>270</td>
<td>124</td>
<td>41</td>
<td>1099</td>
<td>20</td>
<td>272</td>
<td>24193</td>
<td>432</td>
<td>187</td>
<td>2579</td>
<td>187</td>
<td>2804</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y. pestis CO92</td>
<td>4.65</td>
<td>581716</td>
<td>530</td>
<td>764</td>
<td>496</td>
<td>1</td>
<td>4819522</td>
<td>45</td>
<td>844</td>
<td>13680</td>
<td>953</td>
<td>392</td>
<td>2640</td>
<td>392</td>
<td>2656</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7.6. Comparison of assembly approaches using 80bp reads. The column Aligned (%) shows the percentage of sequence reads anchored to a reference genome using fuzzy k-mers. The columns S-Contigs and S-N50 show the contig number and N50 value after scaffolding.
<table>
<thead>
<tr>
<th>Organism</th>
<th>Ferox</th>
<th>Amos</th>
<th>Velvet</th>
<th>Cabog</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Contigs</td>
<td>N50</td>
<td>G (%)</td>
<td>M.</td>
</tr>
<tr>
<td>B. suis 1330</td>
<td>264</td>
<td>13295</td>
<td>98.47</td>
<td>17</td>
</tr>
<tr>
<td>C. pneumoniae</td>
<td>136</td>
<td>16216</td>
<td>99.41</td>
<td>8</td>
</tr>
<tr>
<td>CWL029</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E. coli 536</td>
<td>809</td>
<td>12087</td>
<td>97.78</td>
<td>24</td>
</tr>
<tr>
<td>K. pneumoniae</td>
<td>777</td>
<td>11794</td>
<td>98.29</td>
<td>21</td>
</tr>
<tr>
<td>M. genitalium</td>
<td>89</td>
<td>10702</td>
<td>99.05</td>
<td>9</td>
</tr>
<tr>
<td>S. aureus COL</td>
<td>422</td>
<td>15253</td>
<td>97.69</td>
<td>42</td>
</tr>
<tr>
<td>S. epidermidis</td>
<td>319</td>
<td>13964</td>
<td>90.35</td>
<td>46</td>
</tr>
<tr>
<td>RP62A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S. oneidensis</td>
<td>882</td>
<td>10603</td>
<td>94.02</td>
<td>89</td>
</tr>
<tr>
<td>S. typhimurium</td>
<td>632</td>
<td>12323</td>
<td>97.58</td>
<td>23</td>
</tr>
<tr>
<td>Synechococcus</td>
<td>270</td>
<td>16728</td>
<td>98.23</td>
<td>20</td>
</tr>
<tr>
<td>sp. WH-8109</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y. pestis CO92</td>
<td>764</td>
<td>10648</td>
<td>91.34</td>
<td>66</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7.7. QUAST analysis of assemblies using 80bp reads. The column G (%) refers to the percentage of the target genome assembled. M. denotes the number of misassemblies. NGA50 represents the re-computed N50 value after fracturing misassembled contigs at breakpoints. The metrics for the AMOS assemblies of K. pneumoniae, S. oneidensis, S. typhimurium and Synechococcus were computed on the contigs produced by the assembler, as QUAST was unable to process the extensive runs of ‘N’ symbols in scaffolds.
While exhibiting running times significantly higher than those for Velvet and AMOS, the comparative approach used by the Ferox prototype includes the overhead associated with performing a fuzzy k-mer alignment of each read against a reference genome, in addition to the construction of a de Bruijn graph. Despite having to perform the dual tasks of alignment and graph construction, the running times exhibited by the prototype are neither excessive nor punitive.

In contrast, the running times for Cabog are an order of magnitude greater than those achieved with the k-mer based Velvet assembler. Notwithstanding the time spent computing and creating an overlap graph, most of the assembly time exhibited by Cabog was spent in identifying unitigs and scaffolding. Notably, with the exception of Cabog, the length of sequence reads had little effect on overall running time of the different assemblers. This suggests that the processing of sequence information has a greater impact on the running time of Cabog than the addition of read nodes to an overlap graph.

### 7.2.2 Completeness of Assembly

In terms of completeness, those assemblies produced by the prototype that contain the most misassemblies also assembled a lower percentage of the target genome. In particular, the percentage of the genomes of *S. epidermidis*, *S. oneidensis* and *Y. pestis*, assembled with both 800 and 400bp reads, is significantly less than that achieved by with the two de novo assemblers. While this may partly be attributed to the structural complexities of these genomes and the degree of synteny between the target and reference sequences, it also indicates that using anchor alignments to extend contigs through the boundaries of repeats is unsuitable for some genomes. This is evidenced by the increased percentage of these genomes assembled with short sequence reads, where a reduced number of overlaps decreased the overall connectivity of the de Bruijn graph, resulting in less opportunity for the assembler to extend contigs through repetitive nodes. By extending contigs through repeat nodes in the compressed sequence graphs of *S. epidermidis*, *S. oneidensis*, and *Y. pestis*.
oneidensis and Y. pestis, the assembler selected a number of shorter paths between anchored nodes, resulting in a loss of some sequence.

While the anchoring heuristics used in the contig assembly mechanism of the prototype stymied the completeness of some assemblies, the ability of AMOS to assemble a genome is directly affected by the degree of homology between the target and reference sequences. For highly homologous sequence assemblies, such as that of C. pneumonia, the percentage of the target genome assembled by AMOS is comparable to the best results achieved with de novo assemblers. The lack of completeness of the remaining assemblies underlines the inherent weakness of comparative assembly, where small differences in structure between the target and reference genomes truncates contig extension.

With the exception of S. epidermidis, the completeness of the Cabog assemblies, using longer sequence reads, surpasses the results achieved by any of the other assembly approaches. The Overlap-Layout-Consensus model consistently produced high quality assemblies with a large NGA50 value using 800 and 400bp reads. In particular, the assembly of Y. pestis illustrates the robustness of the assembly model in detecting repeats and correctly ordering and orientating contigs around repeats during scaffolding. While the de Bruijn graph model of de novo assembly, implemented in Velvet, does not achieve the same level of completeness using longer reads, it consistently produced high quality assemblies with a comparable NGA50 value for all read lengths.

7.2.3 Assembly Size and Quality

As noted by Salzberg et al [10] in a recent comparison of assembly algorithms, structural errors in contigs, such as inversions, relocations and translocations are common to all assemblers. At a general level, the assemblies produced by the prototype contained fewer contigs, with a higher N50 value and a greater number of misassemblies than the other approaches. All assembly models showed an increase in the number of misassemblies and a decrease in the NGA50 contig size at shorter read lengths. Shorter reads
require a higher level of coverage as the number of \( k \)-sized overlaps decreases along with read length [62]. In particular, Cabog failed to create an assembly with an NGA50 value \( \geq 3 \text{Kbp} \) in any of the tests with 80bp sequence reads. This is consistent with the results of studies by Losada et al [213] and Li et al [4], who found that Cabog performs better with mixed-length reads. The assemblies produced by AMOS were significantly affected by the degree of homology between the genome being assembled and the reference sequence. AMOS produced high quality contigs with a large NGA50 value where the DDH distance between the two genomes was < 0.1, with contig size decreasing rapidly as the DDH distance increases. In contrast, the de Bruijn graph model used by Velvet exhibited a greater capability to consistently generate substantial contigs sets with few misassemblies for all read lengths.

