Contributions to Computing and Modeling Multiple Whole-Genome Alignments

DISSERTATION

zur Erlangung des Grades

eines Doktors der Naturwissenschaften (Dr. rer. nat.)

am Fachbereich für Mathematik und Informatik

der Freien Universität Berlin

vorgelegt von Birte Kehr

Betreuer: Prof. Dr. Knut Reinert

Berlin 2013

Erstgutachter:	Prof. Dr. Knut Reinert
Zweitgutachter:	Prof. Aaron E. Darling, PhD

Tag der Disputation: Montag, 19. Mai 2014

Abstract

Recent advances in sequencing technologies have opened up the opportunity to study whole genomes at the nucleotide level. Similarities in the nucleotide sequences of genomes provide new insights in the relationships of organisms and species. Multiple whole-genome alignments represent these similarities, however their computation is challenging. In contrast to approaches for other sequence alignment problems, genome alignment methods have to deal with very long sequences and with non-colinearity of similarities.

This thesis makes three contributions to the development of multiple whole-genome alignment methods. The prevailing strategy of such methods is to combine a set of local alignments to a global genome alignment. This thesis suggests an efficient and fully sensitive local alignment approach, compares graph data structures for representing genome alignments, and describes hidden rearrangement breakpoints that become visible only in the comparison of more than two genomes.

All three contributions provide potential for significant improvements to the computation or modeling of genome alignments. In a comparison to other local alignment approaches, the new local aligner is the fastest of three fully sensitive ones and competitive with seed-and-extend approaches despite having full sensitivity. The assessment of graph data structures describes for the first time all graphs using the same terminology, and demonstrates how the graph structures differ in their information content. Finally, an analysis of breakpoints in simulated genome alignments suggests that hidden breakpoints are abundant and relevant for measuring the accuracy of genome alignments. In summary, the three contributions provide a promising basis for future genome alignment methods.

Zusammenfassung

In den letzten Jahren wurden enorme Fortschritte mit Sequenziertechnologien gemacht, die es ermöglichen, ganze Genome auf Nukleotidebene zu untersuchen. Genomische Nukleotidsequenzen weisen erstaunliche Ähnlichkeiten auf, die neue Einblicke in Verwandschaftsbeziehungen von Organismen und Arten gewähren können.

Multiple Genomalignments stellen diese Ähnlichkeiten zwischen verschiedenen Genomen dar, ihre Berechnung ist allerdings sehr anspruchsvoll. Im Gegensatz zu Methoden für andere Sequenzalignmentprobleme muss bei der Berechnung von Genomalignments zum einen die gewaltige Länge von genomischen Sequenzen bewältigt werden und zum anderen beachtet werden, dass ähnliche Abschnitte in verschiedenen Genomen in unterschiedlicher Anordnung vorliegen können (Nichtkolinearität).

Methoden zur Berechnung von Genomalignments suchen meist zunächst lokale Alignments und fügen diese anschließend zu einem globalen Genomalignment zusammen. Die vorliegende Arbeit leistet drei Beiträge zu solchen Methoden: Es wird ein effizienter Ansatz für lokales Alignment vorgeschlagen, Graphdatenstrukturen für Genomalignments verglichen und genomische Bruchstellen beschrieben, die erst im Vergleich von mehr als zwei Genomen sichtbar werden und auf zusätzliche nichtkolineare Veränderungen hinweisen.

Der erste Beitrag, ein Ansatz um effizient lokale Alignments zu berechnen, hebt sich durch eine Garantie, dass alle lokalen Alignments gefunden werden, von anderen effizienten Ansätzen ab. Diese Garantie wird sowohl theoretisch bewiesen als auch in Tests auf simulierten und echten Daten verdeutlicht. Ein Vergleich mit gängigen Programmen für lokales Alignment bestätigt, dass der Ansatz trotz Sensitivitätsgarantie ähnlich schnell und damit eine geeignete Alternative für den Einsatz in Methoden zur Berechnung von Genomalignments ist.

Der zweite Beitrag ist ein Vergleich von vier Graphdatenstrukturen, die Genomalignments repräsentieren können. Erstmals werden die Graphstrukturen unter Verwendung eines einheitlichen Fachvokabulars beschrieben, welches verdeutlicht, wo die Graphstrukturen Unterschiede im Informationsgehalt aufweisen. Alle untersuchten Strukturen können Nichtkolinearität nicht vollständig darstellen – Kanten- bzw. Knotenbeschriftungen sind hierfür in allen Graphen notwendig. Dennoch sind die Graphen auch konzeptionell ein wichtiger Bestandteil von Methoden

zur Berechnung von Genomalignments.

Der dritte Beitrag beschreibt schließlich genomische Bruchstellen, die erst im Vergleich von mehr als zwei Genomen sichtbar werden. Diese versteckten Bruchstellen sind auf evolutionären Verlust von Sequenzmaterial zurückzuführen oder erscheinen bei geringer Alignmentauflösung. Es wird eine Zählmethode für versteckte Bruchstellen vorgestellt, die auf ein Standardgraphenproblem der Informatik zurückgreift: gewichtetes, perfektes Matching. Eine Analyse von Bruchstellen in simulierten Genomalignments zeigt, dass versteckte Bruchstellen relevant sind, um die Genauigkeit und Güte von Genomalignments zu messen.

Alle drei Beiträge konzentrieren sich auf eine der zwei spezifischen Eigenschaften von Genomalignments: auf die Länge oder die Nichtkolinearität. Die Berechnung ganzer Genomalignments bleibt allerdings weiterhin eine anspruchsvolle Aufgabe und bietet zahlreiche Möglichkeiten für weitere Forschungsarbeiten. Die drei Beiträge dieser Arbeit bieten jedoch bereits eine vielversprechende Grundlage für ein zukünftiges Programm zur Berechnung multipler Genomalignments.

Acknowledgments

I wish to thank all the people who supported me during my time as a PhD student. First and foremost, I am grateful to my advisor, Knut Reinert, who proposed the topic of genome alignment for this thesis, a topic I would choose again. I very much appreciate that he was always available to give advice the many times I knocked on his door – even when he was extremely busy. The enthusiasm he has for his work has been a great example and his encouragement was vital during the last years.

I am especially thankful for getting the chance to work with Aaron Darling during a three-months stay at the UC Davis Genome Center. He taught me a lot in uncountable discussions about genome alignments. In addition, he proved to be an infinite source of ideas and I learned from him how much fun it is to do research. He was in large parts responsible for making my stay in California as enjoyable and successful as it was. I thank Jonathan Eisen for letting me stay in his lab and the International Max Planck Research School (IMPRS) and Dahlem Research School for making it possible.

