APPLICATIONS OF THE COMPACTED DE BRUIJN GRAPH IN

COMPARATIVE GENOMICS

A Dissertation in
Computer Science and Engineering
by
Ilia Minkin

© 2020 Ilia Minkin

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

May 2020
The dissertation of Ilia Minkin was reviewed and approved by the following:

Paul Medvedev  
Associate Professor of Computer Science and Engineering  
Associate Professor of Biochemistry and Molecular Biology  
Dissertation Advisor  
Chair of Committee

Kamesh Madduri  
Associate Professor of Computer Science and Engineering

Mingfu Shao  
Charles K. Etner Early Career Assistant Professor of Computer Science and Engineering

Kateryna Makova  
Francis R. and Helen M. Pentz Professor of Biology

Chitaranjan Das  
Distinguished Professor of Computer Science and Engineering  
Head of the Department of Computer Science and Engineering
Abstract

Comparative genomics is one of the most powerful tools in molecular biology. The comparative paradigm addresses the challenges of genome annotation and studying evolution by comparing genomes with each other. However, effectively carrying out comparative analyses relies on our technical ability to effectively find homologous nucleotides in the genomic sequences.

On the genome-wide scale, sequence homology is usually represented in the form of whole-genome alignment, or groups of homologous genomic intervals like syntenic blocks or pairwise homology maps. Although these objects can be effectively computed for shorter sequences like bacterial genomes, there are very few tools that can handle datasets consisting of multiple mammalian-sized genomes. In addition, they require large computer clusters to run.

Compacted de Bruijn graph is a promising data structure for storing and indexing multiple assembled genomes in a computer’s memory. The graph “collapses” collections of common substrings into single edges in the graph, which results in very dense representation of the sequences. As the result, the compacted de Bruijn graph is especially useful for inputs containing long, closely-related genomes. We believe that such data structure presents a great opportunity for efficient comparative analyses of large datasets.

In this dissertation, we address the challenge of computing sequence homology for datasets consisting of multiple mammalian-sized genomes by applying the compacted de Bruijn graph. First, we introduce TwoPaCo, a scalable method for constructing the compacted de Bruijn graph from multiple completed genomic sequences, which is able to handle tens of long genomes. Then we describe SibeliaZ, a method for whole-genome alignment of closely-related genomes relying on TwoPaCo. SibeliaZ is highly efficient and is able to align 16 mice genomes on a single server in less than a day while having comparable recall of orthologous genes and even having higher recall of paralogous genes of high similarity than other state-of-the-art aligners. Finally, we introduce BubbZ, an efficient method for computing pairwise whole-genome homology maps for collections of long genomes. BubbZ is able to find whole-genome homology mapping extremely quickly, being up to 10 faster than competing programs without dropping in accuracy.

We believe that this software suite will democratize the comparative analysis of large datasets consisting of complete genomes. By being highly efficient and easy to use, our tools provide an opportunity to perform high-scale comparative analyses for using reasonable computational resources. This way, we hope they will facilitate many future biological discoveries.
## Table of Contents

List of Figures ................................................................. vi
List of Tables ................................................................. viii
Acknowledgments .............................................................. ix

### Chapter 1
**Introduction** 1
  1.1 Comparative genomics ................................................. 1
  1.2 The problem of sequence alignment .................................. 2
  1.3 Alternative representations of sequence homology .................. 2
  1.4 Challenges in computing sequence homology for long sequences ... 3
  1.5 Contributions and organization of the dissertation ............... 4

### Chapter 2
TwoPaCo: An efficient algorithm to build the compacted de Bruijn graph from many complete genomes 5
  2.1 Introduction .............................................................. 5
  2.2 Methods ................................................................. 7
    2.2.1 Preliminaries ...................................................... 7
    2.2.2 Reduction to the problem of finding junction positions ...... 9
    2.2.3 Single round algorithm .......................................... 10
    2.2.4 Multiple rounds: dealing with memory restrictions .......... 12
    2.2.5 Parallelization scheme .......................................... 14
    2.2.6 Theoretical analysis and comparison ........................... 14
  2.3 Results ................................................................. 15
  2.4 Discussion .............................................................. 19

### Chapter 3
Scalable multiple whole-genome alignment and locally collinear block construction with SibeliaZ 22
  3.1 Introduction ............................................................. 22
  3.2 Methods ................................................................. 24
    3.2.1 Preliminaries ...................................................... 24
    3.2.2 Problem formulation .......................................... 24
    3.2.3 The collinear blocks reconstruction algorithm .......... 27
    3.2.4 Other considerations ........................................... 29
Chapter 4
Scalable pairwise whole-genome homology mapping of long genomes with BubbZ
4.1 Introduction ............................................. 39
4.2 Methods ................................................. 40
4.2.1 Preliminaries ......................................... 41
4.2.2 Chains ................................................ 41
4.2.3 Problem formulation and recurrence solution .......... 42
4.2.4 Algorithm ............................................. 43
4.2.5 Data structures and other considerations ................. 44
4.2.6 Computational complexity ................................ 46
4.3 Results .................................................. 46
4.3.1 Datasets .............................................. 46
4.3.2 Evaluated tools ....................................... 46
4.3.3 Evaluation metrics .................................... 47
4.3.4 Results on the mice data ............................... 48
4.3.5 Results on the simulated data ............................ 48
4.4 Discussion ............................................... 50

Chapter 5
Conclusion and future work ........................................ 51

Appendix A
Supplementary information for Chapter 2 .......................... 53
A.1 Parameters used for benchmarking .............................. 53

Appendix B
Supplementary information Chapter 3 .............................. 55
B.1 Parameter details and command lines ........................... 55
B.2 Simulation details ......................................... 57

Bibliography ................................................. 67
List of Figures

2.1 The de Bruijn graph and its compacted version. ............................... 8
2.2 Parallel speedup of the different parts of TwoPaCo. .......................... 17
2.3 Effects of the input length and structure on the memory and running time. . 20

3.1 The de Bruijn graph and an example of a collinear block. .................... 25
3.2 An example of computing the score of a walk relative to a carrying path. . . 26
3.3 An illustration of running Algorithm 5. .............................................. 28
3.4 An example of running Algorithm 4. ............................................... 30
3.5 Running times of the different pipelines on the mice datasets (on a log scale). . 34
3.6 Recall of orthologous and paralogous nucleotide pairs by SibeliaZ and Cactus. . 36

4.1 An example of the de Bruijn graph and running Algorithm 6 on it. .......... 42
4.2 Recall of the position pairs belonging to pairs of orthologous and paralogous
protein-coding genes by BubbZ, Minimap2 and MashMap2. ....................... 49
4.3 Accuracy of BubbZ, Minimap2 and MashMap2 on the simulated bacterial datasets. 49

B.1 Properties of the pairwise alignments constructed from pairs of homologous
protein-coding genes in the various mice datasets. ................................. 58
B.2 Histogram of the column scores $f(C)$ of SibeliaZ’s and Cactus’ alignment. . 59
B.3 The recall as a function of inferred family size, using the two-mice dataset. . 60
B.4 Effects of the parameters $k$ and $b$ on recall on the simulated dataset. ........ 61
B.5 Effects of the parameters $k$ and $b$ on precision on the simulated dataset. . . . . 62

B.6 The accuracy of SibeliaZ as a function of genomic divergence. . . . . . . . . . . 63
# List of Tables

2.1 Running times and memory consumption of different algorithms for constructing the de Bruijn graph from multiple complete genomes. ........................................ 15

2.2 Benchmarking comparisons of different methods for constructing the compacted de Bruijn graph. ................................................................. 17

2.3 The minimal number of rounds it takes for TwoPaCo to compress the graph without exceeding a given memory threshold. ................................. 19

2.4 Number of marks in the array $C$ initially and after each pass of Algorithm 2. . 19

3.1 Properties of the multiple whole-genome alignments computed by SibeliaZ and Cactus from the mice datasets. ................................................. 34

4.1 Running times and memory usage of BubbZ, Minimap2 and MashMap2 on mice datasets. ................................................................. 48

4.2 Running times and memory usage of BubbZ, Minimap2 and MashMap2 on the simulated data. ................................................................. 50

B.1 Properties of the assembled mice genomes available at GenBank. ............... 64

B.2 Running times and memory usage of SibeliaZ and Cactus on the mice datasets. 65

B.3 GitHub revisions of the software used for benchmarking. ......................... 66
Acknowledgments

First, I would like to thank my advisor, Dr. Paul Medvedev for his guidance and invaluable support during the course of my Ph.D. I thank Dr. Bob Harris and Dr. Anton Nekrutenko for many fruitful scientific discussions. I thank Dr. Son Pham for introducing me to the problem of finding synteny blocks back in 2012, which laid down the course of my research all the way to my Ph.D. I would also like to thank my dissertation committee for their service: Dr. Kamesh Madduri, Dr. Mingfu Shao, and Dr. Kateryna Makova.

This material is based upon work supported by the National Science Foundation under Grant DBI-1356529, CCF1439057, IIS-1453527, and IIS-1421908. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.
Dedication

To my parents, Galina Petrovna Minkina and Valeriy Vladimirovich Minkin.
Chapter 1  
Introduction

1.1 Comparative genomics

Genomic sequences are the key source of information about heritable phenotypic traits in organisms. However, the sequences themselves do not readily reveal how particular traits are encoded. To find the correspondence between genotype and phenotype, the sequences have to be annotated with information about their functions, and producing such annotation is a very challenging problem. In addition, genomic sequences provide important evidence about the evolutionary history of organisms. However, as with encoding traits, it is difficult to extract evolutionary history right out of the sequences: we do not know which nucleotides were inherited from an ancestor and which ones are novel.

One way to address these challenges is comparative genomics [1]. The comparative approach for annotation relies on an observation that similar phenotypic traits are likely to be encoded by similar sequences. This way, we can project already existing knowledge about functions of particular DNA sequences of one genome into another. Comparing genomes also provides invaluable information about their evolution: highly similar sequences are likely to be inherited from a common ancestor. This observation opens a way to get information about sequence conservation which can be used as a precursor for finding evolutionary distances or even a history of evolutionary events.

However, carrying out comparative analyses relies on our technical ability to effectively find nucleotides inherited from a common ancestor; such nucleotides are called homologous. Unfortunately, it is impossible to definitively reconstruct all homologous relationships in sequences of living organisms since we cannot directly observe evolution. At the same time, we can construct a reasonable approximation by relying on the fact that similar DNA substrings are likely to be homologous.

In general, computing sequence homology is a difficult problem. However, there are several ways of representing it, and computing different forms require different amounts of resources. One of the most widely used and studied is the sequence alignment.
1.2 The problem of sequence alignment

In sequence alignment, homologous nucleotides are grouped in blocks, where each block is a table with homologous nucleotides written in the same column. There are several models of the alignment that vary in their evolutionary assumptions. Historically, global alignment was the first such model [2]. The global alignment model assumes that the relative order of homologous nucleotides is not changed. In other words, global alignment only allows point mutations, insertions, and deletions.

Although global alignment is immensely useful in comparing relatively short stretches of DNA up to the gene length, it is not applicable in case of comparing the whole genomes. In reality, genomes undergo rearrangements changing the order of large blocks of nucleotides which makes global alignment inapplicable. This challenge was first addressed by developing ideas behind the global alignment into local alignment model [3]. In this setting, the problem is to find pairs of nucleotide blocks that can be globally aligned with high confidence.

Pairwise local alignment does provide homology for genomes that underwent genome rearrangements, but such a model has several limitations. Mainly, pairwise alignment is usually limited to two genomes and is not directly applicable when a comparison of multiple genomes is needed. Several workarounds for this issue are known; one of them involves designating one genome as a reference and reporting all pairwise alignments relative to it [4, 5]. Such alignment can be useful in certain contexts, but they fail to capture many homologous relationships between sequences non-existing in the reference. Second, pairwise alignment blocks usually can overlap which makes the downstream analysis more challenging.

These issues are addressed in the most general form of alignment: multiple whole-genome alignment. In this representation, blocks with nucleotides can have multiple rows with fully resolved overlap between them. Although such a view provides the most complete and convenient information about homologies between a set of genomes, it is hard to compute. So far, there are only a few tools that allow computing such alignments for multiple mammalian-sized genomes and they require large computer clusters to run on real-world datasets [4, 6, 7].

1.3 Alternative representations of sequence homology

Alignment is not the only way to represent homology between genomes. Another class of models seeks to find a set of homologous intervals or blocks. Unlike alignment, such a view does not describe homology at the resolution of single nucleotides, but still it can be useful in many contexts.

One such context is synteny reconstruction. In this setting, genomes are decomposed into long blocks such that the gene order within each block is preserved. This is similar to locally collinear blocks, but collinear blocks are usually shorter and represent single genes or
exons (or non-coding DNA). Collinear blocks can be viewed as high resolution synteny blocks and, in general, the distinction between the two concepts can be blurry. For a discussion on representation of synteny blocks at multiple scales, see [8]. Synteny blocks are often reconstructed from anchors such as genes [9–12] and, less commonly, from the nucleotide sequences directly [8,13].

A similar model to synteny blocks are **locally-collinear blocks**, or LCBs: homologous blocks that might have multiple instances, with fully resolved overlap and free of non-linear rearrangements inside. In other words, LCBs can be seen as the borders of the whole-genome alignment blocks. LCBs can be seen as synteny blocks of the lowest resolutions and are used as a precursor for large-scale synteny blocks [11,12,14]. In addition, LCBs can be easily turned into the whole genome alignment by globally aligning sequences constituting a block.

Whole-genome homology mapping [15] also uses genomic intervals. Such mapping contains pairs of (possibly overlapping) intervals corresponding to homologous sequences. Analogously to locally-collinear blocks, this mapping can be seen as a set of borders of pairwise alignment blocks.

### 1.4 Challenges in computing sequence homology for long sequences

Computing sequence homology for shorter sequences, like bacterial genomes, is a more or less solved problem. For example, there are multiple tools for computing whole-genome alignment, and they can align tens or even hundreds genomes at once [8]. Unfortunately, datasets containing long genomes is a much more difficult problem. In a recent competitive assessment of whole-genome aligners only a handful of tools could align 20 flies [6], and most of them required a large computer cluster. Application of these tools to mammalian-sized genomes requires even more computational resources.

There are several aspects making this problem challenging. Let us consider multiple whole-genome alignment as an example as it is the most general formulation of homology finding problem. First, even global alignment of multiple genomes is NP-hard [16]. As the result, all real-world implementations are based on heuristics like progressive alignment [17]. In addition, when considering homology at the whole-genome scale, one has to take into account possibility of genome rearrangement disrupting the order of homologous nucleotides. The necessity of taking rearrangements into account makes algorithm design even more difficult. Second, many genomes are riddled with repeats of large multiplicity, for example transposable elements and genes belonging to large families. These repeats clog up data structures and slow down the algorithms. Although high-frequency repeats can be often filtered out, in certain contexts it could be undesirable, e.g. when analyzing regions containing genes having many homologs. Third, the sheer size of the data require very efficient algorithms operating on dense data structures.
We believe that the way to address these issues is to employ data structures that can store and index large collections of genomic sequences efficiently. A promising data structure fitting this purpose is compacted de Bruijn graph. Ordinary version of the graph contains all exact string matches of length $k$ in the input genomes and can be used as an index. The compacted de Bruijn graph represents long exact matches of length at least $k$ as single edges, which saves a lot of time and space. This property is especially useful for representing similar genomes as they often have long stretches of identical nucleotides.

Ubiquitously used for genome assembly [18–24], de Bruijn graph also found some applications in comparative genomics [8, 11, 14, 25–27]. However, these algorithms were limited to handling shorter sequences, like bacterial genomes. Scaling those algorithms to mammalian-sized genomes, including construction of the graph itself, remained an unsolved problem, which we aimed to address in this dissertation.

1.5 Contributions and organization of the dissertation

In this dissertation, we address the challenge of computing sequence homology for datasets consisting of multiple mammalian-sized genomes. Our approach is to use the compacted de Bruijn graph for indexing the genomes. First, we come up with a fast algorithm for constructing the said graph. Then we use this graph constructor to compute homology between sequences in the form of whole-genome alignment and pairwise homology maps.

Chapter 2 introduces TwoPaCo, a scalable method for constructing the compacted de Bruijn graph from multiple completed genomic sequences, which allows handling tens of long genomes. Paper depicting TwoPaCo was published in journal Bioinformatics [28]. Then, Chapter 3 describes SibeliaZ, a method for whole-genome alignment of closely-related genomes relying on TwoPaCo for indexing the input genomes. Manuscript reporting SibeliaZ is available as a preprint on biorXiv server [29]. In Chapter 4 we show BubbZ, an efficient method for computing pairwise whole-genome homology maps for collections of long genomes. Finally, Chapter 5 contains concluding remarks and possible directions for future work.
Chapter 2  
TwoPaCo: An efficient algorithm to build the compacted de Bruijn graph from many complete genomes

2.1 Introduction

The study of related features across different genomes is fundamental to many areas of biology, such as pan-genome analysis and comparative genomics. These studies often start with a representation of the relationship between genomes as a multiple alignment [30] or as a graph [31]. With the ubiquity of cheap sequencing, the number of genome sequences available for these studies has expanded tremendously [32–34]. The type of genomes available has also expanded: we have whole genomes, as opposed to only genic sequences, and we now have many mammalian sized (∼3 Gbp) genomes. In addition, novel long-read sequencing technologies like Oxford Nanopore promise to make such genomes even easier to obtain. Thus, we expect to have hundreds of whole mammalian genome sequences for comparison, in both the population and comparative genomic settings. However, our current computational ability to analyze such large datasets is, at best, limited.