While the NGA50 metric provides a better measure of contig quality than the traditional N50 value, computing a quality value by fracturing a contig along breakpoints can result in a misleading number of misassemblies being reported. This is illustrated in Fig. 7.1, which shows dot plots of the B. suis 1330 assemblies against the original complete genome. Despite reporting two misassemblies for the contigs produced from 800bp reads, there was no difference in the N50 and NGA50 values computed by QUAST. This implies that the errors occurred in contigs smaller than the N50 size and that the errors represent a small proportion of the overall quality of the assembly. As illustrated in Fig. 7.1(a), the two misassemblies can be accounted for by the relocation of the two small contigs visible at the top of the figure. For the assembly of 400bp reads, a total of 4 misassemblies were responsible for a 51% difference in the N50 and NGA50 values. In this case, as shown in the dot plot in Fig. 7.1(b), the contig breakpoints were relatively small, as the large gap in the assembly is due to the absence of a contig, resulting in just 85.77% of the genome being assembled.

For all the assemblers used in the evaluation, there are less significant differences between the N50 and NGA50 values at shorter read lengths, despite a reported increase in the number of misassemblies. As contig size tends to decrease with read length, structural errors will have less of an impact on the overall quality of an assembly. This is demonstrated by the
results for the assembly of \emph{B. suis} 1330 with 80bp reads, depicted in Fig. 7.1(c). Notwithstanding the 17 misassemblies reported by QUAST, there was no difference between the N50 and NG50 values.

![Fig 7.1. MUMmer plots of the Ferox assemblies of B. suis 1330, showing contigs produced with (a) 800bp reads, (b) 400bp reads and (c) 80bp reads.](image)

A salient aspect of the assembly results is the difficulty common to all the approaches in correctly assembling particular genomes. Specifically, the genomes of \emph{S. epidermidis} and \emph{Y. pestis} are outstanding across all approaches, for the number of misassemblies reported and the low percentage of the genomes assembled. In an evaluation of \emph{de novo} assembly models, Haiminen \emph{et al} [111] concluded that assembly quality is heavily influenced by a number of different factors, including the architecture of the target genome, the assembly programme used, the average read length and the observed sequencing error rate. In the case of \emph{Y. pestis}, the complex repeat structure of the genome is known to fragment assembly [213]. Moreover, Salzberg \emph{et al} [10] identified compressed repeats as a systemic problem with short read assemblers. This observation is borne out by the results and explains why the percentage of the \emph{Y. pestis} genome assembled is lower for the de Bruijn graph model of assembly used in the prototype and Velvet than it is for the overlap graph employed by Cabog.
Notable aspects of the *S. epidermidis* assemblies are the similar NGA50 values and the level of misassemblies reported. Fig 7.2 shows a syntenic plot of the genome of *S. epidermidis* RP62A against *S. epidermidis* ATCC, along with dot plots of the contigs produced by the prototype and AMOS. Although the NGA50 value computed for the Ferox assembly was reduced to 22% of the N50 contig size, most of the breakpoints detected by QUAST were relatively small, as the larger contigs shown in Fig 7.2(b) do not contain significant structural errors.

![Fig 7.2](image)

**Fig 7.2.** MUMmer plots of the Ferox assemblies of *S. epidermidis* RP62A showing (a) a syntenic plot against the genome of *S. epidermidis* ATCC, (b) Ferox assembly with 800bp reads and (c) AMOS assembly with 800bp reads.

While the extension of contigs through repeat boundaries by the prototype generated more misassemblies, the results of the other tests illustrate the potential of the approach. In particular, the multiplicity of contigs and the N50 value produced for the assembly of *S. typhimurium* with 400 and 800bp reads demonstrates the utility of the comparative technique. For the *S. typhimurium* assembly with 800bp reads, the prototype surpassed all other approaches, generating 15 contigs with an N50 value of 2,334Kbp, with a total of 4 misassemblies. While the NGA50 value of 707Kbp represents just 30% of the N50 value, the reduced value is still significantly higher than the best results obtained using other assemblers. Moreover, the presence of a single small breakpoint in a 2334Kbp contig was responsible for QUAST re-computing the smaller NGA50 value.
The reduced NGA50 value computed for the assembly of *K. pneumoniae* resulted from a small relocation along the length of a 397Kbp contig, yielding a value of 292Kbp compared to the original 355Kbp N50 value. Similarly, two breakpoints in contigs of 821Kbp and 596Kbp were responsible for QUAST reducing the N50 value from 596Kbp to 421Kbp for the assembly with 400bp reads. In the tests with *C. pneumonia*, the prototype assembled 99.45% of the genome into just two contigs using 800bp reads, including three small breakpoints in a 1,211Kbp contig.

![MUMmer plots of the Ferox assemblies](image)

**Fig 7.3.** MUMmer plots of the Ferox assemblies of (a) *K. pneumoniae*, (b) *S. typhimurium* and (c) Synechococcus with 800bp reads.

QUAST reported just two errors along a short contig for the assembly of *C. pneumonia* with 400bp reads. There was no discrepancy between the N50 and NGA50 values using shorter 80bp sequence reads. The prototype assembled the 800bp reads of the small *M. genitalium* genome into five contigs with no errors. The full genome was assembled into a single contig by the prototype using 400bp reads, also with no errors. The assembly of *M. genitalium* with 80bp sequence reads contained nine errors but no difference in the N50 and NGA50 values.

A notable feature of the tests with the prototype is the reduction in N50 and NGA50 values for short sequence reads. This is primarily due to the read length confounding read and mate threading during contig assembly and the inability of short reads to span the gaps between anchor sequences in a
reference genome. As the NGA50 value decreases for all the genomes assembled when read length is reduced, this suggests that the fuzzy k-mer anchoring mechanism used by the prototype is biased towards longer sequence reads.

7.2.4 Comparison with the Alignment-Layout-Consensus Model

The limitations of the Alignment-Layout-Consensus model are illustrated by the results with different sequence lengths. The success of this approach is predicated on accurately aligning sequence reads against a reference genome and then using alignment and paired-read information to create a layout graph and compute a consensus sequence. The results show a direct correlation between the DDH distance [154, 155] of the target and reference genomes and the contig sizes produced by AMOS. This is not unexpected, as the DDH distance metric and the alignments employed by AMOS are both computed from the high scoring pairs produced by MUMmer alignments. While the AMOS assemblies of B. suis, C. pneumonias and S. aureus produced contigs with a high NGA50 value, the target and reference genomes are highly homologous, with DDH values of 0.0861, 0.0260 and 0.1095 respectively. The contig sizes produced by AMOS rapidly reduce as the DDH distance increases.