I am grateful to Martin Vingron and Peter Arndt for having been my IMPRS mentors and for giving helpful comments and valuable advice in PhD seminars. Many thanks go to my co-authors David Weese, Kathrin Trappe, and Manuel Holtgrewe, to all past and present members of the `SeqAn` team, and to Falk Hüffner for his algorithmic support. Further, I wish to thank Jens Stoye for his motivating feedback on `STELLAR` and hidden breakpoints at RECOMB-CG and WABI.

I would like to thank all group members from AG-ABI in Berlin and the Eisen lab in Davis as well as all fellow IMPRS students for the good times we had and for fun group activities. I am particularly grateful to Anne-Katrin Emde as office mate and the first person for all research-related and research-unrelated questions. Thanks to Kirsten Kelleher and Hannes Luz for their excellent coordination, for organizing great IMPRS retreats, and for diverse last Wednesday events.

I am indebted to Anne-Katrin, Christina, René, Kathrin, Sharon, Judith, and Ryan for proofreading the many parts of this thesis and, last but not least, to my parents for their steady support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Millions of different species populate the earth [110] including humans and mice, birds and fish, insects, plants, fungi, and not least bacteria and archaea. Underlying this immense diversity of life is the process of self-replication. Every living organism has the ability to reproduce and passes its genetic information on to its offspring. Over billions of years, changes to the genetic material created the variety of species alive today.

Remarkable similarity in biochemical mechanisms across all domains of life suggest that all present-day species derived from a single origin [24]. Likewise, this similarity can be found in genetic information. We assume that all diversity of life was created from this single origin through a process of change. The marks of this process are similarities in the genetic material of extant species. This thesis contributes to computational methods for finding such similarities on the nucleotide level of genomes.

Genomes, chromosomes, and DNA

All of an organism's genetic information is encoded in its *genome*. A copy of the genome is present in each cell of every organism. Genomes define the shape of organisms, regulate the mechanisms that operate living cells, and are passed on to offspring. Common traits of different organisms can mostly be attributed to common features in their genomes.

The vast amount of information in genomes requires a high degree of organization. The largest units of organization in genomes are *chromosomes*. Each chromosome consists of a single DNA double helix that is packaged with proteins. The DNA encodes genetic information and the proteins regulate accessibility to the DNA. In some species, the complete genetic information of that species is encoded in a single chromosome, and in other species it is spread over several chromosomes (see Fig. 1.1).

The *karyotype* of a species describes the organization of its genome on the chromosome level. It includes the genome size and the shape and number of chromosomes. The sizes vary greatly both within and among different clades of species (see Fig. 1.2). The shape of a chromosome is either *linear* or *circular*. In linear chromosomes, the two ends are protected by repetitive elements called *telomeres*. The number of chromosomes is both defined by the *ploidy*, the number of sets of different chromosomes, as well as by the number of chromosomes per set.

The karyotype can be a first indicator of the similarity of two species. However, a much larger resource for studying the relationships between organisms is the DNA molecules themselves. DNA, *desoxyribonucleic acids*, are linear macromolecules formed by four basic building blocks, the *nucleotides* adenine (A), cytosine (C), guanine (G), and thymine (T). The sequence of nucleotides in DNA molecules encodes the genetic information.

In each chromosome, two strands of DNA form a double helix. In the center of the double helix, base pairings of the nucleotides keep the two strands together, where A always pairs with T, and C always pairs with G. Base pairs (bp) are a common unit of length for DNA sequences. The two strands in the helix are antiparallel, and thus run in opposite chemical directions. Owing to this structure, the two strands are said to be *reverse complements*.

The discovery of the double helix structure of DNA in 1953 received a lot of attention because it led directly to the replication mechanism of genetic material [162]. For replication, the two strands are separated and complemented with new nucleotides using the base pairings of A-T and C-G. The result is two copies of the same double stranded DNA helix. In the cell replication process, one copy is passed on to each of two daughter cells. This explains the high degree of similarity between the genomes of parents and offspring.

Mutation, evolution, and homology

The genomes of related organisms typically have a high degree of similarity but are not identical. *Mutation events* alter the DNA sequence and introduce differences at the scale of a single or a few nucleotides, of larger consecutive regions of DNA, or of whole chromosomes.

Errors in the replication process can lead to the insertion, deletion, or substitution of single nucleotides. In addition, exposure to chemicals and light can alter the sequence at the scale of nucleotides (see Fig. 1.3). Recombination through breakage and rejoining of DNA strands can affect the sequence of DNA at a larger scale. Another common mechanism of change, especially in bacteria, is horizontal gene transfer, where a functional region of DNA from another species is transferred to an organism's genome. Eukaryotic cells contain transposable elements that duplicate and change their location and orientation in the genome over time. At the chromosome scale, fusion or fission and duplication or loss can occur.

Although mutation of DNA has been well-studied, we may assume that there are mechanisms of change that have not been discovered until today. Only recently, the mechanism of *chromothripsis* was described [152] where massive genome rearrangement occurs through shattering of whole chromosomes. Similarly, other mechanisms might be discovered in the future.

Changes in the genomes can have an impact on biochemical processes and consequently on the

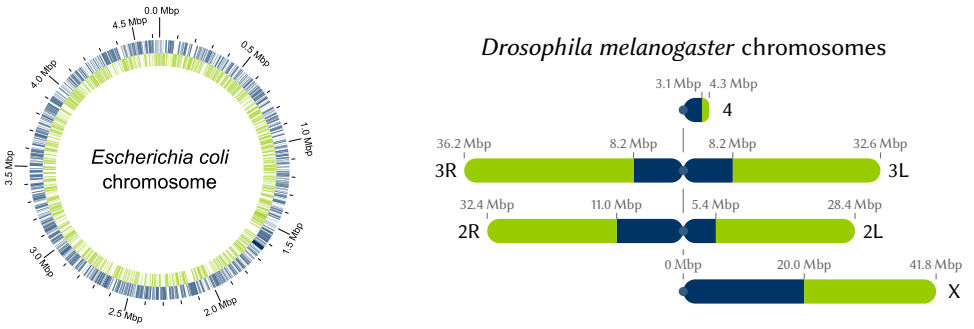

Figure 1.1: The circular chromosome of *E. coli* (left) and the linear chromosome set of *D. melanogaster* (right). In the *E. coli* genome, green bars denote genes on the forward strand and dark blue bars denote genes on the reverse strand. Gene positions were downloaded from the PEC database [167]. In the *D. melanogaster* genome, green denotes euchromatin and dark blue denotes heterochromatin. Chromosome sizes were taken from Adams *et al.* [2].