A major bottleneck toward the goal of comparing hundreds of whole mammalian genomes are scalability issues due to the problem of repeats. Multiple alignment is a computationally hard problem due to the presence of high copy-count repeats, which are absent in many lower-order species but cover roughly half of a mammalian genome. For example, the human genome contains over a million ALU repeats. Most multiple alignment methods mask repeats due to the computational challenge of handling them, resulting in a loss of important features. Without masking repeats, most approaches do not scale well to modern data, both in terms of computation time and memory usage. A competition of whole-genome aligners demonstrated that some recent tools are able to handle larger data sets; however, these were still limited to
≤ 20 genomes of length < 200 Mbp [6].

As an alternative to multiple alignment, de Bruijn (or the closely related A-Bruijn) graph approaches for comparing whole genome sequences have been proposed [8,11,14,35]. De Bruijn graphs have traditionally been used for de novo assembly [23,24], but in the case of already assembled genomes, they are built from a few long sequences, as opposed to billions of short reads. In the setting of population genomics, a de Bruijn graph representation of closely related genomes can be used to discover polymorphism in a population [25,36]. In metagenomics, the de Bruijn graph was used as a reference representation for read mapping [37,38] and to predict virulence in bacteria [39]. In the comparative genomics setting, a de Bruijn graph representation can be used to detect synteny blocks [8,11]. Other graph representations besides the de Bruijn graph have also been proposed [36,40].

The use of de Bruijn or related graphs brings up a host of algorithmic questions that have been studied: how to construct those graphs efficiently [26,41–46], how to design fast querying indices [47–49], how to align read data to such graphs [50–52], and how to efficiently represent them in memory. Proposed representations of the de Bruijn graph include succinct [53,54], compacted [8,26,41–43], and Bloom filter based [55,56].

In this paper we study the efficient construction of the compacted de Bruijn graph. In a compacted de Bruijn graph, non-branching paths are replaced by single edges, which results in an equivalent, but smaller graph. The construction of such a graph is a resource intensive step and often poses the major bottleneck in applications. There have been recent tools to tackle the problem of efficiently constructing the compacted graph in the whole genome sequence setting: Sibelia [8], SplitMEM [26], and the tools of [42] and [43]. The fastest algorithm to date was able to process seven whole mammalian genomes in under eight hours [43]. However, constructing the compacted graph is still prohibitive for larger inputs.

In this paper, we present TwoPaCo, a novel algorithm for constructing the compacted de Bruijn graph from whole genome sequences. We demonstrate that it can construct the graph for 100 human genomes in less than a day and eight primates in less than two hours, on a typical shared-memory machine. TwoPaCo is based on the following key insight. We start with a basic naive algorithm, which has a prohibitively large memory usage but has the benefit that it is easily parallelizable. We then create a two pass algorithm that uses the naive one as a subroutine. In the first pass, we use a probabilistic data structure to drastically reduce the size of the problem, and in the second pass, we run the naive algorithm on the reduced problem. One of our key design principles was to make the algorithm simple and embarrassingly parallelizable, in order to take advantage of multi-thread support of most shared-memory servers. We also developed a procedure that splits the input into subsets that can be processed independently. As a result, TwoPaCo can trade-off memory usage for the running time, enabling processing large datasets on machines with small memory. The result is a simple and scalable low memory algorithm for the direct construction of the compacted de Bruijn graph for a set of complete
2.2 Methods

2.2.1 Preliminaries

For a string $x$, we denote by $x[i..j]$ the substring from positions $i$ to $j$, inclusive of the endpoints. We say that a string $x$ is the prefix of a string $y$, if $x$ constitutes the first $|x|$ characters of $y$, where $|x|$ is the length of $x$. A string $x$ is the suffix of a string $y$, if $x$ constitutes the last $|x|$ characters of $y$. At first, we define the de Bruijn graph built from a single string. For a string $s$ and an integer $k$, we designate the de Bruijn graph as $G(s, k)$. Its vertex set consists of all substrings of $s$ of length $k$, called $k$-mers. Two vertices $u$ and $v$ are connected with a directed edge $u \rightarrow v$ if $s$ contains a substring $e$, $|e| = k + 1$ such that $u$ is the prefix of $e$ and $v$ is the suffix of $e$. We will use terms “$k$-mer” and “vertex” interchangeably, as well as “$(k + 1)$-mer” and “edge.” For clarity of presentation, we have defined the de Bruijn graph as a simple graph, but we in fact store it as a multi-graph.

Now we define the de Bruijn graph for multiple strings. The union of two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ is the graph $G_1 \cup G_2 = (V_1 \cup V_2, E_1 \cup E_2)$. For a collection of strings $S = \{s_1, s_2, \ldots, s_n\}$ and an integer $k$, the de Bruijn graph is the union of the graphs constructed from individual strings, i.e. $G(S, k) = G(s_1, k) \cup G(s_2, k) \cup \ldots \cup G(s_n, k)$. Fig. 4.1a shows an example of a graph built from two strings. Recall that a path through a graph is a sequence of adjacent vertices where the only repeated vertices may be the first and last one, whereas a walk can repeat both vertices and edges. We say that a walk or path $p$ in the de Bruijn graph $G(S, k)$ spells a string $t$ if $G(t, k) = p$. We say that a vertex $v$ is a bifurcation if at least one of the following holds (1) $v$ has more than one incoming edge (2) $v$ has more than one outgoing edge. A vertex $v$ is a sentinel if it is a first or last $k$-mer of an input string. We call a vertex a junction if it is a bifurcation, or a sentinel, or both. The set $J(s, k)$ is the set of positions $i$ of the string $s$ such that the $k$-mer $s[i..i + k - 1]$ is a junction.

For a collection of strings $S$ the set $J(S, k)$ is defined analogously.
Figure 2.1: The de Bruijn graph and its compacted version. (a) An example of an ordinary de Bruijn graph built from the genomes \( S = \{ "TGGCACGTC", "TGGCAGTC" \} \) and \( k = 2 \). Junctions are indicated by square vertices. (b) Graph obtained after compaction. (c) The two genomes that generate the graph, with the junction \( k \)-mers in bold; the arrows between them indicate edges in the compacted graph and non-branching paths in the ordinary graph. The strings between them label the edges in the compacted graph. (d) If we store edges in a Bloom filter, we may observe false edges (dotted line) in the ordinary graph; this can lead to detection of false junctions, like the vertex “GC” in this case.

A de Bruijn graph can be compacted by collapsing non-branching paths into single edges. More precisely a non-branching path in an ordinary de Bruijn graph is a path \( u \leadsto v \) (a path from \( u \) to \( v \)) such that the only junction vertices on this path are possibly \( u \) or \( v \).

The compaction of a non-branching path \( p = u \leadsto v \) is removal of edges of \( p \) and replacing it with an edge \( u \rightarrow v \).\(^1\) A maximal non-branching path is a non-branching path that cannot be extended by adding an edge. The compacted graph \( G_c(S,k) \) is the graph obtained from \( G(S,k) \) by compaction of all its maximal non-branching paths. This graph is sometimes referred to as the compressed graph in the literature [42]. It is easy to see that the vertex set of \( G_c(S,k) \) is the set of junctions of the graph \( G(S,k) \) and two vertices \( u \) and \( v \) of \( G_c(S,k) \) are connected if there is a non-branching path \( u \leadsto v \) in \( G(S,k) \). Fig. 4.1b shows an example of a compacted de Bruijn graph. Note that a compacted graph is a multi-graph: after compaction a pair of vertices can be connected by edges going in the same direction that corresponded to different paths in the ordinary graph.

Graph compaction is the first step of most algorithms working with de Bruijn graphs, since it drastically reduces the number of vertices. It can be obtained from the ordinary graph in linear time by a simple graph traversal. However, building and storing the ordinary graph takes lots of space, which we seek to avoid in our algorithm by constructing the compacted graph

\(^1\)The graph is output such that for each vertex we maintain a list of its occurrences in the input so that one can easily reconstruct the original path.
directly.

A **Bloom filter** is a space efficient data structure for representing sets that supports two operations: storing an element in the set and checking if an element is in the set [57]. A Bloom filter offers improvements in space usage but can generate false positives during membership queries. Bloom filters have previously been successfully applied to assembly [55, 58–60] and to indexing and compression of whole genomes as well as large RNA-seq datasets [48, 61, 62]. In particular, they have been applied to the closely related problem of constructing and compacting a de Bruijn graph from short read sequences. While this paper addresses the whole genome setting, we find that the Bloom filter remains useful to represent a set of $k$-mers.

### 2.2.2 Reduction to the problem of finding junction positions

TwoPACo is based on the observation that there is a bijection between maximal non-branching paths of the de Bruijn graph and substrings of the input whose junctions are exactly the two flanking $k$-mers (Observation 1 below). This observation reduces the problem of graph compaction to finding the set of junction positions $J(S, k)$, as follows. The vertex set of the compacted graph is the set of all $k$-mers located at positions $J(S, k)$. To construct the edges, we need to find substrings flanked by junctions. To do this, we can traverse positions of $J(S, k)$ in the order they appear in the input. For every two consecutive junction positions $i$ and $j$, we record an edge between the $k$-mer at $i$ and the $k$-mer at $j$. Fig. 4.1c shows an example of how sequences of junctions generate non-branching paths in the ordinary graph and edges in the compacted one.

The observation follows in a straightforward way from the definitions, but we state and prove it here for completeness.

**Observation 1.** Let $s$ be an input string and $P$ be the set of maximal non-branching paths of the graph $G(s, k)$. Let $T$ be the set of substrings of $s$ such that each $t \in T$ starts and ends with a junction of $G(s, k)$ and does not contain junctions in between. Then there exists a bijective function $g : T \rightarrow P$.

**Proof.** Let $g$ be the function mapping substrings of $s$ to walks in $G(s, k)$, where $g$ maps a substring to the vertices corresponding to its constituent $k$-mers. To prove that $g$ is a bijection when restricted to $T$, we have to show that it is both an injection and surjection. Note that $g$ is injective by construction, that is, any walk is spelled by a unique string. To prove that it is surjective, we need to show that for any maximal non-branching path $p = u \sim v$, there is a $t \in T$ such that $g(t) = p$. That is, $p$ is spelled by a string in $T$. Since the walk $g(s)$ must traverse all vertices in the graph, and the internal vertices of $p$ have in- and out- degrees equal to one, the walk $g(s)$ must contain $p$ as a subwalk. Hence, the string $t$ spelled by $p$ must be a substring of $s$, i.e. $g(t) = p$. The internal $k$-mers of $t$ are non-junctions because $p$ is non-branching, and the first and last $k$-mers of $t$ are junctions because $p$ is maximal. Hence, $t \in T$. 

9
Algorithm 1 Filter-Junctions

**Input:** strings $S = \{s_1, \ldots, s_n\}$, integer $k$, and an empty set data structure $E$. A candidate set of marked junction positions $C \supseteq J(S,k)$ is also given. When the algorithm is run naively, all the positions would be marked.

**Output:** a reduced candidate set of junction positions.

1: for $s \in S$ do
2:     for $1 \leq i < |s| - k$ do
3:         if $C[s,i] = \text{marked}$ then ▷ Insert the two $(k+1)$-mers containing the $k$-mer at $i$ into $E$.
4:             Insert $s[i..i+k]$ into $E$.
5:             Insert $s[i-1..i+k+1]$ into $E$.
6: for $s \in S$ do
7:     for $1 \leq i < |s| - k$ do
8:         if $C[s,i] = \text{marked}$ and $s[i..i+k-1]$ is not a sentinel then ▷ Consider possible edges and count how many of them exist
9:             $in \leftarrow 0$ ▷ Number of entering edges
10:            $out \leftarrow 0$ ▷ Number of leaving edges
11:     for $c \in \{A, C, G, T\}$ do ▷ The symbol $\cdot$ depicts string concatenation
12:         if $v \cdot c \in E$ then
13:             $out \leftarrow out + 1$
14:     if $c \cdot v \in E$ then
15:         $in \leftarrow in + 1$
16:         if $in = 1$ and $out = 1$ then ▷ If the $k$-mer at $i$ is not a junction.
17:             $C[s,i] \leftarrow \text{Unmarked}$
18: return $C$

Generalization of the observation to the case of multiple strings is straightforward.

### 2.2.3 Single round algorithm

In the previous section, we reduced the problem of constructing a compacted de Bruijn graph to that of finding the locations in the genome where junction vertices are located. We will now present our algorithm for finding junction positions, in increasing layers of complexity. First, we will describe Algorithm 1, which can already be used as a naive algorithm to identify the junctions. However, Algorithm 1 alone has a prohibitively large memory footprint. To address this, we will present Algorithm 2, which uses Algorithm 1 as a subroutine but reduces the memory requirements. In cases of very large inputs, even Algorithm 2 can exceed the available memory. In Section 2.2.4, we finally present Algorithm 3, which addresses this limitation. It limits memory usage, at the expense of time, by calling Algorithm 2 over several rounds. We refer to this final algorithm (Algorithm 3) as TwoPaCo.

In Algorithm 1, we start with a candidate set $C$ of junction positions in the genomes. A set of positions $C$ is called a candidate set if $C \supseteq J(S, k)$ and any two positions that start with the
same \( k \)-mer can be either both present or both absent from \( C \). \( C \) is represented using boolean flags which mark every position of the genomes which is present in the set. If Algorithm 1 is used naively, it would be called with every position marked; in general, however, we can use \( C \) to capture the fact that the unmarked positions have been previously eliminated from consideration as junctions.

First, we store all edges of the ordinary de Bruijn graph in a set \( E \). We do this by a linear scan and for a \((k + 1)\)-mer at position \( i \) in a string \( S \), if either of the \( k \)-mers at positions \( i \) or \( i + 1 \) are marked, we insert the \((k + 1)\)-mer into the set \( E \) (Lines 1 to 5). Second, we again scan through the genomes and consider the \( k \)-mer \( v \) at every marked position. We use \( E \) to check how many edges in \( G(S, k) \) enter and leave \( v \) (Lines 9 to 15). Since the DNA alphabet is finite, we can do this by merely considering all eight possible \((k + 1)\)-mers– four entering, and four leaving – and checking whether they are in \( E \). If the in- and out-degrees do not satisfy the definition of a junction, we unmark position \( i \); otherwise, we leave it marked.

Algorithm 1 can be used naively to find all junction positions, by initially marking every position as a potential junction. Storing the set \( E \) in memory, however, is infeasible for large datasets. To reduce the space requirements, we develop the two pass Algorithm 2. In the first pass, we run Algorithm 1, but use a Bloom filter to store the set \( E \) instead of a hash table. A Bloom filter takes significantly less space than a hash table; however, the downside is that it can generate false positives during membership queries. That is, when we check if a \((k + 1)\)-mer is present in \( E \) (Lines 12 and 14 in Algorithm 1) we may receive an answer that it is present, when it is in reality absent. The effect is that the calculated in- and out-degrees may be inflated and we may leave non-junctions marked (Line 17), see Fig. 4.1d. Nevertheless, the marked positions still represent a candidate set of junctions, since a junction will never be unmarked. Thus, running Algorithm 1 with the Bloom filter reduces memory but does not always unmark non-junction positions. In order to eliminate these marks, we run Algorithm 1 again, using the positions marked in the first pass as a starting point, but this time using a hash table to store \( E \) (Line 4 in Algorithm 2). This second pass will unmark all remaining marked non-junction positions. Since the set of candidate marks has been significantly reduced after the first pass, the memory use of the hash table is no longer prohibitive. As with Algorithm 1, Algorithm 2 can be used to find all junction positions by initially marking every position as a potential junction.

Our implemented algorithms also handle the reverse complementarity of DNA, using standard techniques. We summarize this briefly for the sake of completeness. For a string \( s \), let \( \bar{s} \) be its reverse complement, and define the comprehensive de Bruijn Graph as the graph \( G_{\text{comp}}(s, k) = G(s, k) \cup G(\bar{s}, k) \); the graph for multiple strings and the compacted graph is defined analogously. To build the compacted comprehensive graph, we have to modify Algorithm 1 so that \( E \) represents each \( k \)-mer and its reverse complement jointly. For example, this can be done by always storing the canonical form of a \( k \)-mer, which is the lexicographically smallest
Algorithm 2  *Filter-Junctions-Two-Pass*

**Input:** strings \( S = \{s_1, \ldots, s_n\} \), integer \( k \), a candidate set of junction positions \( C_{in} \), integer \( b \)

**Output:** a candidate set of junction positions \( C_{out} \)

1. \( F \leftarrow \) an empty Bloom filter of size \( b \)
2. \( C_{temp} \leftarrow \text{Filter-Junctions}(S, k, F, C_{in}) \)  
   \( \triangleright \) The first pass
3. \( H \leftarrow \) an empty hash table
4. \( C_{out} \leftarrow \text{Filter-Junctions}(S, k, H, C_{temp}) \)  
   \( \triangleright \) The second pass
5. \( \text{return } C_{out} \)

String between the \( k \)-mer and its reverse complement [46]. Similarly, we have to be careful when we make membership queries to \( E \) in Algorithm 1, so that we are always querying canonical \( k \)-mers.

### 2.2.4 Multiple rounds: dealing with memory restrictions

While Algorithm 2 significantly reduces the memory usage, it is still possible that the hash table in the second pass may not fit into the main memory, for some very large inputs. To deal with this issue, we develop Algorithm 3, which splits the input \( k \)-mers into \( \ell \) parts and runs Algorithm 2 in \( \ell \) rounds. In each round, Algorithm 3 will consider only approximately \( 1/\ell \) of the \( k \)-mers to check if they are junctions. Each round processes only one part, thus decreasing memory usage. When \( \ell = 1 \), Algorithm 3 reduces to Algorithm 2 and does not limit its memory use, but when \( \ell \) is increased, the peak memory usage decreases at the expense of more rounds and hence longer running time.