In contrast with the integrated assembly approach used in the Ferox prototype, AMOS failed to produce sizable contigs for the assemblies of K. pneumoniae, M. genitalium, S. typhimurium and Synechococcus, where the high DDH distance stymied the alignment phase of assembly. Despite the weak homology between these genomes and their reference sequence, the prototype anchored a significant number of sequence reads and produced assemblies with a large NGA50 value and a small number of errors. Using the same DDH metric to compute a required percentage identity match, the prototype anchored 46.65% of the K. pneumoniae reads, 64.1% of the M. genitalium reads and 56.62% of the 800bp reads used to assemble S. typhimurium. This suggests that the fuzzy k-mer alignments, using a single 25-mer fuzzy seed, are more sensitive than the 20-mer minimum exact match required to seed a
MUM alignment [16, 80]. Furthermore, the low NGA50 value computed for the AMOS assemblies of S. typhimurium and Synechococcus demonstrate the vulnerability of conventional comparative assembly to structural variations between the target and reference genomes, as the reference genomes used in both assemblies contain significant inversions and rearrangements.

The assemblies of E. coli, S. aureus, and Y. pestis illustrate the difficulty of comparative assembly with short reads. Despite producing both high contiguity and quality assemblies for these genomes with 400 and 800bp sequences, the NGA50 value of the AMOS assembly of E. coli 536 with 80bp reads is less than 1% of the N50 value. There are similar large discrepancies between the N50 and NGA50 values for the AMOS assemblies of S. aureus and Y. pestis. Given the relatively low DDH distances between these genomes and their reference sequence, structural variations, exacerbated by the small cluster size used by AMOScmp-shortReads, are the likely cause of the misassemblies reported. This explanation is supported by the high quality AMOS assembly of C. pneumonia, where 99.72% of the genome was assembled into a single contig with an NGA50 value of 1216Kbps. Read length also stymied the assembly of 80bp sequence reads using the Ferox prototype. The heuristics employed to sort and constrain membership of the priority queue used by the contig assembler, rely heavily on read threading and anchor alignments, both of which are significantly influenced by read length.

7.2.5 Comparison with de novo Assembly Models

Vezzi et al [59] argue that short reads have made the assembly problem harder, due to the complexity involved in resolving long repeats. Despite its successful adaptation to hybrid assemblies [4, 213], the inability of Cabog to generate substantial contigs sizes for 80bp reads highlights a limitation of the Best Overlap Graph (BOG) [34] used by the assembler. In a hybrid assembly, the transitive reduction of the BOG creates a bias towards longer sequences, as shorter Illumina-sized reads are more likely to be discarded from the graph. Consequently, the contig sizes produced by a hybrid assembly will be
characteristic of that expected from longer sequence reads. As the N50 metric is a measure of the fragmentation of an assembly [111], the small N50 and NGA50 values computed for the Cabog assemblies with 80bp reads demonstrate that the assembler is better suited for use with the longer sequence reads that it is designed for. This is consistent with the findings of Li et al [4], who noted the shortcomings of Overlap-Layout-Consensus approach in assembling short Illumina-length reads.

A notable feature of the Cabog assemblies with longer reads is the effective use of scaffolding to reduce the multiplicity of contigs and to boost contig size. The explicit support for read coherence in the overlap graph facilitates the identification of repeats during scaffolding, as repeat contigs will contain more reads than the level of coverage permits. Despite aggressively merging unitigs using mate-pair constraints as a greedy heuristic, Cabog produced assemblies of high contiguity and high accuracy with longer reads.

Although Velvet utilises read threading during graph construction to compute the level of coverage at each node [49, 149], the conventional de Bruijn graph approach lacks read coherence [65]. Consequently, de Bruijn graph assemblers have a limited capacity to extend contigs through repetitive nodes, without resorting to greedy heuristics. Li et al [4] contend that the ability of a de Bruijn graph assembler to resolve repeats it primarily determined by the size of $k$. Their view is supported by Simpson and Durbin [66] who argue that decomposing a sequence into $k$-mers results in a collapse of repetitive sequences whose size $> k$. This observation is borne out by the results, as the NGA50 value of the assemblies produced by Velvet is generally significantly lower than that produced by Cabog, particularly for the assemblies of Y. pestis, S. oneidensis and S. epidermidis. In contrast with the conventional de Bruijn graph approach, the incorporation of read threading and anchoring information into the assembly process added read coherence to the de Bruijn graph used by the prototype. Using read indices, mate-pair links and anchor alignments as a path selection heuristic enabled the prototype assembler to boost the NGA50 value, by extending contigs through repeat nodes in the de Bruijn graph.
7.3 Chapter Summary

The integrated approach to comparative assembly, evaluated in this chapter, enables the extension of contigs through repeat boundaries in a de Bruijn graph, by exploiting read threading and fuzzy $k$-mer alignments. Unlike current approaches to comparative assembly, which are constrained by a requirement for both strong homology and synteny, a hybrid assembly functions well in the presence of structural rearrangements and weak homology. The results of tests and benchmarks against the preeminent implementations of alternative assembly models suggest that integrating comparative techniques directly into the contig assembly process can significantly increase N50 and NGA50 contig size.

Using fuzzy $k$-mer alignments to anchor sequence reads against a reference genome, the assembly module of the prototype generated a highly contiguous assembly for many genomes, particularly using longer length sequence reads. The prototype outperformed AMOS where the target and reference genomes did not have a low DDH value and produced assemblies comparable with those generated using the two main models of *de novo* assembly. While sequence length hampered the read threading and anchoring heuristics with short reads, the results with longer sequences demonstrate the utility of the integrated approach.
Chapter 8

Summary and Conclusions

8.1 Summary of Thesis and Structure

Rapid advances in DNA sequencing technologies have enabled the routine sequencing of large genomes in a matter of days and at low cost [30, 59]. With throughput continuously expanding and sequencing cost reducing faster than Moore’s Law [66], we are approaching an inflexion point, where the main biological sequence repositories are at risk of being overwhelmed with sequence data [7, 28]. Despite a resurgence of research and the publication of novel techniques to accommodate the challenges posed by SGS sequence reads [24], the impending wide availability of third generation sequencing platforms will necessitate a further reappraisal of existing models and methods of sequence alignment and genome assembly [1, 4, 25].

This thesis has investigated the question of approximate $k$-mer matching and its application to sequence alignment and comparative assembly. The rationale for the investigation centred upon current index-based approaches to sequence alignment, where both consecutive and spaced seed methods are limited by the positional constraints of their $k$-mer models. Despite the ubiquity of $k$-mer matching techniques in sequence alignment and genome
assembly, the requirement for an exact match along some or all positions in a $k$-mer constrains current approaches, necessitating a compromise between alignment speed, sensitivity and specificity. This thesis presented a novel mechanism for $k$-mer matching that combines the speed of hashing with the sensitivity of dynamic programming. The fuzzy $k$-mer model exploits the two-step mechanism used by hash maps to resolve collisions, allowing the discrete phases of the “seed and extend” strategy to be unified into a single operation that executes in $O(1)$ average time. Tests and benchmarking have demonstrated the fuzzy $k$-mer approach to be fast, accurate and adaptable to both short and long sequences.

The related question of applying fuzzy $k$-mer alignments to genome assembly was also addressed by this thesis. By integrating fuzzy $k$-mer alignments directly into the contig assembly process, the hybrid approach described in this thesis enables the extension of contigs through the boundary of repeat nodes in a de Bruijn graph. The results from tests and benchmarks demonstrate that using read alignments as a path selection heuristic can significantly increase the contiguity of an assembly, without inducing a large number of misjoins and structural errors.