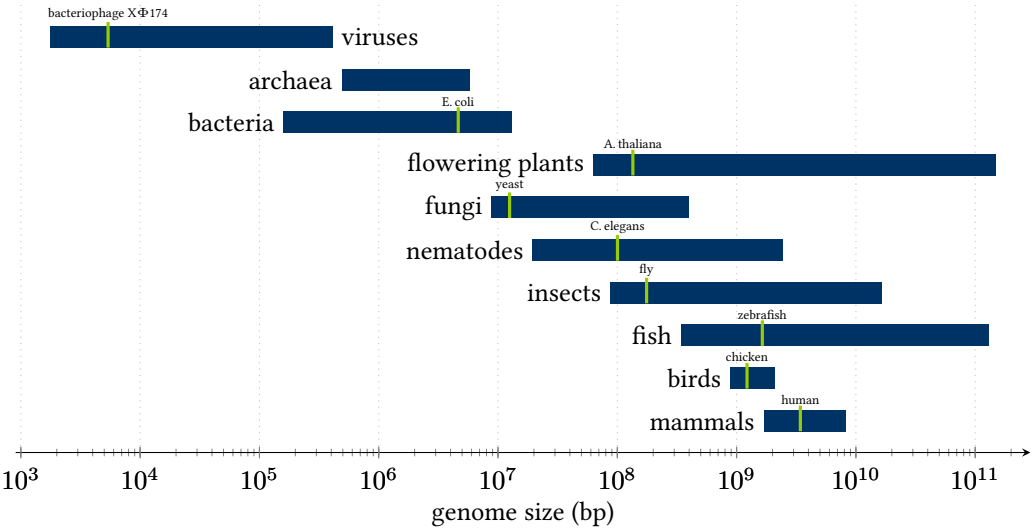

Figure 1.2: Ranges of known genome sizes for different clades of species. The numbers are given on a logarithmic scale and were collected from multiple sources [5, 15, 66, 67, 68, 79].

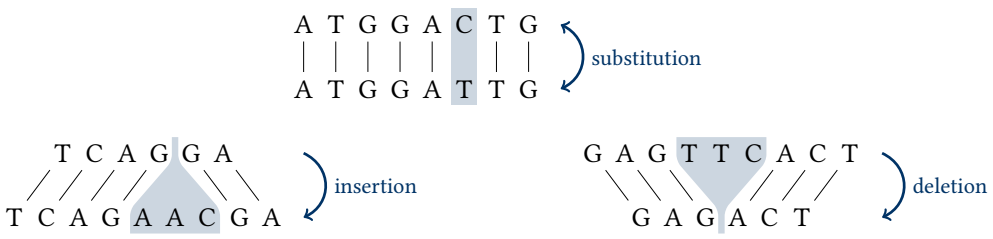

Figure 1.3: A nucleotide substitution, and an insertion and deletion of three consecutive nucleotides. Note that insertions and deletions are directed mutations whereas substitutions are mutual.

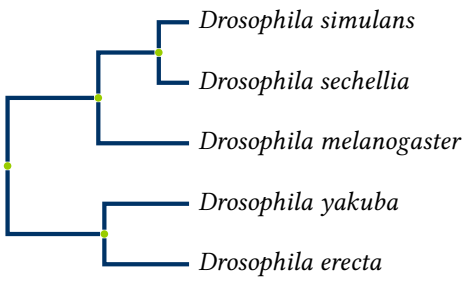

Figure 1.4: The phylogenetic tree of five drosophila species from the melanogaster subgroup. The topology was drawn guided by the phylogeny of sequenced species from FlyBase [157]. Inner nodes (green) represent speciation events.

shape of an organism. As long as these changes do not affect reproducibility, new traits in unicellular organisms and germ line cells of multicellular organisms are passed on to the following generations. Eventually, *evolution* of genomes, the process of change over time, can lead to *speciation*, the birth of new species.

Closely related species still share a considerable amount of genomic sequence [154], and even between distantly related species, parts of the genomes remain highly conserved over time. Regions of DNA that originate from the same common ancestor are called *homologous*. Depending on the relationship of homologous sequences, they can be further classified: Homologous sequences that are related through duplication events within the same genome are *paralogous*, and homologous sequences that are related through speciation are *orthologous*. For more details on homology classification see [44].

We can display evolutionary relationships using a *phylogenetic tree* (see Fig. 1.4). The tree structure represents the relationships of species and organisms to each other. In addition, a phylogenetic tree makes assumptions on common ancestors of species by clustering subsets of species. Thus, leaf nodes represent extant species and inner nodes of the tree represent ancestral species. Branch lengths of the tree represent the number of changes between genomes, and hence indicate evolutionary distances.

Evolutionary biology is the discipline that studies the relationships of species and the processes that create diversity. On the genomic level, the ultimate goal is to reconstruct the evolutionary history of each and every nucleotide. In order to learn about past changes, research has to resort to studying similarities among extant species (with few exceptions [65, 108]). Therefore, the comparison of genome sequences plays a key role for gaining new insights into the history of life.

Genome sequencing

A requirement for all analyses of genome sequences at the nucleotide resolution is *DNA sequencing*, the ability to transcode the sequence of nucleotides in a DNA molecule to a format readable

by modern computers.

In the 1970s, Frederick Sanger established the first practical techniques for DNA sequencing [143] and sequenced the first entire genome, that of bacteriophage Φ X174 [142]. Together with the invention of PCR in the 1980s [141], Sanger's sequencing techniques laid the basis for larger-scale genome sequencing initiatives. The international human genome project was launched in 1990 [161] with the goal of determining all three billion base pairs of the human genome. Yet it took until 1995 to sequence the first entire genome of a free living organism, *Haemophilus influenzae* [59]. This 1.8 Mbp genome was followed by many (small) genomes.

In 2001, four years ahead of the original time schedule, a draft of the human genome was announced by two independent groups, by the publicly-funded human genome project consortium [96] and by its private competitor Celera Genomics [158]. Since then, more and more genomes have been sequenced – of different species, but also of different organisms of the same species [1]. As of February 2012, the NCBI website counts 3327 genome sequencing projects with roughly a third completed [115].

The enormous upscaling of sequencing projects was facilitated by rapid developments in sequencing technologies, which brought about a steady fall in sequencing costs [29, 30]. All technologies can only sequence short fragments of DNA called *reads*, which subsequently need to be assembled to whole genomes. While Sanger sequencing has achieved reliable and relatively long reads of several hundreds of base pairs, it is too expensive and laborious for genome-scale sequencing. Less expensive methods that produce only very short reads emerged as soon as computing power became available. Computers allow for the automated assembly of short reads into longer fragments. The advancements that followed were made towards high-throughput sequencing by automation and parallelization. Although the technologies for this *next generation* of *sequencing* (NGS) started off with very short reads (a few dozens base pairs), the length has been gradually increasing over the years, making the assembly process more robust to errors. The newest technologies at the time of this writing promise average read lengths of 8,500 base pairs [120].