Suppose that the set of \( k \)-mers is partitioned into \( \ell \) classes \( V_1, \ldots, V_\ell \). Then, in round \( i \), our algorithm begins by marking the positions whose \( k \)-mers are in class \( V_i \) (Line 15). Note that each position is considered in exactly one round. We then call Algorithm 2, which unmarks those positions which are not junctions. After all the rounds are complete, the junction vertices are exactly those that remain marked (Line 17).

To obtain the partition classes \( V_i \), we first note that the maximum memory usage of Algorithm 3 is minimized when the partition leads to an equally sized hash table in every round. To achieve this, we would like the sizes of sets \( V_i \) to be as equal as possible. We are not concerned with obtaining an optimal partition, since a small discrepancy in the memory in each round is permissible. Also note that we must partition the \( k \)-mers, which is different from partitioning the positions. In particular, if two different positions have the same \( k \)-mer, they must belong to the same class; hence, we cannot simply divide our strings into chunks.

Our idea is based on a two-step process. First, we partition the universe of all \( k \)-mers into \( q \gg \ell \) parts (e.g. \( q = 2^{32} \)). This is done implicitly by defining a uniform hash function \( f \) over the universe of \( k \)-mers with range \([0, q)\). A \( k \)-mer \( h \) belongs to part \( i \) iff its hash value \( f(h) = i \). We then obtain the number of input \( k \)-mers that belong to each part, \( c_0, \ldots, c_{q-1} \), as
follows (Lines 1 to 8). We make a pass through the input and use a Bloom filter to store all the k-mers. For every k-mer, if it is not already present in the Bloom filter, we increase the corresponding counter. This way, we try to count only unique k-mers, though the count can be slightly inflated by false positives. Due to the heuristic nature of our partitioning process, we can tolerate slightly inaccurate counts. Notice that the partition defined by $f$ is not necessarily balanced when applied to the input k-mers, i.e. there might be a big difference between $c_i$ and $c_j$, for some $i$ and $j$. However, this initial partition is much more fine grained than we need, which makes it a useful starting point.

Second, we obtain our desired partition into $V_1, \ldots, V_\ell$ by agglomerating consecutive parts of our fine-grained partition. Specifically, we implicitly define $V_1, \ldots, V_\ell$ using a sequence of integers $0 = p_0 \leq p_1 \leq \cdots \leq p_\ell - 1 \leq p_\ell = q$, where a $k$-mer $h$ belongs to $V_i$ iff $p_{i-1} \leq f(h) < p_i$.

To find a sequence $p_0, \ldots, p_\ell$ that would create a balanced partition, we use a greedy heuristic (Lines 10 to 12). It makes a linear scan through the counts $c_0, \ldots, c_{q-1}$, and fills the current partition with as many $k$-mers as possible, until the number of $k$-mers exceeds the $1/\ell$ of the total $k$-mers.

Algorithm 3 TwoPaCo

**Input:** strings $S = \{s_1, \ldots, s_n\}$, integer $k$, integer $\ell$, integer $b$

**Output:** the compacted de Bruijn graph $G_c(S, k)$

1: Initialize counters $c_0, \ldots, c_{q-1}$ to zeroes
2: $F \leftarrow$ an empty Bloom filter of size $b$
3: for $s \in S$ do
4:     for $1 \leq i \leq |s| - k + 1$ do
5:         $h \leftarrow s[i..i+k-1]$
6:         if $h$ not in $F$ then
7:             Insert $h$ into $F$
8:             $c_f(h) \leftarrow c_f(h) + 1$
9:     $T \leftarrow \sum_{0 \leq t < q} c_t / \ell$ \quad \text{$\triangleright$ Mean number of $k$-mers per partition}$
10:    $p_0 \leftarrow 0, p_\ell \leftarrow q$
11: for $1 \leq i < \ell$ do
12:      $p_i \leftarrow$ biggest integer larger than $p_{i-1}$ such that $(\sum_{p_{i-1} \leq j < p_i} c_j) \leq T$, or $\min\{\ell, p_{i-1} + 1\}$ if it does not exist.
13: $C_{\text{init}} \leftarrow$ boolean array with every position unmarked
14: for $1 \leq i \leq \ell$ do
15:      $C_i \leftarrow$ mark every position of $C_{\text{init}}$ that starts a $k$-mer $h$ with hash value $p_{i-1} \leq f(h) < p_i$
16: $C'_{\text{final}} \leftarrow$ Filter-Junctions-Two-Pass($S, k, b, C_i$)
17: $C_{\text{final}} = \bigcup C'_{\text{final}}$
18: return Graph implied by $C_{\text{final}}$, as described in Section 2.2.2.
2.2.5 Parallelization scheme

We designed our algorithm so that it can be effectively parallelized on a multi-processor shared memory machine. The bulk of the computation happens in Algorithm 1, which consists of two parts. Each part is a loop over all the positions in the input, Lines 1 to 5 in the first part and Lines 6 to 17 in the second. The first loop is embarrassingly parallelizable as long as the data structure representing the set \( E \) supports concurrent writes. We use a lock-free Bloom filter when Algorithm 1 is called during the first pass of Algorithm 2, and a concurrent hash table when it is called during the second pass. The second loop is trivially parallelizable: threads will get non-overlapping portion of genomes, hence the synchronization on \( C \) is not needed. A synchronization barrier separates the two loops. The compacted edge generation step that we discussed in the Section 2.2.2 is embarrassingly parallelizable as well.

We implement the parallelization using the standard single producer/multiple consumer pattern [63]. According to this design pattern we create (1) a single reader thread that splits the input into equal sized substrings and puts them into worker queues, and (2) many worker threads that dequeue and process the substrings. We utilized parallel programming primitives from the Intel’s Threading Building Blocks library [64]. Note that this way we store only part of the input and the corresponding array \( C \) in the input to save memory.

2.2.6 Theoretical analysis and comparison

In this section, we will analyze the running time and memory usage of our algorithm, and compare it with that of other algorithms. Suppose that the de Bruijn graph \( G(S, k) \) has \( E \) edges, \( J \) junctions and \( L \) non-junctions that we call links. First, we will analyze the number of false positive junctions. A false positive junction is a link whose positions in \( S \) are incorrectly left marked at the end of the first pass. We assign an indicator variable \( I_\ell \) to each link \( \ell \), \( I_\ell = 1 \) if the link \( \ell \) is a false positive junction and \( I_\ell = 0 \) otherwise. This way, the total number of false positive junctions is \( FP = \sum_{1 \leq \ell \leq L} I_\ell \). Let the probability that a link is a false positive junction be \( p \). By linearity of expectation we have \( \mathbb{E}[FP] = \mathbb{E}\left[\sum_{1 \leq \ell \leq L} I_\ell\right] = Lp \). To calculate the probability \( p \), note that each link has exactly one incoming and one outgoing true edge. Hence, querying the Bloom filter in Line 12 and Line 14 of Algorithm 1 may discover at most six false edges: three incoming and three outgoing ones. At least one false positive from those six queries results in the link misclassified as a junction. [65] show that the probability of a single false positive resulting from querying a Bloom filter is \( q = (1 - e^{-hE/b})^h \). where \( h \) is the number of hash functions used by the Bloom filter and \( b \) is the number of bits in the filter. Assuming that queries are independent, \( p = 1 - (1 - q)^6 = 1 - (1 - (1 - e^{hE/b})^h)^6 \).

Now we will analyze the running time. Let \( m \) be the total length of the input strings. First, note that storing and querying \( k \)-mers with the Bloom filter requires calculation of \( h \) hash values for each operation. We use a family of sliding window hash functions, so both filling
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Running Time</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sibelia</td>
<td>$O(m)$</td>
<td>$O(m)$</td>
</tr>
<tr>
<td>SplitMEM</td>
<td>$O(m \log g)$</td>
<td>$O(m +</td>
</tr>
<tr>
<td>bwt-based*</td>
<td>$O(m)$</td>
<td>$O(m)$</td>
</tr>
<tr>
<td>TwoPaCo</td>
<td>$O(mh + (</td>
<td>G_c</td>
</tr>
</tbody>
</table>

*from [43]

Table 2.1: Running times and memory consumption of different algorithms for constructing the de Bruijn graph from multiple complete genomes. For SplitMEM $g$ stands for the size of the longest genome in the input. An explanation of other variables is given in the Section 2.2.6.

and querying the Bloom filter in the first pass takes $O(mh)$ operations. In the second pass the algorithm employs a hash table to store and query $(k+1)$-mers. Denote by $M$ the number of marks left in the array $C$ after the first pass. The expected running time is then $O(mh + Mk)$, since each hash table operation takes $k$ time and there are $O(M)$ operations total. To calculate $M$, let us assume that the average number of times a false positive junction occurs in all the input strings is given by $r$. Then, the expected value of $M$ is $|G_c| + Lpr$, where $|G_c|$ is the number of edges in the compacted de Bruijn multi-graph. The expected running time is then $O(mh + (|G_c| + Lpr)k)$.

To calculate the memory usage, note that the first pass allocates $b$ bits of memory for the Bloom filter and the second pass uses a hash table that contains at most $8(J + FP)$ elements. Hence, the expected memory usage is $O(\max[b, (J + Lp)k])$. The array $C$ of marks is accessed sequentially by the algorithm and can be stored in the external memory without loss of performance. As discussed in Section 2.2.5, at each moment the memory contains only a constant amount of characters of the input strings, so the input length does not contribute to the asymptotic bound.

Table 2.1 contains asymptotic upper bounds on memory usage and running times of different algorithms for constructing the compacted de Bruijn graph from multiple complete genomes. The performance of TwoPaCo depends highly on the number of junctions present. On practical instances of related genomes datasets, there is a lot of shared sequence and the number of junctions is low. Unlike other algorithms, our expected memory usage depends only on the structure of the input, but not directly on its size. At the same time, dependence on $k$ makes TwoPaCo less applicable in case of very large $k$.

### 2.3 Results

To evaluate the performance of TwoPaCo, we conducted several experiments. We compared its running time and memory footprint with other available implementations of de Bruijn graph compaction algorithms. We then ran TwoPaCo on a real dataset of biological interest as well
as a large dataset of simulated data. We assessed the parallel scalability of our implementation and capabilities of running the algorithm on machines with limited memory using the round splitting procedure. Finally, we evaluated the effects of input length and structure on the running time and memory usage.

First, we benchmarked TwoPaCo against Sibelia [8], SplitMEM [26] and the bwt-based algorithm of [43], using default parameters. As far as we understood, the algorithm in [42] was subsumed by [43]. There were two important caveats. First, in most genomics application, it is necessary to account for both strands in the de Bruijn graph. To make SplitMem and bwt-based work with both strands, we appended the reverse complements of the sequences to the input, as suggested by their authors. In our results, we show SplitMEM and the bwt-based in two versions: (1) considering only one strand, and (2) considering both strands. Second, both minia and Sibelia not only constructs the compacted graph but also modifies it after construction. We therefore ran these tools only in the mode where they construct the graph only (contrary to the bechmarks in [26]). In addition to whole genome tools, one can also apply tools from genome assembly to construct the compacted graph. In this case, one would run a k-mer counter on the genomes, and then run a graph compaction tool on the resulting k-mers. Note that graph compaction algorithms that start with a set of k-mers must build non-branching paths by combining appropriate k-mers, while in the whole genome setting the non-branching paths are contained in the input (per Observation 1).

We tested how well these pipelines would compare against specialized approaches for whole genomes. We tried two pipelines: minia 2.0.3 [55] and the DSK 2.1.0 k-mer counter [66] followed by BCALM [46]2. We allowed them to use the maximum memory. Although minia has some parallelism, its graph compaction implementation is single-threaded. We allowed minia and DSK to use all 15 threads. BCALM is single-threaded. Supplementary material for the paper contains the exact command lines we used for benchmarking.

For benchmarking purposes, we used the following datasets: (1) 62 E.coli genomes (310 Mbp) from [26], and (2) seven human genomes (∼21 Gbp) used by [43] which includes five different assemblies of the human reference genome and two paternal haplotypes of NA12878 (see [43] for more details).

We ran our experiments on the highest memory Amazon EC2 instance (r3.8xlarge): a server with 32 Intel Xeon E5-2670 processors and 244 GB of RAM. We set the default number of internal hash functions in the Bloom filters to four. We also verified the correctness of TwoPaCo by comparing its output to that of a naive compaction algorithm on feasible test cases. A direct comparison to the output of other tools is impractical since each algorithm handles edges cases differently (e.g. the presence of undetermined nucleotides (Ns) in the input).

The results are shown in the first four rows of Table 2.2. For seven human genomes,
TwoPaCo was run using just one round, with a Bloom filter size $b = 0.13$ GB for *E. coli*, 4.3 GB for 7 humans with $k = 25$, $b = 8.6$ GB with $k = 100$, $b = 34$ GB for primates, and $b = 69$ GB for (43+7) and larger human dataset. A dash in the SplitMem and *bwt-based* columns indicates that they ran out of memory, a dash in the Sibelia column indicates that it could not be run on such large inputs, a dash in the minia column indicates that it did not finish in 48 hours, a dash in the BCALM column indicates that it ran out of disk space (4 TB). A double dash indicates that the software had a segmentation fault. An empty slot indicates that the experiment was not done.

TwoPaCo was at least 7 times faster than the second best algorithm, when we used 15 threads. When only a single thread was used, TwoPaCo was still slightly better than the second best DSK+BCALM for $k = 25$, and 2.5 – 3.4 times faster than the second best *bwt-based* on $k = 100$.

We also assessed TwoPaCo’s ability to handle (1) large numbers of long closely-related genomes, and (2) more divergent genomes. To do so, we generated 93 human genomes using the
FIGG genome simulator [68] and “normal” simulation parameters. The FIGG genome simulator generates complete sequences based on a reference genome and variations’ frequencies extracted from the datasets from projects like [69] and [70]. The mutations comprise single-nucleotide alterations as well as indels and structural variations of larger size. We ran TwoPaCo on two datasets: (1) 43 simulated genomes plus the seven used in Table 2.2, (2) 93 simulated human genomes plus the seven. The results are shown in the last five rows of Table 2.2. We construct the graph for 100 human genomes in 23 hours using 77 GB of RAM and 15 threads. For eight primates, we used under two hours and 34-62 GB of RAM on 15 threads.

To measure the parallel scalability of TwoPaCo, we fixed a dataset consisting of five simulated human genomes. Figure 2.2 shows scaling results for 1-32 worker threads. The first pass of Algorithm 2, and the conversion of junction vertices to the graph (as described in Section 2.2.2), scale almost linearly up to 16 threads. The second pass does not scale past four worker threads, due to what we believe is the limited parallel performance of the concurrent hash table, which we plan to improve in the future.

Next, we evaluated the performance of TwoPaCo under memory restrictions. For each run, we set a different memory threshold and checked how many rounds were necessary so that TwoPaCo did not exceed the threshold (Table 2.3). This experiment illustrates that TwoPaCo is capable of constructing the compacted graph for a dataset of five human genomes under memory restrictions commensurate with a low-end laptop.

For the benchmarks and real datasets in the experiments above, we recorded the number of marks that Algorithm 2 left in the array \( C \) after each stage (Table 2.4). We did not record those numbers for the larger datasets due to the associated cost restrictions of re-running the larger experiments.

Our last experiment assessed the effects of the input size and structure (number of junctions and number of distinct \( k \)-mers) on running time and memory consumption (Figure 2.3). As expected from the theoretical analysis, the running time depends both on the input size and structure, while memory consumption depends only on structure. For example, consider the dataset from [43], which has highly similar genomes. As a result, the number of distinct \( k \)-mers and junctions is nearly constant even as the number of genomes increases. This dataset has the lowest running time, and the amount of memory TwoPaCo uses does not increase with the number of genomes. Unlike the memory usage, the running time does see a dominant effect of the input size, as the running time increases with the number of genomes for this dataset. On the other hand, consider the primates dataset, which is more variable and contains more distinct \( k \)-mers and junctions than the simulated human dataset. As a result, TwoPaCo takes a longer time and has larger memory consumption.
Table 2.3: The minimal number of rounds it takes for TwoPaCo to compress the graph without exceeding a given memory threshold, using five simulated human genomes. Memory quantities are in gigabytes and running times are in minutes. It was carried out on a machine with an Intel Xeon E7-8837 processor. We used $k = 25$ and ran the computation with eight worker threads. In each run we used the largest possible Bloom filter size that fitted a given restriction (in our implementation the number of bits it has to be a power of two).

<table>
<thead>
<tr>
<th>Memory threshold</th>
<th>Used memory</th>
<th>Bloom filter size</th>
<th>Running time</th>
<th>Rounds</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>8.62</td>
<td>8.59</td>
<td>259</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>6.73</td>
<td>4.29</td>
<td>434</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>5.98</td>
<td>4.29</td>
<td>539</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>3.51</td>
<td>2.14</td>
<td>665</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 2.4: Number of marks in the array $C$ initially and after each pass of Algorithm 2.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Initially (Total Positions)</th>
<th>First Pass</th>
<th>Second Pass</th>
</tr>
</thead>
<tbody>
<tr>
<td>62 $E. coli$ ($k = 25$)</td>
<td>310,157,564</td>
<td>24,649,489</td>
<td>24,572,562</td>
</tr>
<tr>
<td>62 $E. coli$ ($k = 100$)</td>
<td>310,157,489</td>
<td>22,848,018</td>
<td>9,492,091</td>
</tr>
<tr>
<td>7 humans ($k = 25$)</td>
<td>21,201,290,922</td>
<td>3,489,946,013</td>
<td>2,974,098,154</td>
</tr>
<tr>
<td>7 humans ($k = 100$)</td>
<td>21,201,290,847</td>
<td>1,374,287,870</td>
<td>188,224,214</td>
</tr>
<tr>
<td>8 primates ($k = 25$)</td>
<td>24,540,556,921</td>
<td>5,423,003,377</td>
<td>5,401,587,503</td>
</tr>
<tr>
<td>8 primates ($k = 100$)</td>
<td>24,540,556,846</td>
<td>1,174,160,336</td>
<td>502,441,107</td>
</tr>
</tbody>
</table>

2.4 Discussion

In this paper we gave an efficient algorithm for constructing the compacted de Bruijn graph for a collection of complete genomic sequences. It is based on identifying the positions of the genome which correspond to vertices of the compacted graph. TwoPaCo works by narrowing down the set of candidates using a probabilistic data structure, in order to make the deterministic memory-intensive approach feasible. We note that the effectiveness of the algorithm relies on having whole genome sequences, making it inapplicable to the case when genomes are represented as shorts read fragments. Parallel speedup of the second pass of Algorithm 2 is an important direction of the future work that we are going to pursue.