A traditional thesis structure was applied to address the research questions, where abstract models, methods and techniques are discussed and appraised in advance of the research contribution and analysis. Consequently, established models of $k$-mer matching and genome assembly were investigated first to ground the subsequent research in academically sound best practice and to inform and provide a scaffold for the research contribution itself. This is evidenced by the discussion and critique of $k$-mer matching techniques and their centrality to de novo and comparative assembly provided in chapter two. Chapter three described how an experimental computer science methodology was used to incorporate established models and methods into a prototype application to test and evaluate the research questions. Chapters four and five introduced the research contributions of this thesis, presenting the model and design of the fuzzy approach for approximate string matching and its implementation in the prototype aligner and assembler. Chapters six and seven evaluated the results
of tests and benchmarks on the prototype and explored the effectiveness of the fuzzy $k$-mer approach in addressing the challenges posed by the research questions.

### 8.2 Research Contribution

Consistent with the key objectives set down in chapter one, the primary research contribution of this thesis is the development and explication of a novel approach for approximate $k$-mer matching. By manipulating the collision detection mechanism of a hash map in the manner described by Topac [197] and by reducing the effective load factor of the map, fuzzy $k$-mers combine the sensitivity of small $k$-mer seeds with the speed and specificity of larger seeds. The fuzzy $k$-mer model is flexible and extensible, permitting different patterns of $k$-mer seeds to be used in conjunction with any approximate string-matching algorithm. The level of abstraction inherent in the design of fuzzy $k$-mers allows consecutive and spaced seeds to be directly incorporated into the fuzzy approach without violating the basic assumptions of their seed models.

The key features of the fuzzy $k$-mer model that impact on the speed, sensitivity and specificity of alignments have been identified, exercised and evaluated. Testing and benchmarking have demonstrated the fuzzy $k$-mer approach to be a fast and accurate mechanism for approximate $k$-mer matching that does not impose a significant computational burden on sequence alignment. The results of tests and benchmarks also show that a single fuzzy $k$-mer seed is both more sensitive and more specific than a consecutive $k$-mer seed and can achieve approximately the same sensitivity and specificity obtained by $k$-mer matching using multiple spaced seeds.

The second research contribution relates to the application of fuzzy $k$-mer alignments to genome assembly. By combining de novo and comparative assembly, the hybrid approach can extend contig construction through the boundary of repeat nodes in a de Bruijn graph. The results of testing and benchmarking demonstrate that integrating read alignment information
directly into the contig construction process significantly increases the contiguity of an assembly. In contrast with current comparative assembly models, the integrated approach produced assemblies of high contiguity and quality, even in the presence of structural rearrangements and weak homology between the target and reference genomes. Although short sequence reads stymied the exploitation of read threading and anchoring heuristics, the assemblies produced with longer sequences demonstrate the utility of the integrated approach. Notwithstanding the wide availability of short reads produced by SGS sequencing platforms, the integrated approach should lend itself well towards adaptation for the long sequence reads generated by emerging third generation sequencing technologies.

8.3 Conclusions and Future Research

Given the ubiquity of k-mer centric techniques in sequence alignment and genome assembly, the fuzzy k-mer model has the potential to be applied to a diverse set of problems in bioinformatics. The approach has obvious applications to the problem of aligning protein sequences, where a large alphabet of amino acid symbols will permit an approximate string-matching algorithm to be applied to a significantly greater portion of a fuzzy k-mer. Assuming the presence of an efficient mechanism for searching an amino acid substitution matrix, the fuzzy approach has the potential to enable an in-exact k-mer match of protein sequences and substantially improve specificity, without imposing a punitive running time on the process.

The fuzzy k-mer model also has applications to the problems of repeat identification and error correction in biological sequence data. Current k-mer centric approaches to these problems rely either on graph data structures or k-mer expansion techniques to identify near-matching k-mers or erroneous bases [45]. Applying fuzzy k-mers to these contexts would enable approximately matching k-mers to be identified and grouped together in a data structure that supports constant time insertion and search operations.
The fuzzy $k$-mer model may even have the potential to be directly incorporated into an assembly graph. Given that even a single erroneous base will induce bubbles and spurs of length $k$ in a de Bruijn graph, using approximate $k$-mer matching to model nodes or edges may provide a mechanism to moderate or accommodate base-calling errors. While only further study can establish the practicality of applying such an approach to a graph structure that is already exhaustive of memory, applying the technique described by Quitzau and Stoye [144] to directly construct a fuzzy $k$-mer sequence graph may offer a reasonable alternative.

At a more abstract level, the fuzzy $k$-mer model can potentially be extended to embrace the full gamut of fuzzy set capabilities described by Tanaka and Niimura [198]. In particular, as fuzzy logic deals with vagueness in information [200], using hedges and other fuzzy set methods may enable biologists to deal with alignment and synteny mapping where the homology between a target and reference genome is unclear or unknown.

Although the prototype established the viability of extending contig construction through repeat boundaries, the read threading and alignment heuristics were confounded by short length sequences. Indeed, given the capability of modern sequencing platforms to produce ultra-high levels of genome coverage, the scalability of read thread techniques in general warrants further investigation. Reformulating the assembly problem in terms of path-coherence may be a viable alternative way of addressing graph-based assembly, where a high quality contiguous assembly is already predicated on the computation of an optimal path through the assembly graph. The integrated assembly approach described in this thesis could easily be incorporated into such a structure and offer wider scope to redress the limitations of the technique.

Ultimately, as long as sequencing technologies change and evolve, alignment and assembly algorithms will have to change and evolve with them. This process of evolution should however lead to a simplification of the assembly problem and perhaps, when sequencing platforms can produce sufficiently long reads, to a complete solution. As an ever increasing number of finished
and draft genomes become available, the problems of sequence alignment, storage, retrieval, optimisation and data mining will provide bioinformaticians with equally daunting set of challenges and opportunities.
Appendix A