Due to this revolution in sequencing technologies, a huge amount of genome sequencing data has become available. The technologies offer new opportunities to study function and relationships of genomes. At the same time, the new kind of data poses new challenges for analysis techniques. The bottleneck in genome research is shifting from sequencing ability towards analysis of sequencing data.

For now, most analyses (ranging from variant detection within the same species to genome annotation of newly sequenced species) are still being carried out on primary read data and not on assembled genomes. This may have several explanations: First of all, accurate assembly of short reads to whole genomes requires large amounts of sequencing data and large computational resources, and thus is still very expensive. In addition, analyses on primary data may be less error-prone since an assembly step can introduce errors. Finally, the majority of available methods and analysis tools are designed for analyzing read data and not yet whole genomes.

Considering the development of sequencing technologies, and in particular the increasing read lengths, we can expect genome assemblies to further improve in the near future; if in the future

entire genomes can be sequenced in one read, assembly might become unnecessary. It is to be expected that whole-genome analysis methods will gain much more importance, allowing exploration of genomes from an integrated, holistic point of view.

Sequence alignments

For the analysis of both short reads and fully assembled genomes, comparative approaches play an important role. Through comparison we can identify similarities, predict homologies, and infer evolutionary changes. This is the basis for phylogenetic studies, which infer evolutionary relationships among species [42, 139]. Under the assumption that similar genomic sequences have similar functions, it is also possible to predict some functional regions of a newly sequenced genome, given similarities with a well-studied and annotated genome [8, 106, 138].

The most widely used approach for the comparison of genomic sequences is *alignment*. An alignment places linear sequences below each other in a line, one sequence per row. The alignment process adds *gap* symbols to the sequences at positions where other sequences have extra nucleotides. As a result, all nucleotides that are assumed to be homologous appear in one column. Such a representation highlights the differences between the sequences and allows the probability of homology to be estimated.

A variety of different alignment problems and alignment types exists. The number of given sequences, the fraction of the sequences that is expected to be similar, and the order of multiple similarities in the sequences influence the formulation of an alignment problem as described below. Furthermore, the length of the sequences plays a role when choosing an appropriate alignment method.

The number of sequences to be compared has a direct influence on the complexity of the problem. A *pairwise alignment* of two sequences is much easier to compute than a *multiple alignment* of several sequences [159]. But already in the comparison of two sequences, it is difficult to decide which nucleotides are homologous at locations where several mutations have changed the sequences (see Fig. 1.5). Here, additional sequences can often clarify the correct assignment of gaps. Thus, multiple alignments are usually more accurate in predicting homology [140] although they are more difficult to compute.

The exact fraction of the sequences that is similar is usually unknown, but a general expectation about global or local similarity is commonly given. Sequences that are similar over their full length can be aligned *globally*. For sequences that only have similar regions but are not similar over their full length, a *local alignment* is more appropriate. To compute assemblies of genomes, *overlap alignments* of reads are necessary, where the left end of one read aligns with the right end of another read. Read mapping is the computation of *semi-global alignments* of short reads to a reference sequence. Here, the short reads align globally with a local region of the long reference sequence.

Depending on whether the evolutionary changes affected sequence order and orientation, the aligned sequences are *colinear* or *non-colinear* with respect to each other. In the presence of only substitutions, deletions, and insertions of nucleotides (see Fig. 1.3) including gain and loss of

Figure 1.5: Two alignments of the same sequences demonstrating the problem of gap placement in alignments. The left alignment implies a substitution and an insertion/deletion, whereas the right alignment implies multiple insertion/deletions.

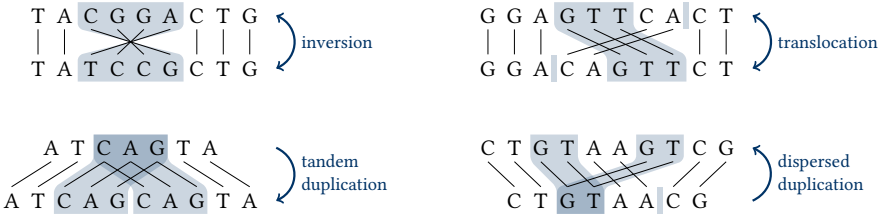

Figure 1.6: Non-colinear changes between DNA sequences. An inversion (top left) reverse complements a segment of DNA. A translocation (top right) changes the position of a segment, and duplications insert additional copies of a segment of DNA (bottom).

Figure 1.7: Two alternative alignments of the same sequences that evolved through duplications. The left alignment represents the conservation of an 11 bp long segment after speciation of the two sequences. The right alignment represents the duplication of the triplet CGT prior to speciation and cuts the longer conserved region into smaller segments.

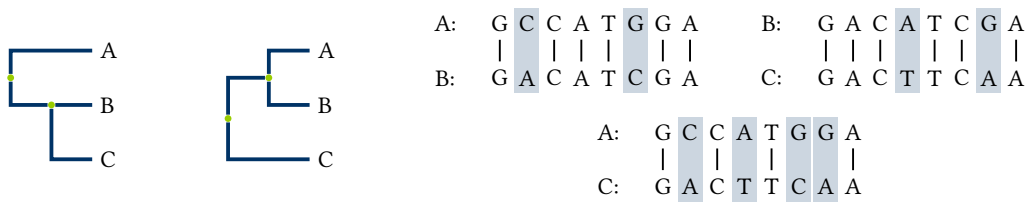

Figure 1.8: Transitivity of the homology relation and its difference to sequence similarity. Given three genomic regions A, B, and C, let A and B be homologous and let B and C be homologous. Then, A and B share a common ancestor and B and C share a common ancestor. The divergence of A and B happened either before or after the divergence of B and C. If it happened before (left tree), then the ancestor of A and B is also an ancestor of C. If it happened after (right tree), then the ancestor of B and C is also an ancestor of A. Thus, A and C have a common ancestor and are homologous, which proves transitivity.
Let A and B be similar with 75 % identity and let B and C be similar with 75 % identity (top alignments). Without assumptions on relationships of the sequences, A and C can have an identity as low as 50 %, which is hardly recognizable as similarity (bottom alignment). Thus, sequence similarity is not transitive.

longer DNA segments, a *colinear alignment* can capture all similarities. We also call these types of changes *colinear changes*, as opposed to *non-colinear changes* that include rearrangements and duplications. We classify non-colinear changes as inversions, translocations, or duplications (see Fig. 1.6). In the presence of non-colinear changes, a colinear alignment will not identify all homologies. Therefore, *non-colinear alignments* are sought for the comparison of whole genomes (see below).

Duplications are special among non-colinear changes as they create self-similarity within one sequence. They add another dimension to non-colinear alignments. In the presence of duplications, segments from one genome are similar to several segments in another genome. A conventional non-colinear alignment chooses between representing either the similarity to all copies or to only the positionally conserved copy. These two options capture similarities at different resolutions and thereby at different evolutionary layers (see Fig. 1.7 and [44, 45]). To capture all layers, a nested or hierarchical representation of similarity is necessary [91, 109, 122].