A critical parameter of the TwoPaCo is the size of the Bloom filter ($b$). We recommend the user to set $b$ to be the maximum memory they wish to allocate to the algorithm. If the memory usage then exceeds $b$ (which would happen due to the size of the hash table), then the number of rounds should be increased until the memory usage falls below $b$. In future work, we plan to implement an algorithm to automatically select a value of $b$ that would minimize the maximum memory used by the algorithm. We also plan to automate the choice of the number of rounds, given a desired memory limit.

The algorithm can also be used to construct a partially compacted graph by omitting the
Figure 2.3: Effects of the input length and structure on the memory and running time. Here we varied the number of input genomes from one to seven and recorded the running time (a) and memory usage (b). We also calculated the number of distinct k-mers (c) and junctions (d) in the input to illustrate their effect on the algorithm’s performance. We used three datasets: simulated humans, primates, and 7 human assemblies from [43]. The experiment was performed on a machine with a Intel Xeon E7-8837 processor. We used $k = 25$ and ran the computation with eight worker threads and a single round. For each run we used the optimal Bloom filter size, i.e. the filter size that minimizes the maximum memory consumption. The number of distinct k-mers was computed using the KMC2 k-mer counter [71]. In our implementation, the number of bits in the Bloom filter has to be a power of two, which leads to the non-smooth growth of the memory curve in (b).

second pass of Algorithm 2. A partially compacted graph is one where some, but not necessarily all, of the non-branching paths have been compacted. Partially compacted graphs are faster to construct and can be useful in applications when the size of the graph is not critical or full compaction takes too much resources.

TwoPACo makes significant progress in extending the number and size of genomes from which a compacted de Bruijn graph can be constructed. We believe that this progress will enable novel biological analyses of mammalian-sized genomes. In this paper, we focus on the graph compaction algorithm only, and present a detailed analysis of its performance, both theoretical and experimental. However, future applications of the compacted de Bruijn graph is an exciting and important question. The de Bruijn graph can be the core of a tool to answer biological
questions, however, such a tool must not only construct the graph but implement additional algorithms to analyze it. For example, the synteny block reconstruction tool Sibelia [8] not only builds the compacted de Bruijn graph but also performs iterative graph simplification while increasing the value of $k$. Another very recent example is the use of the de Bruijn graph to construct the Burrows-Wheeler transform of many complete genomes [72]. TwoPACo can also be useful in other applications, such as the representation of multiple reference genomes or variants between genomes. See [73] for a detailed discussion of applications of the de Bruijn graph in computational pan-genomics. Similar efforts are under way in the GA4GH technical group. TwoPACo could be particularly useful on poorly assembled draft genomes, whose accurate alignment is especially challenging.
Chapter 3  
Scalable multiple whole-genome alignment and locally collinear block construction with SibeliaZ

3.1 Introduction

Multiple whole-genome alignment is the problem of identifying all the high-quality multiple local alignments within a collection of assembled genome sequences. It is a fundamental problem in bioinformatics and forms the starting point for most comparative genomics studies, such as rearrangement analysis, phylogeny reconstruction, and the investigation of evolutionary processes. Unfortunately, the presence of high-copy repeats and the sheer size of the input make multiple whole-genome alignment extremely difficult. While current approaches have been successfully applied in many studies, they are not able to keep up with the growing number and size of assembled genomes [6].

There are two common strategies to tackle the whole-genome alignment problem [74]. The first one is based on finding pairwise local alignments [75–79] and then extending them into multiple local alignments [4,80–82]. While this strategy is known for its high accuracy, a competitive assessment of multiple whole-genome alignment methods (Alignathon, [6]) highlighted several limitations. First, many algorithms either do not handle repeats by design or scale poorly in their presence, since the number of pairwise local alignments grows quadratically as a function of a repeat’s copy number. In addition, many algorithms use a repeat database to mask high-frequency repeats. However, these databases are usually incomplete and even a small amount of unmasked repeats may severely degrade alignment performance. Second, the number of pairwise alignments is quadratic in the number of genomes, and only a few existing approaches could handle more than ten fruit fly genomes [6]. Therefore, these approaches are ill-suited for large numbers of long and complex mammalian genomes, such as the recently
assembled 16 strains of mice [83].

Alternatively, anchor based strategies can be applied to decompose genomes into locally collinear blocks [84]. These are blocks that are free from non-linear rearrangements, such as inversions or transpositions. Once such blocks are identified, they can independently be globally aligned [35, 84–87]. The problem of constructing blocks form anchors is known as the chaining problem which had been extensively studied in the past [88–90]. All of the methods applicable to datasets consisting of multiple genomes are heuristic since the exact algorithms depend exponentially on the number of genomes. Such strategies are generally better at scaling to handle repeats and multiple genomes since they do not rely on the computationally expensive pairwise alignment.

A promising strategy to find collinear blocks is based on the compacted de Bruijn graph [8, 11, 14] Though these approaches do not work well for divergent genomes, they remain fairly accurate for closely related genomes. For example, Sibelia [8] can handle repeats and works for many bacterial genomes; unfortunately, it does not scale to longer genomes. However, the last three years has seen a breakthrough in the efficiency of de Bruijn graph construction algorithms [26, 28, 67, 91]. The latest methods can construct the graph for tens of mammalian genomes in minutes rather than weeks. We therefore believe the de Bruijn graph approach holds the most potential for enabling scalable multiple whole-genome alignment of closely related genomes.

In this paper, we describe a novel algorithm SibeliaZ-LCB for identifying collinear blocks in closely related genomes. SibeliaZ-LCB is suitable for detecting homologous sequences which have evolutionary distance to the most recent common ancestor (MRCA) of at most 0.08 substitutions per site. SibeliaZ-LCB is based on the analysis of the compacted de Bruijn graph and uses a graph model of collinear blocks similar to the “most frequent paths” introduced by [92]. This allows it to maintain a simple, static, data structure, which scales easily and allows simple parallelization. Thus, SibeliaZ-LCB overcomes a bottleneck of previous state-of-the-art de Bruijn graph-based approaches [11, 35], which relied on a dynamic data structure which was expensive to update. Further, we extend SibeliaZ-LCB into a multiple whole-genome aligner called SibeliaZ. SibeliaZ works by first constructing the compacted de Bruijn graph using our previously published TwoPaCo tool [28], then finding locally collinear blocks using SibeliaZ-LCB, and finally, running a multiple-sequence aligner (spoa, [93]) on each of the found blocks. To demonstrate the scalability and accuracy of our method, we compute the multiple whole-genome alignment for a collection of recently assembled strains of mice. We also test how our method works under different conditions, including various levels of divergence between genomes and different parameter settings. Our software is freely available at https://github.com/medvedevgroup/SibeliaZ/.

23
3.2 Methods

3.2.1 Preliminaries

First, we will define the de Bruijn graph and related objects. Given a positive integer \( k \), we define a multigraph \( G(s, k) \) as the de Bruijn graph of \( s \). The vertex set consists of all substrings of \( s \) of length \( k \), called \( k \)-mers. For each substring \( x \) of length \( k + 1 \) in \( s \), we add a directed edge from \( u \) to \( v \), where \( u \) is the prefix of \( x \) of length \( k \) and \( v \) the suffix of \( x \) of length \( k \). Each occurrence of a \((k + 1)\)-mer yields a unique multiedge, and every multiedge corresponds to a unique location in the input. Two edges are parallel if they have the same endpoints. Parallel edges are not considered identical.

The de Bruijn graph can also be constructed from a set of sequences \( S = \{s_1, \ldots, s_n\} \). This graph is the union of the graphs constructed from the individual strings: \( G(S, k) = \bigcup_{1 \leq i \leq n} G(s_i, k) \). See Fig. 4.1 for an example.

The set of a multiedges in a graph \( G \) is denoted by \( E(G) \). We write \((u, v)\) to denote a multiedge from vertex \( u \) to \( v \). A walk \( p \) is a sequence of multiedges \(((v_1, v_2), (v_2, v_3), \ldots, (v_{|p| - 1}, v_{|p|}))\) where each multiedge \((v_i, v_{i+1})\) belongs to \( E(G) \). The length of the walk \( p \), denoted by \(|p|\), is the number of multiedges it contains. The last multiedge of a walk \( p \) is denoted by \( \text{end}(p) \) and the first one by \( \text{start}(p) \). A simple path is a walk that visits each vertex at most once. We use the term path to refer to a simple path.

In a de Bruijn graph, a given multiedge \( x \) was generated by a \((k + 1)\)-mer starting at some position \( j \) of some string \( s_i \). We denote \( \text{gen}(x) = i \) and \( \text{pos}(x) = j \). For a multiedge \( x \), its successor, denoted by \( \text{next}(x) \), is a multiedge \( y \) such that \( \text{gen}(x) = \text{gen}(y) \) and \( \text{pos}(y) = \text{pos}(x) + 1 \). Note that a successor does not always exist. A walk \( p = (x_1, \ldots, x_{|p|}) \) is genomic if \( \text{next}(x_i) = x_{i+1} \) for \( 1 \leq i \leq |p| - 1 \). In other words, a walk is genomic if it was generated by a substring in the input. The \( b \)-extension of a genomic walk \( p \) is the longest genomic walk \( q = (y_1, \ldots, y_{|q|}) \) such that \( y_1 = \text{next}(\text{end}(p)) \) and \( |q| \leq b \). The \( b \)-extension of a walk \( p \) is uniquely defined and usually has length \( b \), unless \( p \) was generated by a substring close to an end of an input string. The concatenation of two walks \( x \) and \( y \) is a walk (if it exists) \( xy \) consisting of multiedges of \( x \) followed by multiedges of \( y \).

3.2.2 Problem formulation

In this section, we will define the collinear block reconstruction problem in de Bruijn graphs. A collinear block is a set of multiedge-disjoint genomic walks with length at least \( m \), where \( m \) is a parameter. We call walks constituting a collinear block collinear walks. In order to quantify how well collinear walks correspond to homologous sequences, we will define a collinearity score of a collinear block. Our problem will then be to find a set of collinear blocks that are pairwise multiedge-disjoint and have the largest score.

We capture the pattern of two homologous collinear walks through the concept of chains...
and bubbles. A **bubble** is a subgraph corresponding to a possible mutation flanked by equal sequences. Formally, a pair of walks $x$ and $y$ form a bubble $(x, y)$ if all of the following holds: (1) $x$ and $y$ have common starting and ending vertices; (2) $x$ and $y$ have no common vertices except the starting and ending ones; and (3) $|x| \leq b$ and $|y| \leq b$, where $b$ is a parameter. A **chain** $c = ((x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n))$ is a sequence of bubbles such that $x = x_1x_2\ldots x_n$ and $y = y_1y_2\ldots y_n$ are walks in a de Bruijn graph. In other words, a chain is a series of bubbles where each bubble is a “proper” continuation of the previous one. Note that two parallel edges form a bubble and a chain arising from equal sequences corresponds to a series of such bubbles. This way, a chain models a pair of sequences that potentially have point mutations and indels. For an example of a bubble and a chain, see Fig. 4.1.

The subgraph resulting from more than two collinear walks can be complex, and there are several ways of capturing it. Our approach is to give a definition that naturally leads itself to an algorithm.\footnote{We note that compared to the previous work on chaining anchors \cite{88–90} our definition of the block will be more relaxed. Namely, we will not require for common $k$-mers to be present in all copies of a block.} As homologous sequences all originate from some common ancestral sequence $s_a$, they should have many common $k$-mers and there should be a path $p_a = G(s_a, k)$ through
Figure 3.2: An example of computing the score of a walk $p$ (solid) relative to a carrying path $p_a = q_1q_2q_3$ (dashed). The path $p$ forms a chain with the subpath $q_2$ of $p$, while subpaths $q_1$ and $q_3$ form “hanging ends”. We count the length of $p$ and subtract lengths of the hanging ends. Thus, the score $f(p_a, p) = 4 - 1 - 1 = 2$.

The graph forming a long chain with each walk $p$ in the collinear block. We call such path a carrying one. We require the chains to be longer than $m$ to avoid confusing spuriously similar sequences with true homologs. At the same time, a collinear walk may only partially form a chain with the carrying path, leaving “hanging ends” at the ends of the carrying path, which is undesirable since it implies that these graphs originate from dissimilar sequences.

We formalize this intuition by introducing a scoring function quantifying how well a carrying path describes a collection of the collinear walks. The function rewards long chains formed by the carrying path and a collinear walk and penalizes the hanging ends. Given a carrying path $p_a$ and a genomic walk $p$, let $q_2$ be the longest subpath of $p_a$ that forms a chain with $p$. Then, we can write $p_a = q_1q_2q_3$. Recall that $m$ is the parameter denoting the minimum length of a collinear block, and $b$ is the maximum bubble size We define the score $f(p_a, p)$ as:

$$f(p_a, p) = \begin{cases} 
0, & \text{if } |p| < m \\
|p| - (|q_1| + |q_3|)^2, & \text{if } |p| \geq m \text{ and } |q_1|, |q_3| \leq b \\
-\infty, & \text{if } |p| \geq m \text{ and } (|q_1| > b \text{ or } |q_3| > b)
\end{cases}$$

The third case forbids walks (i.e. gives them a score of $-\infty$) where the hanging ends are too long, and the first case ignores walks (i.e gives them a score of 0) that weave through $p_a$ but are too short. The second case gives a score that is proportional to the length of the part of $p_a$ that forms a chain with $p$. At the same time, it reduced the score if the collinear walks leave hanging ends $q_1$ and $q_3$ — the parts of $p_a$ not participating in the chain. The penalty induced by these ends is squared to remove spuriously similar sequences from from the collinear block.

This form of scoring function showed better performance compared to other alternatives (data not shown). Fig. 3.2 shows an example of computing the score.

---

2We note that many alignment methods use phylogenetic information for scoring purposes. We decided to not use it since our method targets closely-related sequences, such as strains of the same species, where the phylogeny is often unknown.
The **collinearity score** of a collinear block is given by

\[ f(P) = \max_{p_a} \sum_{p \in P} f(p_a, p), \]

where \( p_a \) can be any path (not necessarily genomic). In other words, we are looking for a path forming longest chains with the collinear walks and thus maximizes the score. The collinear blocks reconstruction problem is to find a set of collinear blocks \( P \) such that \( \sum_{P \in P} f(P) \) is maximum and no two walks in \( P \) share a multiedge. Note that the number of collinear blocks is not known in advance. For an example of a complex collinear block in the de Bruijn graph and a carrying path capturing it, refer to Fig 4.1b.

### 3.2.3 The collinear blocks reconstruction algorithm

**Algorithm 4** Find-collinear-blocks

**Input:** strings \( S \), integers \( k \), \( b \) and \( m \)

**Output:** a set of edge-disjoint subgraphs of \( G(S, k) \) representing collinear blocks

1. \( P \leftarrow \emptyset \) \hspace{1cm} \( \triangleright \) Collinear blocks
2. \( G \leftarrow G(S, k) \) \hspace{1cm} \( \triangleright \) Construct the multigraph
3. **for** all distinct pairs \( (u, v) \in E(G) \) **do** \hspace{1cm} \( \triangleright \) Check possible seeds
4. Initialize the current carrying path \( p_a \) with \( (u, v) \)
5. \( P \leftarrow \emptyset \) \hspace{1cm} \( \triangleright \) Sorted set of collinear walks forming chains with \( p_a \)
6. \( P_{\text{best}} \leftarrow \emptyset \) \hspace{1cm} \( \triangleright \) Highest-scoring collinear block induced by \( p_a \)
7. **for** multiedges \( x \in E(G) \) parallel to \( (u, v) \) not marked as used **do**
8. Add to \( P \) a new collinear walk consisting of \( x \)
9. **while** \( f(P) \geq 0 \) **do** \hspace{1cm} \( \triangleright \) Extend the carrying path as far as possible
10. \( Q \leftarrow \{q \mid q \text{ is the } b\text{-extension of a } p \in P\} \)
11. \( w_0 \leftarrow \text{last vertex in } P \)
12. \( t \leftarrow \text{a vertex, reachable from } w_0 \text{ via a genomic walk, that is visited by the most } \)
\( \text{walks of } Q. \)
13. Let \( r \in Q \) be the shortest walk from \( w_0 \) to \( t \)
14. Denote the vertices of \( r \) as \( w_0, w_1, \ldots, w_{|r|} \), with \( w_{|r|} = t \)
15. **for** \( i \leftarrow 1 \) **to** \( |r| \) **do**
16. Append \( (w_{i-1}, w_i) \) to the carrying path \( p_a \)
17. \( P \leftarrow \text{Update-collinear-walks}(P, w_i) \)
18. **if** \( f(P) > f(P_{\text{best}}) \) **then**
19. \( P_{\text{best}} \leftarrow P \)
20. **if** \( f(P_{\text{best}}) > 0 \) **then**
21. \( P \leftarrow P \cup \{P_{\text{best}}\} \)
22. Mark multiedges visited by walks of \( P_{\text{best}} \) as used
23. **return** \( P \)

Our algorithm’s main pseudocode is shown in Algorithm 4 and its helper function in Algorithm 5. First, we describe the high-level strategy. The main algorithm is greedy and
Algorithm 5 Update-collinear-walks

**Input:** A sorted set of collinear walks $P$, a vertex $w$

**Output:** Updated set $P$

1. for multiedges $x \in E(G)$ ending at $w$ not marked as used do
   2. Let $p \in P$ be a walk such that its $b$-extension $q$ contains $x$ and $pos(\text{end}(p))$ is maximized
      ▷ Find a walk extendable with $x$
   3. if such $p$ exists then
      4. Truncate $q$ so that $\text{end}(q) = x$
      5. Append $p$ with $q$ ▷ Lengthen the chain that $p$ forms with $p_a$
   6. else
      7. Add a new walk consisting of the multiedge $x$ to $P$

8. return $P$

Figure 3.3: Illustration for Algorithm 5. A collinear walk $p$ (solid) requires an update after the carrying path $p_a$ is extended with the dashed edge $(w_0, w)$. The path $p_a$ now ends at the vertex $w$, which has another incoming edge $x$. Since $x$ is a part of $b$-extension of $p$ (denoted by $q$), $p$ can be appended with $q$ to form a longer chain and boost the collinearity score.

works in the seed-and-extend fashion. It starts with an arbitrary multiedge in the graph, and tries to extend it into a carrying path that induces a collinear block with the highest possible collinearity score $P_{\text{best}}$. If the block has a positive score, then $P_{\text{best}}$ is added to our collection of collinear blocks $\mathcal{P}$. The algorithm then repeats, attempting to build a collinear block from a different multiedge seed. New collinear blocks cannot use multidges belonging to previously discovered collinear blocks. This process continues until all possible multiedges are considered as seeds. The algorithm is greedy in a sense that once a block is found and added to $\mathcal{P}$, it cannot be later changed to form a more optimal global solution.