Publications


Appendix B

Supplementary Information
<table>
<thead>
<tr>
<th>Query Genome</th>
<th>Size (Mbp)</th>
<th>Reference Genome</th>
<th>Size (Mbp)</th>
<th>DDH Distance</th>
<th>Fuzzy Map</th>
<th>Hash Map</th>
<th>Tree Map</th>
<th>Nucmer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Memory (MB)</td>
<td>Load Time (s)</td>
<td>Align Time (s)</td>
<td>Memory (MB)</td>
</tr>
<tr>
<td>C.pneumoniae (CWL059)</td>
<td>1.23</td>
<td>C.pneumoniae (J138)</td>
<td>1.22</td>
<td>0.026</td>
<td>247</td>
<td>3</td>
<td>7</td>
<td>215</td>
</tr>
<tr>
<td>B.suis 1330</td>
<td>2.10</td>
<td>B.suis ATCC</td>
<td>1.92</td>
<td>0.0861</td>
<td>416</td>
<td>6</td>
<td>12</td>
<td>374</td>
</tr>
<tr>
<td>S.aureus COL</td>
<td>2.80</td>
<td>S.aureus JH1</td>
<td>2.90</td>
<td>0.1095</td>
<td>537</td>
<td>8</td>
<td>13</td>
<td>483</td>
</tr>
<tr>
<td>S.epidermidis RP62A</td>
<td>2.68</td>
<td>S.epidermidis ATCC 12228</td>
<td>2.53</td>
<td>0.1335</td>
<td>501</td>
<td>8</td>
<td>7</td>
<td>454</td>
</tr>
<tr>
<td>Y.pestis CO92</td>
<td>4.72</td>
<td>Y.pestis KIM</td>
<td>4.66</td>
<td>0.1763</td>
<td>887</td>
<td>14</td>
<td>38</td>
<td>790</td>
</tr>
<tr>
<td>E.coli (O157)</td>
<td>5.52</td>
<td>E.coli (K-12)</td>
<td>4.70</td>
<td>0.2447</td>
<td>898</td>
<td>14</td>
<td>29</td>
<td>804</td>
</tr>
<tr>
<td>Synechococcus sp. WH8109</td>
<td>2.11</td>
<td>Synechococcus sp. CC9902</td>
<td>2.51</td>
<td>0.5618</td>
<td>420</td>
<td>6</td>
<td>7</td>
<td>380</td>
</tr>
<tr>
<td>S.onoei deus (MR-1)</td>
<td>4.96</td>
<td>Shewanella sp (ANA-3)</td>
<td>4.97</td>
<td>0.7819</td>
<td>920</td>
<td>14</td>
<td>13</td>
<td>839</td>
</tr>
<tr>
<td>S.typhiurium</td>
<td>4.92</td>
<td>E.coli (K-12)</td>
<td>4.70</td>
<td>0.8946</td>
<td>942</td>
<td>14</td>
<td>11</td>
<td>843</td>
</tr>
<tr>
<td>K.pneumoniae</td>
<td>5.31</td>
<td>E.coli (K-12)</td>
<td>4.70</td>
<td>0.9265</td>
<td>898</td>
<td>13</td>
<td>13</td>
<td>804</td>
</tr>
<tr>
<td>Synechococcus sp. WH8109</td>
<td>2.11</td>
<td>Synechococcus sp. CC9902</td>
<td>2.23</td>
<td>0.9626</td>
<td>420</td>
<td>6</td>
<td>3</td>
<td>380</td>
</tr>
<tr>
<td>M.gonitilium</td>
<td>0.58</td>
<td>M.pneumoniae</td>
<td>0.81</td>
<td>0.9795</td>
<td>120</td>
<td>1</td>
<td>1</td>
<td>99</td>
</tr>
<tr>
<td>V.cholerae O395</td>
<td>1.12</td>
<td>E.coli (K-12)</td>
<td>4.70</td>
<td>0.9991</td>
<td>898</td>
<td>13</td>
<td>1</td>
<td>804</td>
</tr>
<tr>
<td>Random</td>
<td>5.00</td>
<td>E.coli (K-12)</td>
<td>4.70</td>
<td>0.9999</td>
<td>847</td>
<td>14</td>
<td>7</td>
<td>804</td>
</tr>
<tr>
<td>E.coli (K-12)</td>
<td>4.70</td>
<td>Random</td>
<td>5.00</td>
<td>0.9999</td>
<td>945</td>
<td>13</td>
<td>7</td>
<td>837</td>
</tr>
</tbody>
</table>

Table I. Load and alignment times for different genomes using fuzzy, hash and tree maps. A single 24-mer fuzzy seed, #/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/#/
<table>
<thead>
<tr>
<th>Query Genome</th>
<th>Size (Mbp)</th>
<th>Reference Genome</th>
<th>Size (Mbp)</th>
<th>DDH Distance</th>
<th>Fuzzy Map</th>
<th></th>
<th>Hash Map</th>
<th></th>
<th>Tree Map</th>
<th></th>
<th>Nucmer</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>C. pneumoniae</em> (CWL029)</td>
<td>1.23</td>
<td><em>C. pneumoniae</em> (J138)</td>
<td>1.22</td>
<td>0.026</td>
<td>230</td>
<td>3</td>
<td>2</td>
<td>215</td>
<td>1</td>
<td>2</td>
<td>211</td>
</tr>
<tr>
<td><em>B. suis</em> 1330</td>
<td>2.10</td>
<td><em>B. suis</em> ATCC</td>
<td>1.92</td>
<td>0.0861</td>
<td>372</td>
<td>4</td>
<td>4</td>
<td>374</td>
<td>2</td>
<td>3</td>
<td>359</td>
</tr>
<tr>
<td><em>S. aureus</em> COL</td>
<td>2.80</td>
<td><em>S. aureus</em> JH1</td>
<td>2.90</td>
<td>0.1095</td>
<td>541</td>
<td>6</td>
<td>6</td>
<td>483</td>
<td>3</td>
<td>4</td>
<td>474</td>
</tr>
<tr>
<td><em>S. epidermidis</em> RP62A</td>
<td>2.68</td>
<td><em>S. epidermidis</em> ATCC 12228</td>
<td>2.53</td>
<td>0.1335</td>
<td>469</td>
<td>6</td>
<td>5</td>
<td>454</td>
<td>2</td>
<td>4</td>
<td>442</td>
</tr>
<tr>
<td><em>Y. pestis</em> CO92</td>
<td>4.72</td>
<td><em>Y. pestis</em> KIM</td>
<td>4.66</td>
<td>0.1763</td>
<td>852</td>
<td>10</td>
<td>13</td>
<td>790</td>
<td>7</td>
<td>9</td>
<td>785</td>
</tr>
<tr>
<td><em>E. coli</em> (O157)</td>
<td>5.52</td>
<td><em>E. coli</em> (K-12)</td>
<td>4.70</td>
<td>0.2447</td>
<td>1010</td>
<td>16</td>
<td>8</td>
<td>804</td>
<td>7</td>
<td>7</td>
<td>780</td>
</tr>
<tr>
<td><em>Synechococcus</em> sp. WH8109</td>
<td>2.11</td>
<td><em>Synechococcus</em> sp. CC9605</td>
<td>2.51</td>
<td>0.5618</td>
<td>477</td>
<td>5</td>
<td>3</td>
<td>380</td>
<td>2</td>
<td>2</td>
<td>361</td>
</tr>
<tr>
<td><em>S. oneidensis</em> (MR-1)</td>
<td>4.96</td>
<td><em>Shewanella</em> sp. (ANA-3)</td>
<td>4.97</td>
<td>0.7819</td>
<td>933</td>
<td>11</td>
<td>6</td>
<td>839</td>
<td>8</td>
<td>5</td>
<td>816</td>
</tr>
<tr>
<td><em>S. pyogenes</em></td>
<td>4.92</td>
<td><em>E. coli</em> (K-12)</td>
<td>4.70</td>
<td>0.8946</td>
<td>865</td>
<td>11</td>
<td>5</td>
<td>843</td>
<td>8</td>
<td>4</td>
<td>821</td>
</tr>
<tr>
<td><em>K. pneumoniae</em></td>
<td>5.31</td>
<td><em>E. coli</em> (K-12)</td>
<td>4.70</td>
<td>0.9265</td>
<td>998</td>
<td>16</td>
<td>5</td>
<td>804</td>
<td>7</td>
<td>4</td>
<td>780</td>
</tr>
<tr>
<td><em>Synechococcus</em> sp. WH8109</td>
<td>2.11</td>
<td><em>Synechococcus</em> sp. CC9602</td>
<td>2.23</td>
<td>0.9626</td>
<td>428</td>
<td>5</td>
<td>2</td>
<td>380</td>
<td>2</td>
<td>2</td>
<td>361</td>
</tr>
<tr>
<td><em>M. genitalium</em></td>
<td>0.58</td>
<td><em>M. pneumoniae</em></td>
<td>0.81</td>
<td>0.7975</td>
<td>147</td>
<td>2</td>
<td>1</td>
<td>99</td>
<td>0</td>
<td>0</td>
<td>97</td>
</tr>
<tr>
<td><em>V. cholerae</em> O395</td>
<td>1.12</td>
<td><em>E. coli</em> (K-12)</td>
<td>4.70</td>
<td>0.9991</td>
<td>865</td>
<td>11</td>
<td>1</td>
<td>804</td>
<td>7</td>
<td>1</td>
<td>780</td>
</tr>
<tr>
<td>Random</td>
<td>5.00</td>
<td>Random</td>
<td>5.00</td>
<td>0.9999</td>
<td>873</td>
<td>11</td>
<td>4</td>
<td>804</td>
<td>7</td>
<td>4</td>
<td>780</td>
</tr>
<tr>
<td><em>E. coli</em> (K-12)</td>
<td>4.70</td>
<td>Random</td>
<td>5.00</td>
<td>0.9999</td>
<td>948</td>
<td>12</td>
<td>10</td>
<td>837</td>
<td>8</td>
<td>11</td>
<td>852</td>
</tr>
</tbody>
</table>