Underlying the prediction of homology based on alignments are two assumptions: alignments reveal similarity and similarity indicates homology. However, an alignment is not the same as similarity and similarity is not homology. Given similar sequences, there is an alignment that displays the similarity, but there are always many other alignments that do not display the similarity or display a lower degree of similarity. Furthermore, given a degree of similarity, it is possible to calculate a probability for homology, but even for identical sequences there is a small probability that they are identical by chance.

The task is to find the alignment that displays the highest degree of similarity for the input sequences. Most computational approaches formalize similarity in a quality measure, for example an *alignment score*. The quality measure then indicates the probability that an alignment represents homology. Consequently, the alignment problem becomes an optimization problem.

Although similarity is widely used to model homology, the two relations have different properties. Biologically, similarity only formally describes sequences whereas homology implies an evolutionary history. Mathematically, most similarity definitions are not transitive but the homology relation is transitive (see Fig. 1.8). This becomes relevant in alignment methods, for example when combining pairwise alignments into multiple alignments.

Multiple whole-genome alignments

The term *whole-genome alignment* refers to a global alignment of whole genomes. In order to identify all significant similarities between the compared genomes, genome alignments generally account for non-colinear changes. While methods for computing both pairwise and multiple genome alignments exist, this thesis focuses on multiple alignments. In the following, *genome alignment* refers to multiple global non-colinear alignment of whole genomes.

The goal of genome alignments is to provide a map of homology over the full length of several genomes. Since a genome alignment aims at revealing the similar regions no matter in what order, orientation, and copy number they appear, a genome alignment is essentially a set of local colinear multiple alignments. In the case of duplications, local alignments have rows for several

segments of one genome.

Not every set of local alignments is a genome alignment. In an arbitrary set, local alignments are independent from each other and can overlap. However, genome alignments should conform with transitivity of the homology relation, and thus each nucleotide of every genome should only be part of at most one local alignment that aligns possibly multiple genomic regions.

As a consequence, local alignments impose a *segmentation* on the genomes, where each segment of a genome corresponds to a row of a local alignment. The positions between segments are called *breakpoints* if they indicate a non-colinear change. The number of breakpoints is an indicator for the evolutionary distance besides sequence similarity. The order and orientation (rearrangement) of segments in genome alignments are the basis for further studies of evolutionary history [57].

Since genome alignments integrate as much sequence similarity information as is available, they promise a more accurate prediction of homology than local, pairwise, and colinear alignments also for parts of the genomes. A single local alignment captures only similarities of sequence regions and does not account for potential surrounding similarities and alternative alignments; a pairwise alignment is less accurate than a multiple alignment in the presence of several changes at the same location (see Fig. 1.5 above); and a colinear alignment can only identify those similarities that occur in conserved order and orientation.

Current methods for computing genome alignments formalize the problem in varying forms, so that a general formal definition is difficult to give. Nevertheless, they share a general overall strategy. All of them compute an initial set of local alignments and process this set to eventually become a genome alignment. Among these methods are `ABA` [133], `Cactus` [122], `DRIMM-Synteny` [132], `Enredo` and `Pecan` [124], `Mauve` [35] and `progressiveMauve` [37], `Mercator` and `MAVID` [23, 43], `Mugsy` [9], `Sibelia` [109], `SuperMap` [49], and `TBA` [20].

The strategy of combining local alignments to a global alignment is not new but has been used in many efficient methods for computing colinear alignments, for example `DIALIGN` [111, 112], `MGA` [78], `LAGAN` [25], `SeqAn::T-Coffee` [135], and `FSA` [22]. A substantial difference of genome alignments to colinear alignments is that there are no mutually exclusive choices between local alignments. In colinear alignments, some combinations of local alignments cause conflicts with colinearity. However, genome alignments do not require colinearity and consequently any set of local alignments can be refined to a valid solution. The challenge for methods that compute genome alignments is to identify the set of local alignments that has the highest probability of representing homology.

Computing and modeling genome alignments

The approaches for computing genome alignments are generally modular. All methods begin with a set of local alignments. Next, they process the local alignments with varying algorithms. Often graph data structures support the processing algorithms. Finally, finishing steps can significantly contribute to the accuracy of genome alignments. The following provides more details on local alignment detection, representation in data structures, processing the set of local alignments, and finishing the genome alignment. Figure 1.9 displays an example illustrating many of