To extend a seed into a collinear block $P$, we first initialize the collinear block with a walk for each unused multiedge parallel to the seed (including the seed) (lines 7 to 8). These parallel multiedges represent the different occurrences of the seed string in the input and, hence, form the initial collinear block. We then proceed in phases, where each phase is an iteration of the while loop (lines 9 to 19). During each phase, the carrying path $p_a$ is extended using a walk $r$ of length at most $b$ (lines 10 to 14). Next, we try to extend each of the collinear walks in a way that forms chains with the extended $p_a$ (lines 15 to 19). The extension of a seed into a collinear block is also a greedy process, since we only change $p_a$ and the walks in $P$ by extending them and never by changing any edges. Finally, we check that the collinearity score for our extended
block is still positive — if it is, we iterate to extend it further, otherwise, we abandon our attempts at further extending the block. We then recall the highest scoring block that was achieves for this seed and save it into our final result \( \mathcal{P} \) (lines 20 to 22).

To pick the walk \( r \) by which to extend \( p_a \), we use a greedy heuristic (lines 10 to 14). First, we pick the vertex \( t \) which we want to extension to reach (lines 10 to 12). We limit our search to those vertices that can be reached by a genomic walk from the end of \( p_a \) and greedily chose the one that is most often visited by the \( b \)-extensions of the collinear walks in \( P \). Intuitively, we hope to maximize the number of collinear walks that will form longer chains with \( p_a \) after its extension and thereby boost the collinearity score. We then extend \( p_a \) using the shortest \( b \)-extension of the walks in \( P \) to reach \( t \). We chose this particular heuristic because it showed superior performance comparing to other possible strategies.

Once we have selected the genomic walk \( r \) by which to extend \( p_a \), we must select the extensions to our collinear walks \( P \) that will form chains with \( p_ar \). This is done by the function \textit{Update-collinear-blocks} (Algorithm 5). We extend the walks to match \( r \) by considering the vertices of \( r \) consecutively, one at a time. To extend to a vertex \( w \), we consider all the different locations of \( w \) in the input (each such location is represented by a multiedge ending at \( w \)). For each location, we check if it can be reached by a \( b \)-extension from an existing \( p \in P \). If yes, then we extend \( p \) so as to lengthen the chain that it forms with \( p_a \). If there are multiple collinear walks that reach \( w \), we take the nearest one. If no, then we start a new collinear walk using just \( x \). Fig. 3.3 shows an example of updating a collinear walk and Fig. 3.4 shows a full run of the algorithm for a single seed.

Our description here only considers extending the initial seed to the right, i.e. using out-going edges in the graph. However, we also run the procedure to extend the initial seed to the left, using the in-coming edges. The case is symmetric and we therefore omit the details.

### 3.2.4 Other considerations

For simplicity of presentation, we have described the algorithm in terms of the ordinary de Bruijn graph; however, it is crucial for running time and memory usage that the graph is compacted first. Informally, the compacted de Bruijn graph replaces each non-branching path with a single edge. Formally, the vertex set of the compacted graph consist of vertices of the regular de Bruijn graph that have at least two outgoing (or ingoing) edges pointing at (incoming from) different vertices. Such vertices are called junctions. Let \( \ell = v_1, \ldots, v_n \) be the list of \( k \)-mers corresponding to junctions, in the order they appear in the underlying string \( s \). The edge set of the compacted graph consists of multiedges \( \{v_1 \rightarrow v_2, v_2 \rightarrow v_3, \ldots, v_{n-1} \rightarrow v_n\} \). We efficiently construct the compacted graph using our previously published algorithm TwoPaCo [28].

This transformation maintains all the information while greatly reducing the number of edges and vertices in the graph. This makes the data structures smaller and allows the algorithm
Figure 3.4: An example of running Algorithm 4 on the graph from Fig. 4.1b, starting from edge GC $\rightarrow$ CC as the seed. Each subfigure shows the content of the collinear block $P$ and the carrying path. The collinear walks are solid, the carrying path is dashed, and the rest of the graph is dotted. Subfigure (a) shows the state of these variables after the initialization; subfigures (b-d) show the state after the completion of each phase.

Our previous description of the algorithm remains valid, except that the data structures operate with vertices and edges from the compacted graph instead of the ordinary one. The only necessary change is that when we look for an edge $y$ parallel to $x$, we must also check that $y$ and $x$ spell the same sequence. This is always true in an ordinary graph but not necessarily in a compacted graph.

An important challenge of mammalian genomes is that they contain high-frequency $(k + 1)$-mers, which can clog up our data structures. To handle this, we modify the algorithm by skipping over any junctions that correspond to $k$-mers occurring more than $a$ times; we call $a$ the abundance pruning parameter. Specifically, prior to constructing the edge set of the
compacted de Bruijn graph, we remove all high abundance junctions from the vertex set. The edge set is constructed as before, but using this restricted list of junctions as the starting point. This strategy offers a way to handle high-frequency repeats at the expense of limiting our ability to detect homologous blocks that occur more than a times.

The organization of our data in memory is instrumental to achieving high performance. To represent the graph, we use a standard adjacency list representation, annotated with position information and other relevant data. We also maintain a list of the junctions in the paragraph above in the order they appear in the input sequences, thereby supporting `next()` queries. The walks in the collinear block $P$ are stored as a dynamic sorted set, implemented as a binary search tree. The search key is the genome/position for the end of each walk. This allows performing binary search in line 2 of Algorithm 5.

Another aspect that we have ignored up until now is that DNA is double-stranded and collinear walks can be reverse-complements of each other. If $s$ is a string, then let $\bar{s}$ be its reverse complement. We handle double strandedness in the natural way by using the comprehensive de Bruijn graph, which is defined as $G_{\text{comp}}(s, k) = G(s, k) \cup G(\bar{s}, k)$ [28]. Our algorithm and corresponding data structures can be modified to work with the comprehensive graph with a few minor changes which we omit here.

Our implementation is parallelized by exploring multiple seeds simultaneously, i.e. parallelizing the while loop at line 2 of Algorithm 4. This loop is not embarrassingly parallelizable, since two threads can start exploring two seeds belonging to the same carrying path. In such a case, there will be a collision on the data structure used to store used marks. To address this issue, we process the seeds in batches of fixed size. All the seeds within a batch are explored in parallel and the results are saved without modifying the “used” marks. Once the batch is processed, a single arbiter thread checks if there is any overlap in the used marks of the different threads. If there is, it identifies the sources of the conflict and reruns the algorithm at the conflicting seeds serially. Since most seeds do not yield valid carrying paths, such conflicts are rare. Once there is no conflict, the arbiter updates the used main data structures with the results of the batch. This design allows the computation result to be deterministic and independent of the number of threads used.

### 3.3 Results

#### 3.3.1 Datasets, tools, and evaluation metrics

Evaluation of multiple whole-genome aligners is a challenging problem in its own right and we therefore chose to use the practices outlined in the Alignathon [6] competition as a starting point. They present several approaches to assess the quality of a multiple whole-genome alignment. Ideally it is best to compare an alignment against a manually curated gold standard;
unfortunately, such a gold standard does not exist. We therefore chose to focus our evaluation on real data.

We evaluated the ability of SibeliaZ to align real genomes by running it on several datasets consisting of varying number of mice genomes. We retrieved 16 mice genomes available at GenBank [94] and labeled as having a “chromosome” level of assembly. They consist of the mouse reference genome and 15 different strains assembled as part of a recent study [83] (Table B.1). The genomes vary in size from 2.6 to 2.8 Gbp and the number of scaffolds (between 2,977 and 7,154, except for the reference, which has 377). We constructed 4 datasets of increasing size to test the scalability of the pipelines with respect to the number of input genomes. The datasets contain genomes 1-2, 1-4, 1-8 and 1-16 from Table B.1, with the genome 1 being the reference genome.

To evaluate accuracy, we compared our results against annotations of protein-coding genes. We retrieved all pairs of homologous protein-coding gene sequences from Ensembl and then computed pairwise global alignments between them using LAGAN [5]. The alignment contains both orthologous and paralogous genes, though most of the paralogous pairs come from the well-annotated mouse reference genome. We removed any pairs of paralogous genes with overlapping coordinates, as these were likely mis-annotations, as confirmed by Ensembl helpdesk [95]. We made these filtered alignments as well as the alignments produced by SibeliaZ available for public download from our GitHub repository. We define the nucleotide identity of an alignment as the number of matched nucleotides divided by the length of an alignment, including gaps. The distribution of nucleotide identities as well as the coverage of the annotation is shown in Figure B.1. In our analysis, we binned pairs of genes according to their nucleotide identity.

In addition to computing recall (as defined below), we also measured coverage, which is the percent of the genome sequence that is included in the alignment. We could not measure the precision directly, since the annotation only included genes. A good proxy for precision could have been a statistical analysis of the alignment blocks, implemented by the PSAR tool [96]. However, [6] showed that PSAR score does not have a linear correlation with the true precision. Instead, we define a score corresponding to the percentage of nucleotide pairs that are identical in an alignment column. In an alignment of highly similar genomes that has high precision, we expect that the column scores in the alignment blocks are very high. Formally, given a column $C$ of a multiple whole-genome alignment with $C_i$ being its $i$-th element, the score is given by $f(C) = \sum_{1 \leq i \leq |C|} \sum_{i < j \leq |C|} I[C_i = C_j]/(\binom{|C|}{2})$. The variable $I[C_i = C_j]$ is equal to 1 if both $C_i$ and $C_j$ are the same valid DNA characters and 0 otherwise.

We employed the metrics of precision and recall used in the Alignathon and implemented by the mafTools package [6]. For these metrics, alignment is viewed as an equivalence relation. We say that two positions in the input genomes are equivalent if they originate from the same position in the genome of their recent common ancestor. We denote by $H$ the set of all equivalent
position pairs, participating in the “true” alignment. Let $A$ denote the relation produced by an alignment algorithm. The accuracy of the alignment is then given by $\text{recall}(A) = 1 - \frac{|H \setminus A|}{|H|}$ and $\text{precision}(A) = 1 - \frac{|A \setminus H|}{|A|}$, where $\setminus$ denotes set difference.

We benchmarked the performance of SibeliaZ against Progressive Cactus [7]. We also attempted to run Sibelia [8] (a predecessor of SibeliaZ) and MultiZ+TBA [4], but these could run to completion within a week on even a single mouse genome. Other multiple aligners [81, 87, 97] benchmarked in the Alignathon could not handle a dataset of 20 flies and hence are unlikely to scale to a mammalian dataset. We also chose not to run Mercator [85] since it requires a set of gene exons as input and hence solves a different problem: in this paper we focus on computing the whole-genome alignment directly from the nucleotide sequences without using external information. Further details about parameters, versions, and hardware are in B.1.

3.3.2 Running time and memory

The running times of SibeliaZ and Cactus are shown on Figure 3.5 (Table B.2 contains the raw values). On the dataset consisting of 2 mice, SibeliaZ is more than 10 times faster than Cactus, while on 4 mice SibeliaZ is more than 20 times faster. On the datasets with 8 and 16 mice, SibeliaZ completed in under 7 and 16 hours, respectively, while Cactus did not finish (we terminated it after a week). For SibeliaZ, we note that the global alignment with spoa takes 44-73% of the running time, and, for some applications (e.g. rearrangement analysis), this step may be further omitted to save time. Memory is shown in Table B.2. When it is able to complete, Cactus has better memory performance than SibeliaZ; however, both tools require memory that is well within the range of most modern servers but outside the range of personal computers.
3.3.3 Accuracy

In Table 3.1, we show the properties of the alignments found by SibeliaZ and Cactus. To compute recall, we only used nucleotides from gene pairs having at least 90% identity in the annotation. For the datasets where Cactus was able to complete, SibeliaZ has significantly better recall on paralogous pairs, and similar recall on orthologous pairs. The coverage of both tools is roughly the same, but SibeliaZ has only about half the blocks, indicating a less fragmented alignment. SibeliaZ’s coverage and recall decreases only slightly going up to the whole 16 mice dataset, indicating that the recall scales with the number of genomes.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>N. of blocks</th>
<th>Coverage</th>
<th>Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SibeliaZ</td>
<td>Cactus</td>
<td>SibeliaZ</td>
</tr>
<tr>
<td>1-2</td>
<td>2,083,258</td>
<td>4,228,063</td>
<td>0.88</td>
</tr>
<tr>
<td>1-4</td>
<td>2,739,821</td>
<td>6,133,662</td>
<td>0.86</td>
</tr>
<tr>
<td>1-8</td>
<td>3,179,619</td>
<td>-</td>
<td>0.89</td>
</tr>
<tr>
<td>1-16</td>
<td>4,507,109</td>
<td>-</td>
<td>0.88</td>
</tr>
</tbody>
</table>

Table 3.1: Properties of the multiple whole-genome alignments computed by SibeliaZ and Cactus from the mice datasets.

We further investigate how the recall behaved as a function of nucleotide identity, for the two- and four-mice dataset (Figure 4.2). As expected, recall decreases with nucleotide identity,
though SibeliaZ’s recall remains above 90% for nucleotides from similar (80-100% identity) orthologous genes. SibeliaZ has significantly higher recall than Cactus (89% vs < 20%) in highly similar (≥ 90%) paralogous genes. However, Cactus has slightly better recall in orthologous genes of lower identity on the two-mice dataset. Recall on orthologous gene pairs remains consistent in both two- and four-mice datasets for both datasets; however recall on paralogous gene pairs on the four-mice dataset decreases significantly for Cactus and stays nearly the same for SibeliaZ on highly similar genes.

We also observed that for both Cactus and SibeliaZ, the recall for paralogous nucleotides is lower than for orthologous ones (Figure 4.2). Based on manual inspection, we hypothesize that the problem is likely due to the complex repeat structures embedded inside these genes. These repeat structures of high and varying multiplicity make alignment of those genes challenging, despite their high nucleotide identity.

Since the gene annotation constitutes only a fraction of all the homologous nucleotides in the genomes, it cannot be used to directly access the precision. As a proxy for computing precision, we compute the scores $f(C)$ for the alignment columns (Figure B.2). SibeliaZ’s alignment has a high degree of similarity: 90% of the alignment columns have a score $f(C) \geq 0.9$, which is what we would expect from aligning closely related genomes. Cactus has slightly lower scores, which may simply indicate that it finds blocks with higher divergence.

We note that the results in this Section evaluate the accuracy of SibeliaZ-LCB and spoa simultaneously; however, since SibeliaZ is targeted at closely-related genomes, we observed that the global alignment procedure has a negligible effect on accuracy (data not shown). This is due to the fact that the global alignment of similar sequences is likely to be unambiguous at homologous nucleotides and robust with respect to different algorithms and their parameters.

### 3.3.4 Gene families

We wanted to further understand SibeliaZ’s ability to recall homologous nucleotides from large gene families. Aligning genes having many copies is a challenging task since they generate a tangled de Bruijn graph. To investigate, we took each pair of genes in the two-mice dataset that have greater than 90% nucleotide identity. We then identify any other homologous genes that had a nucleotide identity of at least 90% to one of the genes in the pair. We refer to the number of such genes as the inferred family size of the gene pair, which roughly corresponds to the gene family size in the biological sense. Figure B.3 then shows the recall of nucleotide pairs with respect to the inferred family size of their respective genes. The recall shows a lot of variance with respect to the inferred family size but does exhibit a general trend of decreasing with increasing family size. The largest bin (with inferred family size of 58) corresponds to a single large gene family on the Y chromosome ($PTHR19368$) and actually has relatively high recall.
3.3.5 Effect of parameters and sequence divergence

SibeliaZ-LCB has four primary parameters that affect its performance. The most critical dependence is on the size of a \( k \)-mer (i.e. \( k \)) and the maximum allowed length of a bubble \( b \). For a given sequence divergence, the distance between shared \( k \)-mers forming bubbles in homologous regions increases with \( k \). At the same time, the maximum allowed length of a bubble is \( b \). If the distance exceeds \( b \), then SibeliaZ may fail to uncover such regions and result in lower recall. To avoid this situation, we can either decrease \( k \) or increase \( b \). Decreasing \( k \) is desirable up to a point, but when \( k \) becomes too low, the de Bruijn graph becomes convoluted and our algorithm becomes more time and memory consuming. Increasing \( b \) can also be done but simultaneously increases the allowable gap length, leading to decreased precision.