Table II. Load and alignment times for different genomes using fuzzy, hash and tree maps. A single 24-mer spaced seed of -####-####-####-####-####-####-#### was used for all fuzzy alignments.
<table>
<thead>
<tr>
<th>Query Genome</th>
<th>Size (Mbp)</th>
<th>Reference Genome</th>
<th>Size (Mbp)</th>
<th>DDH Distance</th>
<th>Fuzzy Map</th>
<th>Hash Map</th>
<th>Tree Map</th>
<th>Nucmer</th>
</tr>
</thead>
<tbody>
<tr>
<td>C. pneumoniae (CW1029)</td>
<td>1.23</td>
<td>C. pneumoniae (J138)</td>
<td>1.22</td>
<td>0.026</td>
<td>228 3 4</td>
<td>215 1 2</td>
<td>211 4 8</td>
<td>1</td>
</tr>
<tr>
<td>B. suis 1339</td>
<td>2.10</td>
<td>B. suis ATCC</td>
<td>1.92</td>
<td>0.0861</td>
<td>361 5 7</td>
<td>374 2 3</td>
<td>359 8 14</td>
<td>4</td>
</tr>
<tr>
<td>S. aureus COL</td>
<td>2.80</td>
<td>S. aureus JH</td>
<td>2.90</td>
<td>0.1095</td>
<td>505 7 11</td>
<td>483 3 4</td>
<td>474 11 22</td>
<td>8</td>
</tr>
<tr>
<td>S. epidermidis RP62A</td>
<td>2.68</td>
<td>S. epidermidis ATCC 12228</td>
<td>2.53</td>
<td>0.1335</td>
<td>444 6 10</td>
<td>454 2 4</td>
<td>442 11 19</td>
<td>8</td>
</tr>
<tr>
<td>Y. pestis CO92</td>
<td>4.72</td>
<td>Y. pestis KIM</td>
<td>4.66</td>
<td>0.1763</td>
<td>827 12 21</td>
<td>790 7 9</td>
<td>765 21 42</td>
<td>20</td>
</tr>
<tr>
<td>E. coli (O157)</td>
<td>5.52</td>
<td>E. coli (K-12)</td>
<td>4.70</td>
<td>0.2447</td>
<td>962 14 16</td>
<td>804 7 7</td>
<td>780 21 47</td>
<td>16</td>
</tr>
<tr>
<td>Synechococcus sp. WH8109</td>
<td>2.11</td>
<td>Synechococcus sp. CC9605</td>
<td>2.51</td>
<td>0.3618</td>
<td>460 6 5</td>
<td>380 2 2</td>
<td>361 8 17</td>
<td>12</td>
</tr>
<tr>
<td>S. oneidensis (MR-1)</td>
<td>4.96</td>
<td>Shewanella sp. (ANA-3)</td>
<td>4.97</td>
<td>0.7819</td>
<td>909 16 10</td>
<td>839 8 5</td>
<td>816 22 48</td>
<td>39</td>
</tr>
<tr>
<td>S. pyogenes</td>
<td>4.92</td>
<td>E. coli (K-12)</td>
<td>4.70</td>
<td>0.8946</td>
<td>847 11 8</td>
<td>843 8 4</td>
<td>821 21 38</td>
<td>28</td>
</tr>
<tr>
<td>K. pneumoniae</td>
<td>5.31</td>
<td>E. coli (K-12)</td>
<td>4.70</td>
<td>0.9265</td>
<td>938 17 8</td>
<td>804 7 4</td>
<td>780 21 43</td>
<td>21</td>
</tr>
<tr>
<td>Synechococcus sp. WH8109</td>
<td>2.11</td>
<td>Synechococcus sp. CC9902</td>
<td>2.23</td>
<td>0.9626</td>
<td>421 5 3</td>
<td>380 2 2</td>
<td>361 8 13</td>
<td>5</td>
</tr>
<tr>
<td>M. genitalium</td>
<td>0.58</td>
<td>M. pneumoniae</td>
<td>0.81</td>
<td>0.9795</td>
<td>145 2 1</td>
<td>99 0 0</td>
<td>97 1 3</td>
<td>1</td>
</tr>
<tr>
<td>V. cholerae 0395</td>
<td>1.12</td>
<td>E. coli (K-12)</td>
<td>4.70</td>
<td>0.9991</td>
<td>847 11 2</td>
<td>804 7 1</td>
<td>780 21 9</td>
<td>4</td>
</tr>
<tr>
<td>Random</td>
<td>5.00</td>
<td>E. coli (K-12)</td>
<td>4.70</td>
<td>0.9599</td>
<td>870 11 11</td>
<td>804 7 4</td>
<td>780 21 40</td>
<td>10</td>
</tr>
<tr>
<td>E. coli (K-12)</td>
<td>4.70</td>
<td>Random</td>
<td>5.00</td>
<td>0.9999</td>
<td>939 12 11</td>
<td>837 8 11</td>
<td>852 23 41</td>
<td>10</td>
</tr>
</tbody>
</table>