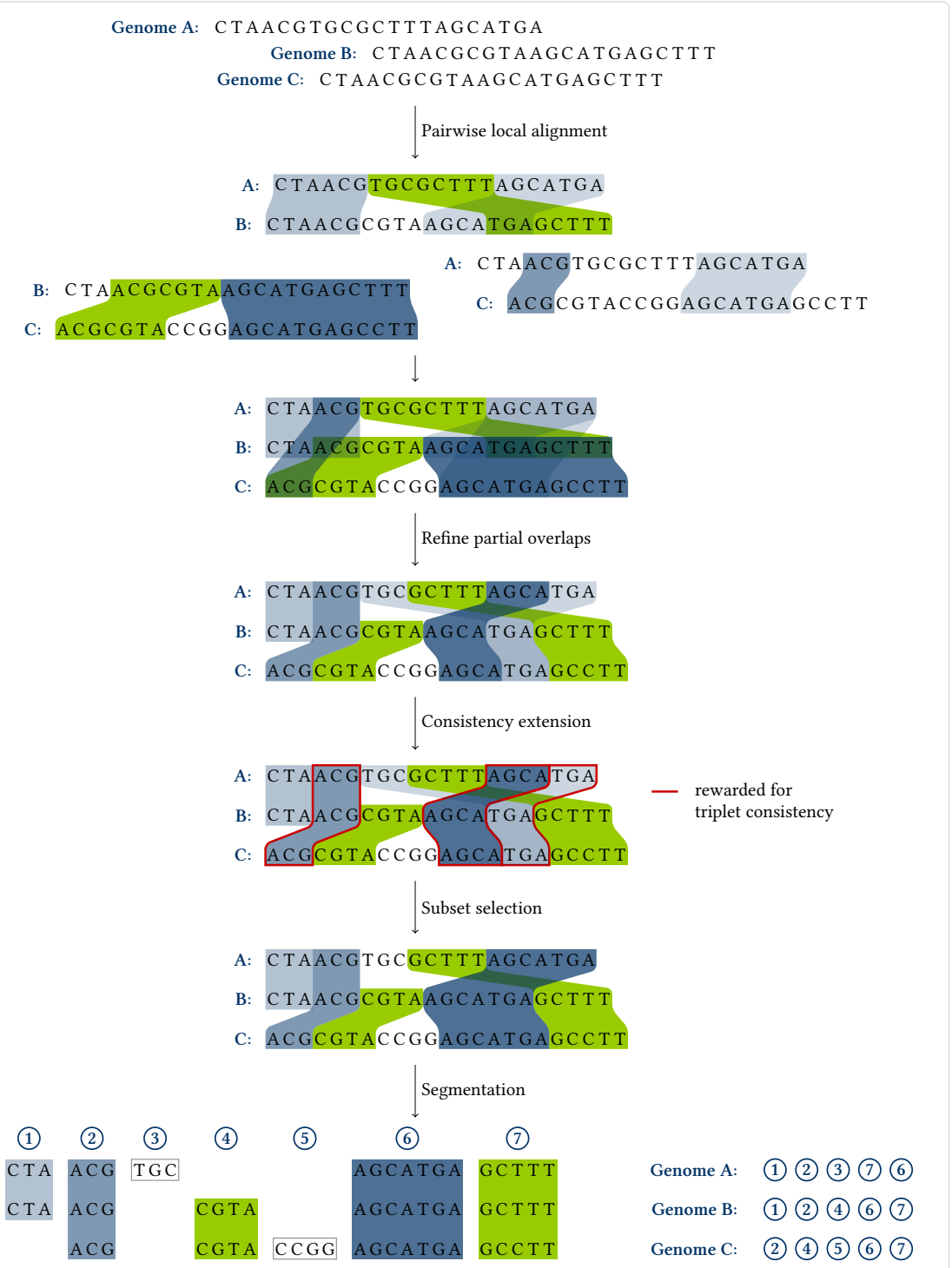

Figure 1.9: An example of common steps computing genome alignments. Here, local alignments of all possible pairs of genomes are combined to a genome alignment through segment match refinement, consistency extension and selection of a subset of local alignments. The processing steps convert the initial seven local pairwise alignments into three three-way alignments, two pairwise alignments and two unaligned segments giving five segments for each input genome. For simplicity, the example contains no inversions (all segments are in the same orientation).

the steps that lead to a genome alignment.

Local alignment detection. The computation of a set of local alignments massively narrows down the enormous search space for genome alignments. This step identifies potential homologies and implicitly marks a large portion of the search space as non-homologous. The set of local alignments determines the global map of similarity and provides a basis for the subsequent steps.

Nevertheless, the set of local alignments may include spurious similarity and it also may miss regions of diverged homologies. The local alignments serve only as anchors for the subsequent steps, where pairwise alignments can be combined to multiple alignments, consecutive local alignments can be chained, and short local alignments can be extended depending on the context. Furthermore, spurious similarities can be identified and removed from the set of local alignments given surrounding similarities.

Similarities that are neither in the set of local alignments nor near a local alignment from the set will not be part of the final genome alignment. In this initial step it is less important that all local alignments represent true homology than that all similarities are included. Spurious local alignments can be removed but similarities of rearranged regions will not be added in later steps.

Methods for computing genome alignment benefit from previous extensive research on local alignments. Some genome alignment methods apply an external local alignment tool (`Cactus` [121] uses `LASTZ` [76] to name just one example) and others use a built-in approach for computing local alignments. External tools have the advantage of being easily interchangeable, while built-in approaches can be more tightly integrated and adapted for the genome alignment purpose.

Most local alignment approaches compute pairwise local alignments. Methods for computing genome alignments that apply these pairwise approaches (for example `Mugsy` [9] that uses `NUCmer` [41], or `SuperMap` [49] that uses `CHAOS` [25]) typically compute pairwise local alignments for all pairs of genomes (see Fig. 1.9) and later combine pairwise alignments to multiple alignments. An alternative is to immediately compute multiple alignments: The genome aligner `Mauve` [35] identifies local similarities shared among all input genomes and its successor `progressiveMauve` [37] additionally identifies local similarities shared by subsets of the input genomes.

Given the length of genomes, all methods for computing genome alignments apply efficient local alignment methods. To achieve computational efficiency, many local alignment programs sacrifice sensitivity. This is acceptable for other applications of local alignments where only some good local alignment is sought. However, genome alignments become less accurate when local alignments are missing. One of the contributions of this thesis is an efficient and fully sensitive local alignment approach.

Representation in data structures. Although the computation of local alignments hugely reduces the search space, the set of local alignments still constitutes a large amount of data. Likewise, the final genome alignment usually represents more information than can be manually captured. Data structures help to efficiently store, explore, and process such large amounts

of data. A data structure for sets of local alignments furthermore combines and integrates the information from individual local alignments.

The output of local alignment approaches is typically a list of local alignments. This list might or might not be ordered, for example by the start positions in one genome. However in a genome alignment with translocations and duplications, such an ordered list of local alignments is useful only in the special case where one genome serves as a reference. Otherwise, a linear order of the local alignments is uninformative.

Methods for computing genome alignments benefit from more sophisticated data structures than lists, predominantly from graphs. Graphs are widely used and well-studied data structures, and thus provide standard solutions to many problems. Genome alignment methods can apply these solutions and profit from previous research (for example `Mugsy` [9] solves a min-cut max-flow problem [61]). One class of methods is entirely based on graphs (including for example `ABA` [133] and `Sibelia` [109]). Here, the graphs assist in the processing of local alignments, conceptually support the algorithm development, and finally visualize the resulting genome alignment.

The precise definition of the graphs influences the amount of information represented and visualized by the data structure. Ideally, the definition directly leads to genome alignment tasks represented by standard graph problems. Unfortunately, the paucity of discussion of differences and distinguishing characteristics of the graph definitions that have been suggested for genome alignments currently impedes broad understanding of the advantages. A second contribution of this thesis is a detailed comparison of four graph data structures.

Processing the set of local alignments. Before the set of local alignments can be considered as a genome alignment, methods for computing genome alignments have to ensure that each nucleotide is part of at most one multiple local alignment. Furthermore, the methods evaluate the probability for homology of each individual local alignment or of parts of the local alignments in the context of the other local alignments. Based on this evaluation, the methods combine local alignments, resolve conflicts, add further local alignments, or remove spurious local alignments.

An overview of all processing steps goes beyond the scope of this thesis. The following paragraphs address three important steps: The refinement of partially overlapping local alignments, the extension of consistent local alignments, and the selection of a subset of local alignments. While refinement and consistency have been adopted from colinear alignment methods, the selection of a subset of local alignments distinguishes methods for computing genome alignments.

Refinement of partial overlaps. Given the set of local alignments for evolutionarily-related genomes, it is very likely that some of the local alignments overlap. If the local alignments are only pairwise, overlaps are even necessary to be able to predict homologies conserved across more than two genomes. For example a region conserved in three genomes will appear as three local alignments, one between each pair of genomes.