Over-alignment is the problem of combining non-homologous sequences in a single block, which is closely related to low precision [98]. In our case, one can control over-alignment by looking at the \( f(C) \) scores, as we have done in our analysis (Figure B.2). A lower score indicates that more divergent sequences are included in a block. If the divergence is deemed too high by the user, it is recommended to reduce \( b \).

To investigate this complex interplay between \( k \) and \( b \) and its relationship to sequence divergence, we used simulations (B.2) to measure recall (Figure B.4) and precision (Figure B.5) under various combinations. As predicted, recall increases with decreasing \( k \) and with increasing \( b \), and precision decreases with increasing \( b \). We note though that the precision varies only a little and remains high. Based on these analyses, we recommend two values of \( k \) for practical usage. For less complex organisms (e.g. bacteria), we recommend \( k = 15 \), since it yields the
highest recall. This value is impractical for complex organisms (e.g. mammals) due to runtime, so we recommend setting $k = 25$ in those cases as it provides a reasonable trade-off between accuracy and required computational resources (we used this for our mice datasets). For the value of $b$, we observed that increasing $b$ lowers the precision at only higher values. Hence, we recommend $b = 200$ as the default in all cases, as it led to high recall across all tested ranges of $k$ on our simulated data without lowering precision.

To test the level of divergence which SibeliaZ-LCB can tolerate, we took the default values of $k = 15$ or 25 and $b = 200$ and plotted the precision vs. recall curve as a function of the root-to-leaf divergence of the dataset (Figure B.6). We see that for $k = 25$ the recall deteriorates significantly for datasets having a root-to-leaf evolutionary distance of more than 0.09 substitutions per site. Based on this, we recommend that for large datasets SibeliaZ-LCB be only used for detecting homologs with an evolutionary distance to the MRCA of at most 0.08 substitutions per site.

The other two parameters that can affect SibeliaZ-LCB’s performance are the minimum size of a locally collinear block $m$ and the abundance pruning parameter $a$. These parameters should be set according to the type of data and its intended use. The parameter $m$ controls the fragmentation of the alignment and the coverage — higher $m$ results in longer blocks spanning less of the genomes, since short blocks are not reported. We recommend the parameter $m$ to be set to the length of the shortest homologous sequence of interest to the downstream analysis. We set $m = 50$ as a default, since this is smaller than 93.1% of the known mice exons [99] and, more generally, we do not expect most applications to be interested in much blocks shorter than 50nt. In the case that a user is interested in larger homologous units, they can increase $m$ together with $b$. Alternatively, they can use either synteny block generation or alignment chaining algorithms for post-processing the alignments produced by SibeliaZ.

The abundance pruning parameter $a$ is a filtering parameter for $k$-mers whose abundance is above $a$. Such $k$-mers are still considered by SibeliaZ-LCB, but to a smaller extent, resulting in reduced recall in regions with such $k$-mers. We recommend setting $a$ as high as the compute resources allow, keeping in mind that homologous blocks with multiplicity higher than $a$ are possibly not going to be captured. For the mice dataset, we used $a = 150$.

### 3.4 Discussion

In this paper, we presented a novel whole-genome alignment pipeline SibeliaZ based on an algorithm for identifying locally collinear blocks. The algorithm analyses the compacted de Bruijn graph and jointly reconstructs the path corresponding to a collinear block and identifies the induced collinear walks. We assume that the collinear walks share many vertices with this carrying path and form chains of bubbles. Each carrying path and the induced block is found greedily, using a scoring function that measures how close it is to all the sequences in the block.
We then globally align the collinear blocks to generate the whole-genome alignment.

The main strength of our approach is speed — we achieve drastic speedups compared to the state-of-the-art Progressive Cactus aligner [7]. On 16 mice genomes, SibeliaZ runs in under 16 hours, while Progressive Cactus is not able to complete for even 8 mice genomes, within seven days. SibeliaZ also achieves higher recall of nucleotide pairs in highly similar paralogous genes, though it had slightly lower recall in orthologous ones. Overall, SibeliaZ is the only tool available that can scale to many long, closely-related genomes.

If the alignments themselves are not needed, SibeliaZ-LCB can be run alone (without spoa) to construct the collinear blocks. This is most useful in applications stemming from studies of genome rearrangements, which can be applied to study breakpoint reuse [100], ancestral genome reconstruction [101] and phylogenies [102]. Locally collinear blocks are also a required input for scaffolding tools using multiple reference genomes [103–106]. For such applications, the output of SibeliaZ-LCB can be used either directly or after postprocessing by a synteny block generator [11,12].

There are several remaining open questions of interest. A formal analysis of SibeliaZ-LCB’s runtime is relevant, but doing it in a useful way is a challenge. The worst-case running time does not reflect the actual one; moreover, we observed that the actual one depends on multi-thread synchronization, which is challenging to model. However, it would be interesting if such a time analysis can be performed parametrized by the crucial properties of the structure of the input. We also did not investigate how close to an optimal solution our greedy heuristic gets. One way to do this would be to find an optimal carrying path using exhaustive enumeration, but the search space even for a small realistic example is too big. We suspect that a polynomical time optimal solution is not possible, but the computational complexity of our problem is open.

SibeliaZ is the first multiple whole-genome aligner that can run in reasonable time on a dataset such as the 16 mouse genomes analyzed in this paper. With ongoing initiatives like the Vertebrate Genomes Project and the insect5k, thousands of species will soon have a reference genome available, and the sequencing and assembly of various sub-species and strains will be the next logical step for many comparative genomics studies. For example, [107] currently holds 18 assembled maize genomes, with more to come in the recent future. Similarly, the Solanaceae Genomics Network has recently released the genomes of 13 diverse tomato accessions⁴. Analysis of such datasets is likely to be carried out in single-lab settings with limited compute resources, rather than at large computing centers like EMBL or NCBI. SibeliaZ makes a significant leap towards enabling such studies.

⁴https://solgenomics.net/projects/tomato13/
Chapter 4
Scalable pairwise whole-genome homology mapping of long genomes with BubbZ

4.1 Introduction

Pairwise whole-genome homology mapping is the problem of finding all pairs of homologous intervals between a pair of genomes. Unlike local pairwise alignment, which provides base-to-base homology resolution, mapping only computes the boundaries of homologous blocks. This is, however, sufficient for many applications. For example, whole-genome homology mapping is a starting point for the analysis of genome rearrangements, which themselves are used in studies of breakpoint reuse [100] and phylogenetics [102]. It can be used as a precursor to whole-genome alignment [74,108] or for exploratory comparative analysis. It is also used as a tool for quality control of genome assembly [109] and for identifying genomic duplications for the purposes of improving RNA mapping [110].

A straightforward solution to compute such a homology map is to do pairwise local alignment; however, alignment is a harder computational problem, requiring more resources than other more direct approaches. A related problem is locally-collinear block reconstruction, where mapping blocks can have multiple instances and the overlap between them is fully resolved [8,11,84–87]. Although solutions to this problem can be used to compute pairwise mappings, known methods perform poorly in regions with complex repeat structures [29]. The first direct approach to address the whole-genome homology map problem was chaining of shorter fragments using a line-sweeping approach inspired by computational geometry [88–90,111]. Another approach treats sequences like audio signals and uses cross-correlations to find homology [112].

However, the number of available whole genomes has been rising dramatically in the last few years, creating the need for more scalable homology mappers. Two recent tools are
particularly notable in tackling this challenge; though both are known for read alignment, they also compute homology maps. Minimap2 [113] is based on a seed-and-extend approach, but using minimizers [114] to quickly identify and reduce the number seeds. MashMap2 [15] uses a minimizer winnowing scheme to quickly identify candidates. These methods are able to compute the mapping of two mammalian-sized genomes in less than an hour. However, in a scenario where the input is a set of multiple genomes and each genome has to be mapped to every other one, even these methods can be too slow. With efforts like the Vertebrate Genomes Project and Insect 5K promising to release thousands more genomes in the future, more scalable approaches will be needed.

The approach we take in this paper is based on the compacted de Bruijn graph. This graph provides an efficient representation of the shared \( k \)-mers between closely related genomes, whereby potentially large shared sequences are represented by small structures within the graph. Approaches based on such graphs had already proven useful to construct synteny blocks [8,11], but recent breakthroughs in the efficiency of graph construction algorithms [26,28,67,91] make them a promising approach for homology mapping. The latest methods can construct the graph for tens of mammalian genomes in minutes rather than weeks and could construct the graph for 100 human genomes in less than a day [28].

In this paper, we propose BubbZ, an algorithm for computing whole-genome pairwise homology mappings, especially in the context of all-to-all comparison for multiple genomes. Our algorithm is based on ideas similar to the line-sweep algorithms [89,90,111], but allows for more efficient data structures that reduce the running time. We evaluate our method on both simulated datasets and on a large dataset composed of 16 mouse genomes, showing approximately an order of magnitude speed improvement, compared to MashMap2 and Minimap2, while retaining similar accuracy. More importantly, BubbZ empirically scales linearly with respect to the number of input genomes, making it more practical as the number of assembled genomes continues to grow.

### 4.2 Methods

In this section, we first introduce de Bruijn graphs (Section 4.2.1) and then give the mathematical definition of homology (chains) that we will use (Section 4.2.2). In Section 4.2.3, we formulate the homology mapping problem. We then present our algorithm (Section 4.2.4) along with the data structures that make it efficient and other considerations (Section 4.2.5). Finally, we analyze the computational complexity of our algorithm (Section 4.2.6).
4.2.1 Preliminaries

Given a positive integer $k$ and a string $s$, we define a multigraph $G(s, k)$ as the de Bruijn graph of $s$. The vertex set consists of all substrings of $s$ of length $k$, called $k$-mers. By $s_i$ we denote the $k$-mer starting at position $i$ of $s$. For each $(k + 1)$-mer substring $x$ in $s$, we add a directed edge from $u$ to $v$, where $u$ is the prefix of $x$ of length $k$ and $v$ the suffix of $x$ of length $k$. Each occurrence of a $(k + 1)$-mer yields a unique multiedge, and every multiedge corresponds to a unique location in $s$. See Figure 4.1a for an example. Note that unlike some other definitions of a de Bruijn graph, a $(k + 1)$-mer that occurs multiple times in $s$ will have multiple corresponding edges. The de Bruijn graph for a set of sequences $S$ is the union of each of the edgesets corresponding to each string: $G(S, k) = \bigcup_{s \in S} G(s, k)$.

The set of a multiedges in a graph $G$ is denoted by $E(G)$. We write $(u, v)$ to denote a multiedge from vertex $u$ to $v$. A walk $w$ is a sequence of multiedges $((v_1, v_2), (v_2, v_3), \ldots, (v_{|w|}, v_{|w|+1}))$ where each multiedge $(v_i, v_{i+1})$ belongs to $E(G)$. The length of the walk $w$, denoted by $|w|$, is the number of multiedges it contains. A walk is genomic if it was generated by a substring in the input, that is, the edges correspond to consecutive locations the input.

For simplicity of exposition, we will describe our algorithm in terms of the de Bruijn graph. Our implementation, however, operates on the compacted de Bruijn graph. In its compacted version, the edges correspond to non-branching paths of the original graph. This way, it represents whole substrings as single edges, leading to substantial time and memory savings. We will not describe the compacted graph further, but the interested reader can find more details in [28].

4.2.2 Chains

Consider two chromosome sequences $s$ and $t$ and their de Bruijn graph $G = G(\{s\} \cup \{t\}, k)$. There are several ways to mathematically define a homologous pair of substrings from $s$ and $t$. The definition that lends itself to de Bruijn graph-based algorithms is that of a chain [88,115]. In our context, a chain is, informally, sequence of common $k$-mers that forms a sub-sequence (i.e. substrings allowing gaps) in both strings interleaved by potential point mutations or indels of bounded length. Formally, a chain $c$ of weight $n$ is two non-decreasing sequences of indices $(i_1, \ldots, i_n)$ and $(j_1, \ldots, j_n)$ such that $s_{i_x} = t_{j_x}$ and $i_x - i_{x-1} \leq b$ and $j_x - j_{x-1} \leq b$, for all $x$. Each chain is associated with two genomic walks in $G$; specifically, the genomic walk corresponding to the substring of $s$ starting from position $i_1$ and ending in position $i_n$, and, similarly, the genomic walk corresponding to the substring of $t$ from position $t_1$ to $t_n$. The walk length of a chain is the maximum of the lengths of the two walks constituting the chain. See Figure 4.1a for an example of a chain.

Let $c = ((i_1, \ldots, i_n), (j_1, \ldots, j_n))$ and $c' = ((i'_1, \ldots, i'_m), (j'_1, \ldots, j'_m))$ be two chains. The
Figure 4.1: (a) De Bruijn graph built from strings \(s=\text{"CACGTC"}\) and \(t=\text{"CACTTC"}\), with \(k=2\). The two strings are reflected by the blue and red walks, respectively. The whole graph is chain \(((1,2,5),(1,2,5))\). (b) State of the list \(Q\), after considering each position \(i\) of the string \(s\). \(Q_1\) is its contents after processing vertex “CA” (there is only one chain consisting of the initial \(k\)-mer); \(Q_2\) contains the extended chain; and \(Q_5\) has the whole graph minus the first chain that was removed due to being too far from the current position.

The concatenation of \(c\) and \(c'\) is the pair of sequences

\[
(c \cdot c') = ((i_1, \ldots, i_n, i'_1, \ldots, i'_m), (j_1, \ldots, j_n, j'_1, \ldots, j'_m))
\]

Note that \(c \cdot c'\) is a chain iff \(i'_1 \geq i_n, j'_1 \geq j_n\) and \(i'_1 - i_n, j'_1 - j_n \leq b\). In practice, we will be interested in the concatenation operation only if the result is a chain.

Chains can overlap in redundant ways, for example, one chain can be completely contained in another. To reduce this redundancy, it will be helpful to consider terminal chain. We say that a chain \(c\) is terminal if there is no other chain \(c'\) such that \(c \cdot c'\) is a chain. We note that it is easy to see if a chain is terminal, as follows. Let \(c\) be a chain ending in positions \(i\) and \(j\) in \(s\) and \(t\), respectively. If \(c\) is not terminal, then there must be some offsets \(1 \leq \alpha \leq b\) and \(1 \leq \beta \leq b\) such that \(s_{i+\alpha} = t_{j+\beta}\). In practice, \(b\) is quite small, and if appropriate mappings between shared \(k\)-mers in \(s\) and \(t\) are maintained, we can check if \(c\) is terminal quickly.

4.2.3 Problem formulation and recurrence solution

We formulate the homology problem as follows. The input is two chromosome sequences \(s\) and \(t\), and positive integer parameters \(b\) and \(m\). Let \(c(i,j)\) be the chain ending at positions \(i\) and \(j\) in \(s\) and \(t\), respectively, that is the “best” among all such chains (if such a chain exists). Let \(C = \{c(i,j) \mid 1 \leq i \leq |s|, 1 \leq j \leq |t|\}\). The homology problem is then to output all chains in \(C\) that are terminal and have walk length of at least \(m\).

A natural notion of “best” is a chain having the biggest weight among all such chains ending at \(i\) and \(j\), and such a definition was considered in other work [88–90, 111, 115]. However, we found that solving the problem exactly using such a definition is not conducive to an efficient algorithm (see more in the Discussion). Instead, we will use a heuristic that makes best chains easy to find while keeping in mind the goal of trying to maximize their weight.

Consider a pair of positions \(i\) and \(j\) such that \(s_i = t_j\) (i.e. they have the same \(k\)-mer). Let


[((i), (j))] be the chain consisting of that single k-mer. We define its possible left-extensions as the set of pairs \((i', j')\) such that \(((i'), (j')) \cdot ((i), (j))\) is a valid chain. Equivalently, these are the set of pairs \((i', j')\) such that \(s_{i'} = t_{j'}, \ i - b \leq i' \leq i, \) and \(j - b \leq j' \leq j\). We define the predecessor function \(\pi(i, j)\) as the left-extension with the largest value of \(j'\) and, if there are multiple, the one with the largest value of \(i'\). If there are no left extensions, we set \(\pi(i, j) = \emptyset\). Our notion of “best” is then to simply extend the closest nearby chain, i.e. the one ending at \(\pi(i, j)\). This gives rise to following recursive definition of \(c(i, j)\):

\[
c(i, j) := \begin{cases} 
\emptyset & \text{if } s_i \neq t_j, \\
((i), (j)) & \text{else if } \pi(i, j) = \emptyset, \\
c(\pi(i, j)) \cdot ((i), (j)) & \text{else}
\end{cases}
\] (4.1)

In other words, to obtain a chain with a large weight ending with a pair of k-mers at genomic positions \(i\) and \(j\), we append them to the closest preceding chain. The hope is that a chain that is as close as possible to \(i\) and \(j\) will have the largest weight maximizing the weight of \(c(i, j)\).