Table III. Load and alignment times for different genomes using fuzzy, hash and tree maps. A single 24-mer fuzzy spaced seed, ****-***-***-***-***-***-***-***-***, configured with the Damerau-Levenshtein algorithm and a β cut-off threshold of 0.25 was used for all fuzzy alignments.
<table>
<thead>
<tr>
<th>Hash Size</th>
<th>Fuzzy Map</th>
<th></th>
<th>Hash Map</th>
<th></th>
<th>Tree Map</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Load Time (s)</td>
<td>β Cut-off threshold</td>
<td>Number of Keys</td>
<td>Seed Pattern</td>
<td>Load Time (s)</td>
<td>Number of Keys</td>
</tr>
<tr>
<td>20</td>
<td>5</td>
<td>0.5</td>
<td>1911477</td>
<td>...............................****</td>
<td>2</td>
<td>1911433</td>
</tr>
<tr>
<td>19</td>
<td>5</td>
<td>0.5</td>
<td>1911200</td>
<td>...............................****</td>
<td>2</td>
<td>1910797</td>
</tr>
<tr>
<td>18</td>
<td>5</td>
<td>0.5</td>
<td>1910372</td>
<td>...............................****</td>
<td>2</td>
<td>1909953</td>
</tr>
<tr>
<td>17</td>
<td>5</td>
<td>0.5</td>
<td>1909906</td>
<td>...............................****</td>
<td>2</td>
<td>1908451</td>
</tr>
<tr>
<td>16</td>
<td>5</td>
<td>0.5</td>
<td>1908052</td>
<td>...............................****</td>
<td>2</td>
<td>1905056</td>
</tr>
<tr>
<td>15</td>
<td>5</td>
<td>0.5</td>
<td>1906628</td>
<td>...............................****</td>
<td>2</td>
<td>1895820</td>
</tr>
<tr>
<td>14</td>
<td>5</td>
<td>0.5</td>
<td>1897097</td>
<td>...............................****</td>
<td>2</td>
<td>1868607</td>
</tr>
<tr>
<td>13</td>
<td>5</td>
<td>0.5</td>
<td>1887999</td>
<td>...............................****</td>
<td>2</td>
<td>1788643</td>
</tr>
<tr>
<td>12</td>
<td>7</td>
<td>0.5</td>
<td>1814809</td>
<td>...............................****</td>
<td>2</td>
<td>1578407</td>
</tr>
<tr>
<td>11</td>
<td>13</td>
<td>0.5</td>
<td>1745850</td>
<td>...............................****</td>
<td>2</td>
<td>1157492</td>
</tr>
<tr>
<td>10</td>
<td>23</td>
<td>0.5</td>
<td>1361851</td>
<td>...............................****</td>
<td>2</td>
<td>620488</td>
</tr>
<tr>
<td>9</td>
<td>50</td>
<td>0.5</td>
<td>1129427</td>
<td>...............................****</td>
<td>1</td>
<td>229749</td>
</tr>
<tr>
<td>8</td>
<td>62</td>
<td>0.5</td>
<td>522671</td>
<td>...............................****</td>
<td>1</td>
<td>64658</td>
</tr>
<tr>
<td>7</td>
<td>125</td>
<td>0.5</td>
<td>342978</td>
<td>...............................****</td>
<td>1</td>
<td>16379</td>
</tr>
<tr>
<td>6</td>
<td>107</td>
<td>0.5</td>
<td>94998</td>
<td>...............................****</td>
<td>1</td>
<td>4096</td>
</tr>
</tbody>
</table>

Table IV. The effect of hash size on key multiplicity in different maps, for the alignment B. suis 1330 against a 2.1Mbp randomly generated genome.
<table>
<thead>
<tr>
<th>Fuzzy Seed</th>
<th>Hamming</th>
<th>Levenshtein</th>
<th>Damerau Levenshtein</th>
<th>Smith Waterman</th>
<th>Exact Seed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Load Time (s)</td>
<td>Align Time (s)</td>
<td>Load Time (s)</td>
<td>Align Time (s)</td>
<td>Load Time (s)</td>
</tr>
<tr>
<td>β Cut-off threshold</td>
<td>k</td>
<td>Hash</td>
<td>Seed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.5</td>
<td>24</td>
<td>20</td>
<td>#######</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>0.5</td>
<td>24</td>
<td>19</td>
<td>#######</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>0.5</td>
<td>24</td>
<td>18</td>
<td>#######</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>0.5</td>
<td>24</td>
<td>17</td>
<td>#######</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>0.5</td>
<td>24</td>
<td>16</td>
<td>#######</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>0.5</td>
<td>24</td>
<td>15</td>
<td>#######</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>0.5</td>
<td>24</td>
<td>14</td>
<td>#######</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>0.5</td>
<td>24</td>
<td>13</td>
<td>#######</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>0.5</td>
<td>24</td>
<td>12</td>
<td>#######</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>0.5</td>
<td>24</td>
<td>11</td>
<td>#######</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>0.5</td>
<td>24</td>
<td>10</td>
<td>#######</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>0.5</td>
<td>24</td>
<td>9</td>
<td>10</td>
<td>17</td>
<td>37</td>
</tr>
<tr>
<td>0.5</td>
<td>24</td>
<td>8</td>
<td>10</td>
<td>17</td>
<td>37</td>
</tr>
<tr>
<td>0.5</td>
<td>24</td>
<td>7</td>
<td>10</td>
<td>17</td>
<td>37</td>
</tr>
<tr>
<td>0.5</td>
<td>24</td>
<td>6</td>
<td>10</td>
<td>17</td>
<td>37</td>
</tr>
</tbody>
</table>

**Table V.** The effect of hash size on running time using different string-similarity algorithms, for the alignment of *B. suis* 1330 against a 2.1Mbp randomly generated genome. The load and alignment times for an exact seed were measured after fixing the `equals()` method to return false.
<table>
<thead>
<tr>
<th>Fuzzy Seed</th>
<th>g Cut-off threshold</th>
<th>Needleman-Wunsch</th>
<th>Smith-Waterman</th>
<th>Exact Seed</th>
<th>Load Time (s)</th>
<th>Align Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hamming</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Levenshtein</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Distance</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table VI. The effect of hash size on running time, using different string-similarity algorithms. For the alignment of B. suis ATCC against the highly homologous genome of B. suis 1330, against the highly homologous genome of B. suis ATCC. The load and alignment times for each seed were measured after fixing the equal(0) method to return false.
<table>
<thead>
<tr>
<th>Genome</th>
<th>Size (Mbps)</th>
<th>Total Sequences</th>
<th>Number Aligned</th>
<th>Correct (TP)</th>
<th>Missed (FN)</th>
<th>False Positives (FP)</th>
<th>Time (s)</th>
<th>Sensitivity (%)</th>
<th>Specificity (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S. epidermidis RF62A</td>
<td>2.61</td>
<td>32706</td>
<td>32664</td>
<td>32327</td>
<td>379</td>
<td>337</td>
<td>151</td>
<td>98.84</td>
<td>98.97</td>
</tr>
<tr>
<td>E. coli 536</td>
<td>4.93</td>
<td>61736</td>
<td>61576</td>
<td>61298</td>
<td>438</td>
<td>278</td>
<td>254</td>
<td>99.29</td>
<td>99.55</td>
</tr>
<tr>
<td>S. aureus COL</td>
<td>2.81</td>
<td>35118</td>
<td>35079</td>
<td>34906</td>
<td>212</td>
<td>173</td>
<td>162</td>
<td>99.40</td>
<td>99.51</td>
</tr>
<tr>
<td>Synechococcus sp. WH8109</td>
<td>2.11</td>
<td>26864</td>
<td>26713</td>
<td>26637</td>
<td>227</td>
<td>76</td>
<td>86</td>
<td>99.16</td>
<td>99.72</td>
</tr>
<tr>
<td>Y. pestis CO92</td>
<td>4.65</td>
<td>58172</td>
<td>57854</td>
<td>56756</td>
<td>1416</td>
<td>1098</td>
<td>212</td>
<td>97.57</td>
<td>98.10</td>
</tr>
<tr>
<td>K. pneumoniae MGH-78578</td>
<td>5.31</td>
<td>66440</td>
<td>66363</td>
<td>66119</td>
<td>321</td>
<td>244</td>
<td>317</td>
<td>99.52</td>
<td>99.63</td>
</tr>
<tr>
<td>M. flagellatus KT</td>
<td>2.97</td>
<td>37144</td>
<td>37002</td>
<td>33547</td>
<td>3597</td>
<td>3455</td>
<td>120</td>
<td>90.32</td>
<td>90.66</td>
</tr>
<tr>
<td>Neisseria meningitidis MC58</td>
<td>2.27</td>
<td>28404</td>
<td>28272</td>
<td>25977</td>
<td>2427</td>
<td>2295</td>
<td>138</td>
<td>91.46</td>
<td>91.88</td>
</tr>
</tbody>
</table>