In this case, where a segment is conserved in three genomes, the local alignments completely overlap from start to end of the segments. This allows combining of the overlapping pairwise alignments to one multiple alignment. However, in many cases alignments only partially overlap.

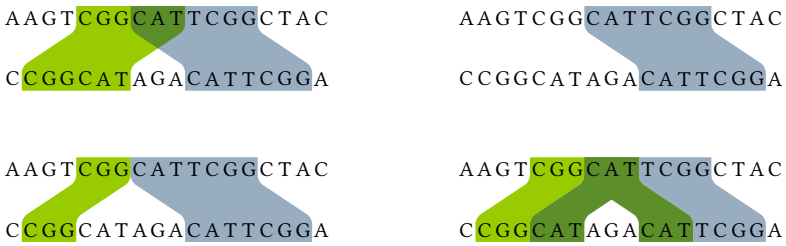

Figure 1.10: Three ways of resolving the partial overlap (top left) of two local alignments (green and blue): selection of only the blue local alignment (top right), trimming of the green local alignment (bottom left), and refinement into three local alignments (bottom right). The refinement leads to a three-way alignment of CAT, which implies a duplication.

In partial overlaps, one end of a segment in a local alignment overlaps with an end of a segment from another local alignment (see Fig. 1.10, top left). In order to combine the local alignments such that each nucleotide is part of at most one local alignment, the partial overlaps need to be resolved.

Partial overlaps can be addressed in various ways (see Fig. 1.10). The authors of the `Enredo` method simply circumvent the problem by preferring the local alignments "to be short because this makes it easier to avoid overlap between the ends" [124]. But even if local alignments are short, overlaps can occur and in addition we risk overlooking local similarity. Another option is to treat partial overlaps as conflicts and remove either of the local alignments from the genome alignment. This is unfavorable, however, if the ends of long alignments overlap by only a few nucleotides. As an alternative, we can trim the overlapping end from either of the alignments, but again this is unfavorable if the overlap is long.

A possible solution is segment match refinement, a procedure that subdivides local alignments into segments that are either disjoint or identical (see Fig. 1.10 and Fig. 1.9). The algorithm for segment match refinement repeatedly projects end positions of local alignments that fall in the middle of other local alignments to the aligned segments so as to partition local alignments into fully or non-overlapping parts. In contrast to the other approaches, segment match refinement resolves partial overlaps without losing similarity information. It was first described for pairs of sequences [69] and later extended to the multiple case [135]. The genome aligner `Mugsy` [9], for example, adopted the implementation of segment match refinement from the colinear aligner `SeqAn::T-Coffee` [135].

The advantage of segment match refinement over the other approaches is that it leaves the removal of local alignments to later steps. The only risk is that refinement subdivides the local alignments into many short local alignments. However, some of them can be recombined to longer conserved segments after selecting a subset of the local alignments (see AGCATGA in Fig. 1.9).

Consistency extension. Before selecting a subset of the local alignments, a re-evaluation step has been proven beneficial for accuracy. Methods for computing local alignments evaluate and

report alignments based on similarity and length of individual local alignments only. The consistency extension step additionally evaluates them based on their consistency with other local alignments in the set. Consistency was established by Notredame *et al.* [118, 119] and later used in many other methods for computing multiple alignments, for example in `ProbCons` [47] and `Pecan` [123].

Commonly, the consistency extension step examines all pairs of alignments that share a segment, for example an alignment of the segments A and B and an alignment of the segments A and C. For these pairs, it checks whether a transitive alignment of segments B and C is present in the set of local alignments. If it is not present in the set, the transitive alignment is added. If it is present, the transitive alignment is rewarded for consistency by increasing its score (see red outlines in Fig. 1.9). After this re-evaluation (often called triplet extension), the score of the alignment between B and C reflects the number of intermediate segments A to which local alignments were initially identified.

The idea of the extension step is to reward consistent local alignments because they are less probable to occur by chance. Thus, they are more likely to represent homologies. Consistency is founded on transitivity of the homology relation. The extension step approximates transitivity in the set of local alignments. The re-evaluated scores transmit this information to the subsequent steps and assist in deciding which local alignments to keep for the final genome alignment.

Subset selection. Methods for computing genome alignments select a subset of the local alignments – under the assumption that some of the local alignments are spurious. Depending on the specificity of the initially applied local alignment approach and the intended degree of genome segmentation, this step removes a small or large fraction of local alignments from the final genome alignment. The idea is that a segmentation into small local alignments is unlikely to represent true homologies; in related genomes, longer regions are expected to be conserved. While the goal is still to align as many nucleotides as possible, the local alignments are maximized both for length and number of aligned segments.

For computing colinear alignments, the selection of a subset is a clearly defined optimization problem: selecting the subset with the largest overall alignment score that conforms with colinearity. Without the colinearity constraint, the full set of local alignments has the largest overall alignment score. Nevertheless, methods for computing genome alignments intend to reduce this set and specify additional criteria that are characteristic for genome alignments.

Many genome alignment methods follow one of two strategies: optimizing an extended scoring function or solving a subgraph problem on a graph data structure. The program `progressiveMauve` [37] is an example for the first strategy. It defines a score that integrates the total alignment score with the segmentation of the genomes. This integrated score adds positive scores for the selected local alignments and applies penalties for breakpoints between pairs of genomes. However, the pairwise breakpoint penalty does not take full advantage of the multiple genomes available but accounts only for rearrangements visible between pairs. The third contribution of this thesis is a counting method for breakpoints in comparisons of multiple genomes.

The second strategy for subset selection is founded on the observation that non-colinear changes create cycles in the graph representations. The `ABA` method [133] for example, solves the maxi-

mum subgraph with large girth problem (MSLG), and thus forbids cycles smaller than a specified girth parameter while keeping the graph as large as possible. The comparison of graph data structures in this thesis discusses how the appearance of non-colinear changes as cycles depends on the precise definition of the graph data structures.

Both `progressiveMauve` and `ABA` regulate the segmentation of the resulting genome alignments with a parameter: the breakpoint penalty in the integrated scoring function or the girth parameter in the subgraph problem. These parameters eventually determine the trade-off between the length of local alignments and the total number of aligned nucleotides.

The algorithms for selecting the best subset of local alignments under the scoring function or subgraph problem are usually greedy. Many methods start with an empty set and iteratively add local alignments. The reverse is also possible, removing local alignments from the initial set. Combining both, the `Cactus` method [121], for example, adds and removes local alignments iteratively in order to obtain a subset of local alignments that best predicts homologous regions.

Finishing the genome alignment. After processing the set of local alignments, many methods post-process the genome alignments to improve accuracy. On the one hand, some regions of the genomes may be left unaligned. On the other hand, some consecutive local alignments can be combined to one longer local alignment after selecting a subset of the local alignments (for example the alignments of AGCA and TGA into block 6 in Fig. 1.9).