4.2.4 Algorithm

Equation (4.1) immediately lends itself to a naive greedy algorithm that progressively builds the chains and stores the results in a matrix where each cell corresponds to a row \(i\) and a column \(j\) (similar to a dynamic programming algorithm, but with a fixed choice for each cell). Such an algorithm can compute all \(c(i, j)\) but will use \(\Omega(|s||t|)\) memory, which is prohibitive. Instead, we present an algorithm that exploits the sparseness and structure of the matrix as well as the fact that the maximum gap is limited by parameter \(b\).

Let \(c(i', j') = c(\pi(i, j))\) denote the predecessor chain of \(c(i, j)\). First, we observe that if we compute the values of \(c(i, j)\) in increasing order of \(i\), we are guaranteed that the predecessor chain has already been computed, i.e. \(i' < i\). Second, by definition of a valid chain, the predecessor chain must lie within the \(b\) previous columns, i.e. \(i' \geq i - b\). Hence, it is not necessary to retain the whole table in memory, but rather, just the previous \(b\) columns. Third, the matrix is mostly sparse, since it only contains values when \(s_i = t_j\). Therefore, storing it as a matrix is impractical. Instead, we will store the elements of the previous \(b\) columns in an ordered and indexed set \(Q\), whose implementation we will in Section 4.2.5.

The pseudocode of our method is in Algorithm 6. The outer for loop iterates over all the values of \(i\). The inner for loop iterates over all values of \(j\) where \(c(i, j) \neq \emptyset\). Lines 4 through 8 implement the logic of Equation (4.1). When column \(i\) is finished, lines 10 through 14 update \(Q\) by removing all chains from the now outdated column \(i - b\) and, for those that are terminal and satisfy the length constraint, outputting them. For clarity, the pseudocode omits some corner cases (e.g. when \(i' = i\) or when we hit the end of the strings). Figure 4.1b shows an example of the contents of \(Q\) after several iterations.
Algorithm 6 Find-chains

**Input:** strings $s$ and $t$, graph $G(\{s\} \cup \{t, k\}$, integers $b$ and $m$

**Output:** the set of all chains in $C$ that are terminal and have walk length $\geq m$.

1. $Q \leftarrow$ an empty doubly-linked list $\triangleright$ The set of current chains $c(i,j)$
2. for $i \leftarrow 1$ to $|s|$ do
   3. for all $j$ such that $t_j = s_i$ do $\triangleright$ Consider all position of $k$-mer $s_i$ in $t$
      4. if $\pi(i,j) \neq \emptyset$ then
         5. $r \leftarrow \text{Lookup}(Q, \pi(i,j))$ $\triangleright$ Equation (4.1)
         6. $\text{PushBack}(Q, r \cdot ((i),(j)))$
      7. else
         8. $\text{PushBack}(Q, ((i),(j)))$
   9. let $c(i',j') \leftarrow \text{Front}(Q)$
10. while $i' < i - b$ do $\triangleright$ Cleaning-up and outputting $Q$
    11. if $c(i',j')$ has walk length $\geq m$ and is terminal then
        12. output $c(i',j')$
    13. $\text{PopFront}(Q)$
    14. let $c(i',j') \leftarrow \text{Front}(Q)$

The correctness of the algorithm follows from the previous discussion and the fact that $Q$ is maintained in a way that satisfies the following invariant:

**Lemma 1.** At the start of the $i$-th iteration, $Q$ contains $c(\pi(i,j))$ for all $j$ such that $\pi(i,j) \neq \emptyset$

**Proof.** We prove the lemma by by induction on $i$. For the base case ($i = 1$), the statement holds since there is no predecessor for a $c(1,j)$ and $Q$ is empty. We prove the induction step by contradiction. Suppose there exists a $c(\pi(i,j)) = c(i',j')$ and $c(i',j') \notin Q$. During the $i'$-th iteration we considered all pairs $((i'),(j'))$ such that $s_{i'} = t_j$. Since the statement holds for the iteration $i' < i$ by induction, we retrieved the appropriate predecessor $\pi(i',j') \in Q$, if one existed, and added $c(i',j')$ to $Q$ after implementing Equation (4.1) in Lines 4 to 8. Afterwards, in subsequent iterations preceding $i$, we only remove elements $c(i'',j'')$ with $i - i'' > b$, and hence did not remove $c(i',j')$ as $c(i',j') = \pi(i,j)$ which implies that $i - i' \leq b$. Hence, $c(i',j') \in Q$, contradiction.

4.2.5 Data structures and other considerations

To make Algorithm 6 efficient, we need additional data structures not described in the pseudocode. First, we implement mappings for the shared $k$-mers that allow us to efficiently access their positions in $s$ and $t$, as well as in $Q$ if they are there (we omit the details here). These mappings allow us to efficiently execute Line 3, i.e. to retrieve the positions in $t$ that have the same $k$-mer as $s_i$. 

44
Second, to support the computation of $\pi$ and the lookup operation for $Q$ (in Line 5), we need a specialized index. A straightforward solution would be to maintain a binary search tree over $Q$, but such an implementation can be too slow in practice. Instead, we keep a bit vector $e$ such that $e[j'] = 1$ iff $Q$ contains a chain $c(i', j')$, for some $i'$. This allows us to compute $\pi(i, j)$ as follows. First, we find the largest value $j'$ within the range of $j - b \leq j' \leq j$. We do this by looking up $b$ consecutive values in the $e$ bit array, which can be done fast on modern hardware by checking $b/64$ integers. Once we identify $j'$, then we use our $k$-mer-mappings to find the largest position $i' \leq i$ in $s$ where $t_{j'}$ occurs, from where we follow a pointer to the location of $c(i', j')$ in $Q$.

Algorithm 6 can also be used to find chains within a single genome, corresponding to duplications. To do this, the user should give as input a pair of identical sequences. To handle this case, we modify our algorithm to forbid chains that overlap with themselves. To implement this, we perform additional checks before concatenating chains in Line 4 (details omitted).

Algorithm 6 can also be used to compute an all-against-all mapping for a set of chromosomes $S = \{s_1, \ldots, s_{|S|}\}$. Rather than performing $\Theta(|S|^2)$ runs of the algorithm, we can modify the algorithm to run only $\Theta(|S|)$ times, at a potential cost of more memory, as follows. We first compute the de Bruijn graph from all of $S$, i.e. $G(S, k)$. Then we run Algorithm 6 $|S|$ times; in the $i$th run, $s_i$ plays the role of $s$ and the chromosomes $\{s_i, \ldots, s_{|S|}\}$ play the role of $t$. The only change we need to make to support a $t$ composed of multiple chromosomes is to consider positions in all sequences of $\{s_i, \ldots, s_{|S|}\}$ in Line 2. It may also be that the underlying graph $G(S, k)$ could have vertices from some chromosome $s_j$ that is not part of the comparison (i.e. $j < i$); however, since the algorithm only looks at $k$-mers that appear in $s$ or $t$, those extra $k$-mers would not effect the execution of the algorithm. This approach to all-against-all mapping will give the same results as the naive $O(|S|^2)$ runs approach. However, it does have an associated memory cost, since we must maintain in memory a de Bruijn graph of $|S|$ sequences, rather than just the graph of 2 sequences. This strategy also lends itself to parallelization, by executing these $|S|$ runs in parallel using multithreading. Finally, note that the same strategy applies to all-against-all mapping of multiple multi-chromosomal genomes, since our algorithm does not distinguish between chromosomes on the same vs. on different genomes.

There are additional aspects that the pseudocode does not address. We described the algorithm considering only the single strand of DNA. To handle both strands, we run a slightly modified version of our algorithm on the graph $G_{\text{comp}}(s, k) = G(s, k) \cup G(\bar{s}, k)$, where $\bar{s}$ is reverse complement of $s$ [28]. We also preprocess the graph by removing all $k$-mers occurring more than $a$ times, where $a$ is a parameter. High-frequency $k$-mers can clog up our data structures and slow down the algorithm. We allow the user to set $a$, thereby controlling the trade-off between speed and potential decrease in accuracy. Finally, to save space, we do not store the actual chains in $Q$, but only their starting and ending coordinates, since this is what the final mapping will return anyway.
4.2.6 Computational complexity

Here we will analyze the complexity of Algorithm 6. Let the maximum degree of a vertex considered by our algorithm be \( a \). This could just be the maximum degree in the graph or it could be the parameter \( a \) set by the user. The number of iterations of the inner loop in Lines 4 to 8 is bounded by \( a|s| \). Computing \( \pi \) and doing the lookup operation in Line 5 takes \( O(b) \) operations in the worst case, as described in Section 4.2.5. The processing of \( Q \) in Lines 10 to 14 takes a total of \( O(a|s|) \) time over the course of the algorithm, since this is the number of elements pushed into \( Q \). As the result, the total time complexity is \( O(ab|s|) \). The space complexity is dominated by the data structures to store the mappings for the shared \( k \)-mers. The amount of memory is strongly dependent on the structure of the input, and we therefore did not perform a worst case analysis.

4.3 Results

4.3.1 Datasets

We evaluated BubbZ speed and accuracy on two types of datasets, the first based on long real mouse genomes, and the second based on short simulated bacterial genomes. For the arge real genomes, we downloaded 16 mice genomes from GenBank [94]. These consisted of 15 different strains, assembled as part of a recent study [83], and the mouse reference genome. The mouse reference has 377 scaffolds, while the other mouse strains have 2,977-7,154 scaffolds; the genomes’ size fluctuates between 2.6 and 2.8 Gbp. To test the scalability of our pipeline in the number of genomes, we created four datasets from these 16 genomes. The four datasets contain genomes 1-2, 1-4, 1-8 and 1-16, respectively, with genome 1 being the reference genome. More details about the datasets, including accession numbers, are available as Table S1 in [29].

The other type of data we used were nine simulated datasets, generated as part of our earlier study [29] and available for download at https://github.com/medvedevgroup/SibeliaZ/blob/master/DATA.txt. Each dataset is an evolution simulation from a single ancestor genome, composed of 1500 genes and of size approximately 1.5 Mbp; the result is 10 genomes in each dataset. The datasets are distinguished by their divergence, with the evolutionary distance from the root genome to the leaves varying between 0.03 to 0.25 substitutions per site.

4.3.2 Evaluated tools

We compared BubbZ against the two recent tools that are able to scale to the size of modern datasets, Minimap2 [113] and MashMap2 [15]. We ran all tools in order to produce an all-against-all mapping, including any duplications (i.e. mappings within a single genome or chromosome). All parameters and command lines are available at https://github.com/medvedevgroup/
To run BubbZ, we first ran TwoPaco [28] to construct the graph, using \( k = 21 \) for real datasets and \( k = 15 \) for the smaller simulated ones. We then ran BubbZ using \( b = 300 \), \( m = 200 \), and \( a = 150 \) for all datasets. The role of these parameters was explored in the context of multiple whole-genome alignment of the same datasets [29], and we used values that were found to work best in that paper.

For Minimap2 and MashMap2, we created a separate run for each of the genome pairs. To find duplications, we also ran it on each genome separately. For mapping different genomes with Minimap2, we used default presets for sequences of 5% divergence. For mapping genomes against themselves, we used the parameters suggested by the author\(^1\). For MashMap2 we used the default parameters, except that for mapping different genomes we used the orthologous filtering, while for computing duplications we disabled the filtering, as suggested by the authors. For all datasets, we ran all tools with 24 threads, except TwoPACo, which we ran with 16 threads.

### 4.3.3 Evaluation metrics

For the smaller simulated dataset we computed both recall and precision using the mafTools package [6]. This package requires an alignment for comparison, rather then just a map; we therefore took each one of the homology blocks in our output and computed an alignments of it using LAGAN [5]. To define precision and recall, mafTools views an alignment as an equivalence relation which is the set of all equivalent position pairs participating in the true alignment. Let \( A \) denote the relation produced by an alignment algorithm, and let \( H \) denote the ground truth alignment relation (in our case, \( H \) is given by the simulator). The accuracy of \( A \) is then given as \( \text{recall}(A) = 1 - |H \setminus A|/|H| \) and \( \text{precision}(A) = 1 - |A \setminus H|/|A| \).

For the larger mouse dataset, there are unfortunately no ground-truth whole-genome homology maps or alignments available, making it difficult to evaluate precision. To evaluate the recall, we used an alignment of homologous protein-coding genes annotated in Ensembl. These ground-truth alignments, generated as part of our earlier study [29] using LAGAN, are available for download at https://github.com/medvedevgroup/SibeliaZ/blob/master/DATA.txt. The alignment contains both orthologous and paralogous gene pairs, though most of the paralogous pairs come from the well-annotated mouse reference genome. For the purposes of analysis, we binned the pairs of homologous genes according to the nucleotide identity in their alignment. These alignments cover around 33% of the input genomes, i.e. 33% of base pairs in the input genome are included in the alignment. We could not compute recall using mafTools due to the computational cost of having to compute all alignments for all the homologous intervals in the output. Instead, consider all the aligned position pairs in a ground truth alignment.

\(^{1}\)https://github.com/lh3/minimap2/issues/106
alignment, and a homology mapping $C$. We define recall as the fraction of aligned position pairs for which there exists a block in $C$ covering both positions.

### 4.3.4 Results on the mice data

The running time and memory consumption of all the tools are shown in Table B.2. The pipeline consisting of TwoPaCo and BubbZ was 4-15 times faster than Minimap2 and 5-20 times faster than MashMap2. Starting at four genomes, we observe roughly linear scaling for BubbZ. For Minimap2 the scaling seems superlinear, while for MashMap2 it seems linear but with a larger constant than BubbZ (though it is difficult to make any firm conclusions given the limited number of datapoints). The linear scaling of BubbZ is consistent with the fact that only a linear number of runs is required (see Section 4.2.5), however, the time of each run also grows with the size of the input (Section 4.2.6). Nevertheless, we empirically found the scaling to be roughly linear.

The memory trade-off of BubbZ is also readily apparent. It uses significantly more memory than others. As discussed in Section 4.2.5, this is due to it using a graph constructed for all the genomes at once, rather than just for the pair being considered.

To compute the recall, we only used the dataset consisting of two genomes, since computing recall is otherwise computationally prohibitive. Figure 4.2 show the recall, broken down by nucleotide identity of the gene pairs and by orthology / paralogy. For the orthologous genes, all mappers have similar recall, though BubbZ has higher recall in genes of lower nucleotide identity. For the paralogous pairs, Minimap2 had slightly higher recall then BubbZ. We could not compute recall for MashMap2 since it was run in non-filtering mode, which produces prohibitively large number of overlapping mappings and computing its recall is computationally unreasonable.

### 4.3.5 Results on the simulated data

Using simulated data, we can measure the accuracy more thoroughly than we could on real data. Figure B.6 shows precision and recall of the three methods, as a function of divergence.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>TwoPaCo + BubbZ</th>
<th>Minimap2</th>
<th>MashMap2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-2</td>
<td>15 (9.3)</td>
<td>9 (35.2)</td>
<td>24 (35.2)</td>
</tr>
<tr>
<td>1-4</td>
<td>22 (9.4)</td>
<td>12 (66.5)</td>
<td>34 (66.5)</td>
</tr>
<tr>
<td>1-8</td>
<td>40 (9.3)</td>
<td>24 (94.9)</td>
<td>64 (94.9)</td>
</tr>
<tr>
<td>1-16</td>
<td>83 (17.8)</td>
<td>42 (164.2)</td>
<td>125 (164.2)</td>
</tr>
</tbody>
</table>

Table 4.1: Running time (minutes) and memory usage (gigabytes, in parenthesis) on the mice datasets.
Figure 4.2: Recall of the position pairs belonging to pairs of orthologous (a) and paralogous (b) protein-coding genes by BubbZ (blue), Minimap2 (green) and MashMap2 (red). MashMap2 recall on paralogs was not computed.

Figure 4.3: Accuracy of BubbZ, Minimap2 and MashMap2 on the simulated bacterial datasets. Between genomes. For all tools, both recall and precision decline with increase of the divergence. Recall is similar for all three methods, with BubbZ having slightly better values for more divergent datasets. BubbZ and Minimap2 have nearly identical precision curves, and they are substantially higher than MashMap2.

Table 4.2 shows the running time and memory usage, though because of the small size of the datasets, it is hard to draw any conclusions about scalability in the size or number of genomes. However, this did allow us to measure how the divergence effected each of the methods. For BubbZ and MashMap2 genomic divergence did not have a significant effect on the running time, while Minimap2 ran slower on more divergent genomes.
<table>
<thead>
<tr>
<th>Dataset</th>
<th>TwoPaCo</th>
<th>BubbZ</th>
<th>Total</th>
<th>Minimap2</th>
<th>MashMap2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.03</td>
<td>5 (1,193)</td>
<td>1 (36)</td>
<td>6 (1,193)</td>
<td>6 (904)</td>
<td>3 (147)</td>
</tr>
<tr>
<td>0.06</td>
<td>5 (1,221)</td>
<td>1 (51)</td>
<td>6 (1,221)</td>
<td>8 (820)</td>
<td>3 (154)</td>
</tr>
<tr>
<td>0.09</td>
<td>5 (1,172)</td>
<td>1 (74)</td>
<td>6 (1,172)</td>
<td>10 (824)</td>
<td>3 (168)</td>
</tr>
<tr>
<td>0.11</td>
<td>5 (1,292)</td>
<td>1 (77)</td>
<td>6 (1,292)</td>
<td>10 (634)</td>
<td>3 (172)</td>
</tr>
<tr>
<td>0.14</td>
<td>5 (1,191)</td>
<td>2 (80)</td>
<td>7 (1,191)</td>
<td>15 (1,341)</td>
<td>3 (171)</td>
</tr>
<tr>
<td>0.17</td>
<td>5 (1,262)</td>
<td>1 (80)</td>
<td>6 (1,262)</td>
<td>15 (1,340)</td>
<td>3 (157)</td>
</tr>
<tr>
<td>0.20</td>
<td>5 (1,187)</td>
<td>1 (82)</td>
<td>6 (1,187)</td>
<td>16 (1,113)</td>
<td>3 (165)</td>
</tr>
<tr>
<td>0.22</td>
<td>5 (1,237)</td>
<td>1 (71)</td>
<td>6 (1,237)</td>
<td>16 (614)</td>
<td>4 (164)</td>
</tr>
<tr>
<td>0.25</td>
<td>5 (1,271)</td>
<td>1 (82)</td>
<td>6 (1,271)</td>
<td>16 (1,204)</td>
<td>3 (168)</td>
</tr>
</tbody>
</table>

Table 4.2: Running time (seconds) and memory usage (megabytes, in parenthesis) on the simulated data. Each dataset is labeled by its corresponding divergence.