**Table VII.** Sensitivity, selectivity and alignment running times of Ferox using the single 24-mer fuzzy seed

The `FuzzyLevenshtein` class was used to compute $\mu_A(S)$. 
<table>
<thead>
<tr>
<th>Genome</th>
<th>Size (Mbps)</th>
<th>Total Sequences</th>
<th>Number Aligned</th>
<th>Correct (TP)</th>
<th>Missed (FN)</th>
<th>False Positives (FP)</th>
<th>Time (s)</th>
<th>Sensitivity (%)</th>
<th>Specificity (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S.epidermidis RP62A</td>
<td>2.61</td>
<td>32706</td>
<td>29467</td>
<td>28004</td>
<td>4702</td>
<td>1463</td>
<td>431</td>
<td>85.62</td>
<td>95.04</td>
</tr>
<tr>
<td>E.coli 536</td>
<td>4.93</td>
<td>61736</td>
<td>53754</td>
<td>52202</td>
<td>9534</td>
<td>1552</td>
<td>546</td>
<td>84.56</td>
<td>97.11</td>
</tr>
<tr>
<td>S.aureus COL</td>
<td>2.81</td>
<td>35118</td>
<td>31426</td>
<td>30352</td>
<td>4766</td>
<td>1074</td>
<td>412</td>
<td>86.43</td>
<td>96.58</td>
</tr>
<tr>
<td>Synechococcus sp. WH8109</td>
<td>2.11</td>
<td>26864</td>
<td>23034</td>
<td>22550</td>
<td>4314</td>
<td>484</td>
<td>211</td>
<td>83.94</td>
<td>97.90</td>
</tr>
<tr>
<td>Y.pstis CO92</td>
<td>4.65</td>
<td>58172</td>
<td>61822</td>
<td>49323</td>
<td>8849</td>
<td>12499</td>
<td>1123</td>
<td>84.79</td>
<td>79.78</td>
</tr>
<tr>
<td>K.pneumoniae MGH-78578</td>
<td>5.31</td>
<td>66440</td>
<td>56899</td>
<td>55876</td>
<td>10564</td>
<td>1023</td>
<td>1147</td>
<td>84.10</td>
<td>98.20</td>
</tr>
<tr>
<td>M.flagellatus KT</td>
<td>2.97</td>
<td>37144</td>
<td>34326</td>
<td>26372</td>
<td>10772</td>
<td>7954</td>
<td>344</td>
<td>71.00</td>
<td>76.83</td>
</tr>
<tr>
<td>Neisseria meningitidis MC58</td>
<td>2.27</td>
<td>28404</td>
<td>26861</td>
<td>20768</td>
<td>7636</td>
<td>6093</td>
<td>1827</td>
<td>73.12</td>
<td>77.32</td>
</tr>
</tbody>
</table>

Table VIII. Sensitivity, selectivity and alignment running times of BLAT using the parameters -tileSize=12 -oneOff=1 -minIdentity=90.
<table>
<thead>
<tr>
<th>Genome</th>
<th>Size (Mbp)</th>
<th>Total Sequences</th>
<th>Number Aligned</th>
<th>Correct (TP)</th>
<th>Missed (FN)</th>
<th>False Positives (FP)</th>
<th>Time (s)</th>
<th>Sensitivity (%)</th>
<th>Specificity (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>S. epidermidis RP62A</td>
<td>2.61</td>
<td>32706</td>
<td>32660</td>
<td>32660</td>
<td>46</td>
<td>0</td>
<td>442</td>
<td>99.86</td>
<td>100.00</td>
</tr>
<tr>
<td>E. coli 536</td>
<td>4.93</td>
<td>61736</td>
<td>61623</td>
<td>61623</td>
<td>113</td>
<td>0</td>
<td>697</td>
<td>99.82</td>
<td>100.00</td>
</tr>
<tr>
<td>S. aureus COL</td>
<td>2.81</td>
<td>35118</td>
<td>35077</td>
<td>35077</td>
<td>41</td>
<td>0</td>
<td>412</td>
<td>99.88</td>
<td>100.00</td>
</tr>
<tr>
<td>Synechococcus sp. WH8109</td>
<td>2.11</td>
<td>26864</td>
<td>26775</td>
<td>26775</td>
<td>89</td>
<td>0</td>
<td>310</td>
<td>99.67</td>
<td>100.00</td>
</tr>
<tr>
<td>Y. pestis CO92</td>
<td>4.65</td>
<td>58172</td>
<td>58092</td>
<td>58092</td>
<td>80</td>
<td>0</td>
<td>1634</td>
<td>99.86</td>
<td>100.00</td>
</tr>
<tr>
<td>K. pneumoniae MGH-78578</td>
<td>5.31</td>
<td>66440</td>
<td>66318</td>
<td>66318</td>
<td>122</td>
<td>0</td>
<td>810</td>
<td>99.82</td>
<td>100.00</td>
</tr>
<tr>
<td>M. flagellatus KT</td>
<td>2.97</td>
<td>37144</td>
<td>37078</td>
<td>37078</td>
<td>66</td>
<td>0</td>
<td>431</td>
<td>99.82</td>
<td>100.00</td>
</tr>
<tr>
<td>Neisseria meningitidis MC58</td>
<td>2.27</td>
<td>28404</td>
<td>28368</td>
<td>28368</td>
<td>36</td>
<td>0</td>
<td>641</td>
<td>99.87</td>
<td>100.00</td>
</tr>
</tbody>
</table>

**Table IX.** Sensitivity, selectivity and alignment running times of Mosaik using the parameters -hs 12 -mm 80 -mmp 0.10 -mhp 100 -act 24 -bw 51.
Bibliography


[40] J. Li, J. Jiang, and F. C. Leung, "6-10X pyrosequencing is a practical approach for whole prokaryote genome studies," *Gene*, 2011.