### 4.4 Discussion

In this paper, we present BubbZ, a novel method for computing pairwise mapping between complete genomes. The algorithm takes the (compacted) de Bruijn graph as input and outputs chains of $k$-mers that are terminal, or cannot be extended anymore. Empirical results indicate that for a large collection of closely-related mammalian genomes, our method can be up to 20 times faster than competing approaches, while maintaining similar accuracy. Our tool is open source and freely available at [https://github.com/medvedevgroup/BubbZ](https://github.com/medvedevgroup/BubbZ).

Our approach for finding chains is based on the problem formulation of the sweep-line algorithms from [90]. Those algorithms similarly defined chains and a dynamic programming formulation to find the longest chains. The main difference is that their algorithm did not make use of the greedy predecessor $\pi$ function that we have, but, rather, explored the space of all possible predecessors in order to find the highest scoring chain. This formulation requires an efficient dynamic range-maximum-query structure [89,90,111] for it to be feasible. Although such data structures have theoretically asymptotic logarithmic query times, in practice they have a high constant due to their implementations relying on search trees with extra information. Our contribution can thus be viewed as a heuristic version of a sweep-line algorithm that sacrifices some accuracy for the sake of lower memory and run time.

One particular limitation of our tool is its high memory usage for large real datasets due to keeping in memory the graph constructed from all input genomes simultaneously. We believe that it should be possible to reduce the memory usage by developing a more efficient memory representation. One possible approach is a succinct data structure for the compacted de Bruijn graph, similar to recently published work [54,116–118]. However, such a representation should contain extra information to permit quick mapping operations required by our algorithm.
Chapter 5 |
Conclusion and future work

Advances in the sequencing technologies and assembly algorithms now allow producing reference-quality complete genomes a lot quicker using fewer resources. This enables comparative analysis of large datasets consisting of fully assembled closely-related genomes of mammalian sizes. However, such analysis is very computationally demanding due to the sheer size of the data and complex organization of genomic sequences. This creates a demand for a set of tools that are capable of analysis of datasets consisting of long genomes using reasonable computer resources.

This dissertation addresses such demand and introduces a suite of programs that allow comparative analysis of such datasets on a single modern server. All of them rely on the compacted de Bruijn graph for indexing the input genomes. First, we introduced TwoPaCo, a software package for constructing the compacted de Bruijn graph for large datasets efficiently. Our software is highly parallelizable and can handle up to 100 human genomes. Then we describe SibeliaZ, a pipeline that uses the graph created by TwoPaCo to produce multiple whole-genome alignment. SibeliaZ is also highly efficient and is able to align 16 mice genomes on a single server in less than a day while having comparable recall of orthologous genes and even having higher recall of paralogous genes of high similarity. The last contribution is BubbZ, another program that uses the compacted graph for comparative analysis. BubbZ is able to find whole-genome homology mapping extremely quickly, being up to 10 faster than competing programs without dropping in accuracy.

We believe that this software suite will democratize the comparative analysis of large datasets consisting of complete genomes. Unlike in the past when such datasets were exclusively generated by large sequencing centers, they are now commonly created by small labs and in higher numbers. By being highly efficient and easy to use, our tools provide an opportunity to perform high-scale comparative analyses for users having access to more modest computational resources. We hope that SibeliaZ will extend the success of its predecessor Sibelia [8], a de Bruijn-graph based synteny finding tool which found multiple uses in biological studies (see for example [119–121]) and was integrated in other software packages [104]. In addition, we believe that TwoPaCo has a great potential in the other areas requiring efficient sequence indexing.
For example, it was adopted in software Pufferfish [118], which used TwoPACo as a backbone for a sequence index.

There are several possible directions for future work. First, there is still room to improve the accuracy of SibeliaZ in duplicated regions with the complex genomic organization. One possible way to do so is to come up with an algorithm for the problem we formulated that has a performance guarantee. Another approach could be using pairwise homology mapping constructed by BubbZ to “guide” the construction of locally-collinear blocks. This direction is especially attractive because BubbZ finds those mappings very quickly and with high sensitivity.

Second, the applicability of our tools could be extended to longer genomic distances. This could be achieved by improving ability of the compacted de Bruijn graph to capture highly-similar substrings. One possible technique that might help is adopting spaced seeds [122] as vertices of the de Bruijn graph instead of exact \( k \)-mers. Another direction is implementing a postprocessing step using more sensitive, but slower tools. This approach is feasible because by constructing locally-collinear blocks using our conventional approach first, we can greatly reduce size of the data. An example of such postprocessing step is clustering similar locally-collinear blocks that were separated in the initial step due to lack of common exact \( k \)-mers. A more engineering direction is to make the implementations more efficient (e.g. by improving parallel scalability) allowing to use computational parameters leading to higher sensitivity.

In addition, both SibeliaZ and BubbZ suffer from relatively high memory consumption on large datasets. Recently researchers in the area of succinct data structures came up with efficient representations of the de Bruijn graph [54, 116–118]. These successes suggest that it should possible to create a more compressed index for the compacted de Bruijn graph that supports the operations required by our applications.
Appendix A
Supplementary information for Chapter 2

A.1 Parameters used for benchmarking

The section contains bash commands we used to run the tools in our experiments. In commands below <k> and <input_file> are the values of k and the input file name correspondingly. For Sibelia, <stage_file> is a text file containing a single 0. For TwoPACo, <threads_number> is the number of worker threads, <bf_size> is the of the Bloom filter, and <hf_number> is the number of hash functions. For bwt-based, <k_file> is a text file containing value(s) of k.

Minia:

```
minia -kmer-size <k>
 -abundance-min 0
 -max-memory 220000
 -traversal unitig -in <input_file>
```

DSK + BCALM:

```
dsk -kmer-size <k>
 -file <input_file>
 -out dsk_out
 -abundance-min 0
 -max-memory 220000
 dsk2ascii -file dsk_out.h5
 -out kmer.txt
 bcalm kmer.txt
```
Sibelia:

Sibelia -k <stage_file>\n  -m <k>\n  --noblocks\n  --graphfile\n  <input_file>

TwoPaCo:

TwoPaCo -k <k>\n  -q <hf_number>\n  -f <bf_size>\n  -t <threads_number>\n  <input_file>

bwt-based from Baier et al:

bwt_based <input_file> graph.out\n  <k_file>

SplitMEM:

splitMEM -file <input_file>\n  -mem <k>
Appendix B
Supplementary information Chapter 3

B.1 Parameter details and command lines

We tried to find the optimal parameters for all tools. For Sibelia, which could only run on simulated data, we used the parameter set designed to yield the highest sensitivity (called the “far” set in Sibelia). Progressive Cactus requires a phylogenetic tree in addition to the input genome which it uses for adjusting the internal parameters. For the simulated datasets, we used the real tree generated by the simulator; for the mice genomes, we used the guide tree from [83]. Multiz+TBA were run with default parameters since its documentation does not provide a clear guideline on how to adjust the parameters according to evolutionary distances between the input sequences. We could not compile the version of MultiZ+TBA publicly available for download and used a slightly modified version provided by Robert S. Harris. For TwoPaCo and spoa, we set the parameters following the guidelines provided with the respective software. SibeliaZ was run with $k = 25$, $b = 200$, $m = 50$, and $a = 150$.

We performed all experiments on a machine running Ubuntu 16.04.3 LTS with 512 GB of RAM and a 64 core CPU Intel Xeon CPU E5-2683 v4. We were limited to using at most 32 threads at any given time. Progressive Cactus was run with 32 threads, since the authors recommended to use as many threads as possible for the best performance. MultiZ+TBA and Sibelia are both single-threaded. (There were several submissions to Alignathon which used an extensively parallelized MultiZ or TBA; unfortunately, the software packages used for those submissions are not available publicly for download.) TwoPaCo and SibeliaZ-LCB were run 16 and 32 threads respectively. We note that spoa is run on each block, and our software includes a wrapper to automate this.

Here are the exact command lines for the tools we ran.

TwoPaCo:
twopaco -k <k_value> -f <bloom_filter_size> -t 16 -o <dbg_graph> <genomes_file>

SibeliaZ-LCB:
SibeliaZ-LCB --fasta <genomes_file> --graph <dbg_graph> -o <output_directory> -k 25 -b 200 -m 50 -a 150 -t 4

spoa:
spoa <input_fasta_file> -l 1 -r 1 -e 8

Sibelia:
Sibelia <genomes_file> -o <output_directory> -s far --lastk 50 -m 50 --nopostprocess

MultiZ:
all_bz <guide_tree>
tba <guide_tree> *.*.maf <outputMafFile>

Progressive Cactus:
runProgressiveCactus.sh --maxThreads 32 <seqFile> <workDir> <outputHalFile>
source ./environment & hal2mafMP.py <outputHalFile> <outputMafFile>

All running times and memory usage numbers were obtained using the GNU time utility. The exact versions of the software (except MultiZ) are in Table B.3.
B.2 Simulation details

We used small simulated data in order to understand the role of a dataset’s genomic distance and of our parameter settings. We used ALF [123] for simulation because it simulates point mutations as well as genome-wide events such as inversions, translocations, fusions/fissions, gene gain/loss, and lateral gene transfer. Furthermore, ALF is useful for benchmarking as it also produces an alignment which represents the true homology between the genomes, making it possible to directly assess the precision and recall. We simulated 6 datasets, each one consisting of 10 bacterial genomes. Each genome is composed of 1500 genes and of size approximately 1.5 Mbp. We used such relatively small datasets to allow us to efficiently explore the parameter space. Each of the 6 datasets corresponded to a different parameter for distance from the root to leaf species, which we varied from 0.03 to 0.1675 substitutions per site with the step of 0.275. In ALF, different proteins evolve with different rates, which are derived from the base value using a probabilistic distribution. See [123] for more details and the simulation recipes for the exact values of the parameters. For genome-wide events, we used ALF’s default rates. Links to download the the simulation parameter files, the simulated genomes, and their alignments are available at the GitHub repository\footnote{\url{https://github.com/medvedevgroup/SibeliaZ/blob/master/DATA.txt}}.
Figure B.1: Properties of the pairwise alignments constructed from pairs of homologous protein-coding genes in the various mice datasets.
Figure B.2: Histogram of the column scores $f(C)$ of SibeliaZ’s and Cactus’ alignment, for datasets 1-2 (a) and 1-4 (b).
Figure B.3: The recall as a function of inferred family size, using the two-mice dataset. The family size is binned into 25 equally sized bins. The top histogram shows the number of gene pairs in each bin.
Figure B.4: Effects of the parameters $k$ and $b$ on recall. Each heatmap corresponds to a simulated dataset with the specified root-to-leaf divergence in substitutions per site and a cell corresponds to a combination of parameters.
Figure B.5: Effects of the parameters $k$ and $b$ on precision. Each heatmap corresponds to a simulated dataset with the specified root-to-leaf divergence in substitutions per site and a cell corresponds to a combination of parameters.
Figure B.6: The accuracy of SibeliaZ as a function of genomic divergence. Each point is labeled with the height of the phylogenetic tree (in terms of substitutions per site) of its respective simulated dataset.
<table>
<thead>
<tr>
<th>ID</th>
<th>Strain</th>
<th>Size (Mb)</th>
<th>N. Scaffolds</th>
<th>Accession number</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>C57BL/6J</td>
<td>2,819</td>
<td>336</td>
<td>GCA_000001635.8</td>
</tr>
<tr>
<td>2</td>
<td>129S1/SvImJ</td>
<td>2,733</td>
<td>7,154</td>
<td>GCA_001624185.1</td>
</tr>
<tr>
<td>3</td>
<td>A/J</td>
<td>2,630</td>
<td>4,688</td>
<td>GCA_001624215.1</td>
</tr>
<tr>
<td>4</td>
<td>AKR/J</td>
<td>2,713</td>
<td>5,953</td>
<td>GCA_001624295.1</td>
</tr>
<tr>
<td>5</td>
<td>CAST/EiJ</td>
<td>2,654</td>
<td>2,977</td>
<td>GCA_001624445.1</td>
</tr>
<tr>
<td>6</td>
<td>CBA/J</td>
<td>2,922</td>
<td>5,466</td>
<td>GCA_001624475.1</td>
</tr>
<tr>
<td>7</td>
<td>DBA/2J</td>
<td>2,606</td>
<td>4,105</td>
<td>GCA_001624505.1</td>
</tr>
<tr>
<td>8</td>
<td>FVB/NJ</td>
<td>2,589</td>
<td>5,013</td>
<td>GCA_001624535.1</td>
</tr>
<tr>
<td>9</td>
<td>NOD/ShiLtJ</td>
<td>2,982</td>
<td>5,544</td>
<td>GCA_001624675.1</td>
</tr>
<tr>
<td>10</td>
<td>NZO/HiLtJ</td>
<td>2,699</td>
<td>7,022</td>
<td>GCA_001624745.1</td>
</tr>
<tr>
<td>11</td>
<td>PWK/PhJ</td>
<td>2,560</td>
<td>3,140</td>
<td>GCA_001624775.1</td>
</tr>
<tr>
<td>12</td>
<td>WSB/EiJ</td>
<td>2,690</td>
<td>2,239</td>
<td>GCA_001624835.1</td>
</tr>
<tr>
<td>13</td>
<td>BALB/cJ</td>
<td>2,627</td>
<td>3,825</td>
<td>GCA_001632525.1</td>
</tr>
<tr>
<td>14</td>
<td>C57BL/6NJ</td>
<td>2,807</td>
<td>3,894</td>
<td>GCA_001632555.1</td>
</tr>
<tr>
<td>15</td>
<td>C3H/HeJ</td>
<td>2,701</td>
<td>4,069</td>
<td>GCA_001632575.1</td>
</tr>
<tr>
<td>16</td>
<td>LP/J</td>
<td>2,731</td>
<td>3,499</td>
<td>GCA_001632615.1</td>
</tr>
</tbody>
</table>

Table B.1: Properties of the assembled mice genomes available at GenBank.
<table>
<thead>
<tr>
<th>Dataset</th>
<th>TwoPaCo</th>
<th>SibeliaZ-LCB</th>
<th>spoa</th>
<th>Total</th>
<th>Cactus</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-2</td>
<td>12 (9.30)</td>
<td>74 (36.00)</td>
<td>68 (121.50)</td>
<td>154 (121.50)</td>
<td>2,279 (37.50)</td>
</tr>
<tr>
<td>1-4</td>
<td>25 (17.70)</td>
<td>96 (72.70)</td>
<td>115 (133.60)</td>
<td>236 (133.60)</td>
<td>6,105 (89.80)</td>
</tr>
<tr>
<td>1-8</td>
<td>49 (34.50)</td>
<td>104 (106.30)</td>
<td>240 (132.50)</td>
<td>393 (132.50)</td>
<td>-</td>
</tr>
<tr>
<td>1-16</td>
<td>101 (68.10)</td>
<td>153 (183.60)</td>
<td>736 (133.50)</td>
<td>990 (183.60)</td>
<td>-</td>
</tr>
</tbody>
</table>

Table B.2: Running time (minutes) and memory usage (gigabytes, in parenthesis) of SibeliaZ and Cactus on the mice datasets. A dash in a column indicates that the program did not complete within in a week.
<table>
<thead>
<tr>
<th>Software</th>
<th>Repository</th>
<th>Commit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sibelia</td>
<td>bioinf/Sibelia</td>
<td>397e6877116006c8591cbe14a7c6d366d1e0751a</td>
</tr>
<tr>
<td>SibeliaZ</td>
<td>medvedevgroup/SibeliaZ</td>
<td>e90f5b25c931b5b011b98c558670f1697334ef69</td>
</tr>
<tr>
<td>TwoPaCo</td>
<td>medvedevgroup/TwoPaCo</td>
<td>9b9fee321dd561b7bd2b18892b0b2653c58eb6dd</td>
</tr>
<tr>
<td>spoa</td>
<td>rvaser/spoa</td>
<td>4c87d6831e9898dcaf2830182afece85e77b09ce</td>
</tr>
<tr>
<td>Progressive Cactus</td>
<td>glennhickey/progressiveCactus</td>
<td>e4bed56c0cd48d23411038acb9c19bcae054837e</td>
</tr>
</tbody>
</table>

Table B.3: GitHub revisions of the software we used. We used the most up-to-date versions available at the time of development of our project.
Bibliography


[95] PERRY, E. (2018), Personal communication.


Vita

Ilia Minkin

Ilia Minkin is a Ph.D. candidate in the Department of Computer Science in the Pennsylvania State University. Before joining the program in 2014, he received a Specialist degree in Computer Engineering from Siberian Federal University, Krasnoyarsk, Russia in 2011. He then got an M.Sc degree in Bioinformatics from University of Russian Academy of Sciences in Saint-Petersburg, Russia in 2013, where he was supervised by Son Pham and Nikolay Vyahhi. After completing the M.Sc degree, he worked in the University of Hong Kong for six months as a research assistant under supervision of professor Francis Chin.