In unaligned regions and in newly combined local alignments, improvements are typically possible. The following describes a recursion step for filling unaligned gaps in a genome alignment and, briefly, a realignment step for refining the alignments on the nucleotide-level. (Both steps are not necessary in the toy example of Fig. 1.9.)

Recursion. Ideally, regions of the genomes remain unaligned only if they are unique to one genome and not homologous to any other region. However, often there are weak similarities that the initial local alignment step did not identify. A recursive search for local alignments in unaligned regions can identify such similarities using the previously identified local alignments as anchors to reduce the search space. A smaller search space allows for more sensitive parameters that tolerate more differences of the aligned regions.

Recursively filling gaps between the local alignments is also relevant in methods for computing colinear alignments. For pairwise colinear alignments, the search space for recursive calls is straightforward due to the colinearity constraint. For multiple colinear alignments, the search space is still clearly defined with additional potential to reduce it even further (see Fig. 1.11). Methods for computing genome alignments have to define additional restrictions to the search space since an initially unaligned region can align to any other region of all genomes.

The genome aligner `progressiveMauve` [37] recursively computes local alignments for pairs of genomes. It searches right and left of all previously identified similarities allowing only new alignments in regions unaligned between the compared pair of genomes. The assumption is that conservation continues over longer conserved regions than the first local alignment step identified. Therefore, this recursive step does not identify new rearrangements.

Figure 1.11: Reduction of the search space for a recursive local alignment search in *colinear* multiple alignments. Assume all blue local alignments were selected from the initial set of local alignments. Clearly, the dark blue local alignments limit the search space for a recursive call on all three genomes. The light blue local alignments reduce the search space for pairwise local alignments between the top and the middle genome and between the middle and bottom genome. In addition, they constrain the search space with colinearity for the top and bottom genome (for example alignments between the segments marked in red are in conflict with the light blue local alignments). Only comparing regions unaligned to all genomes is not sufficient as the valid green alignment shows.

Usually, a recursive step is only applied if the genome alignment method runs the local alignment approach internally. After detecting new local alignments, the processing steps proceed as before. Recursion can be repeated until no new local alignments are selected, although an earlier termination can avoid spending a large portion of the running time on minor improvements [37].

Colinear realignment. When several consecutive local alignments are combined to a single local alignment, a realignment of the colinear segments can correct misplaced gaps. This improves the accuracy of predicting substitutions and short insertions and deletions. Methods for computing genome alignments separately realign each set of local alignments that are consecutive in all alignment rows. The realignment method can be any colinear multiple global alignment approach (for example `Muscle` [51] in `progressiveMauve` [37]). Although realignment does not influence the overall prediction of homology, the resulting genome alignment achieves a higher accuracy at the nucleotide level.

Outline of the thesis

This thesis contributes to three aspects of computing and modeling whole-genome alignments: the detection of local similarities, the representation in graph data structures, and the selection of local similarities. Chapters 3 to 5 each cover one of these aspects. A chapter that precedes the main chapters introduces terminology and mathematical notation, and the last chapter concludes the thesis with a summary, discussion, and outlook.

Specifically, Chapter 2 covers relevant mathematical terminology for sets and orders, defines biological sequences, provides formal background on alignments, introduces vocabulary for rearrangement analysis, and reviews graph theoretical terms. All following chapters rely on the definitions provided in this chapter. While Section 2.1 and 2.5 review common and widely-used terminology and notation, the middle sections of the chapter suggest consistent definitions for only partly-established vocabulary.

Chapter 3 describes a new local alignment approach implemented in the program `STELLAR`. The approach in `STELLAR` can be used on whole genomes due to its efficiency. In addition, it guaran-

tees full sensitivity within a clearly specified quality definition for local alignments. It is therefore well suited for application in methods for computing genome alignments. The chapter demonstrates full sensitivity of STELLAR with a theoretical proof and substantiates the exactness with comprehensive testing.

Chapter 4 reviews and compares the definitions of graph data structures for genome alignments. The main contribution of this chapter is a comparison between four previously introduced graphs using consistent terminology and notation. The comparison includes formal transformations between graph structures, the appearance and identification of substructures such as cycles, and modification operations that select local alignments. This reveals weaknesses of the four graph structures in representing non-colinearity and indicates their advantages over each other.

Chapter 5 contributes to the subset selection step of local alignments by introducing hidden breakpoints. Hidden breakpoints are indicators of rearrangement that become visible only in the comparison of three or more genomes. The number of hidden breakpoints can serve as a metric for measuring the degree of non-colinearity in scoring functions for subset selection. The chapter suggests a method for calculating hidden breakpoint counts. The method resorts to the standard graph theoretical problem of maximum weight perfect matching. A comparison of pairwise and hidden breakpoint counts in simulated alignments confirms the suitability of hidden breakpoint counts for measuring the degree of non-colinearity in genome alignments.

Finally, Chapter 6 summarizes the contributions of this thesis, discusses how the results of the previous chapters can be combined and complemented in a new method for computing genome alignments, and addresses current developments for evaluating and modeling genome alignments.

Chapter 2

Theoretical preliminaries

This chapter introduces the terminology and notation used throughout the thesis. The first part briefly mentions relevant basics about sets and orders. The second part introduces biological sequences including genomes, chromosomes, and segments. Furthermore, the third part provides definitions of alignment concepts and introduces a dotplot and traceback for pairwise alignments. The fourth part establishes vocabulary for the analysis of genome rearrangement. Finally, the last part introduces common graph theoretical terms.

The reader familiar with basic terminology and common notation may skip Parts 2.1 and 2.5. Some of the terms introduced in Parts 2.2 to 2.4 for biological sequences, sequence alignments, and genome rearrangement, however, are used less consistently in the literature. The precise definitions given in these parts are especially relevant for Chapter 4.

2.1 Sets and orders

Following common notation, curly braces denote unordered sets in this thesis, and parenthesis denote ordered pairs or tuples. Given a set $X = \{x_1, \ldots, x_k\}$, $|X|$ denotes the *size* k of X. A set with a size $|X| = 0$ is called *empty* and written as $X = \{\}$. For a non-empty set X, 2^X denotes the *power set*, which is the set of all possible subsets $Y \subseteq X$.

For definitions of basic operations on sets (union, intersection, difference, etc.) see for example [107]. Two sets X and Y are *disjoint* if their intersection $X \cap Y = \{\}$ is empty. A set of n subsets $X_1, \ldots, X_n \subseteq X$ is a *partition* of the set X if $X = \bigcup_{1 \leq i \leq n} X_i$ is the union of all subsets, all subsets $X_i \neq \{\}$ are non-empty, and any two subsets X_i and X_j are disjoint for all $i, j \in \{1, \ldots, n\}$ with $i \neq j$.

Given a set $X = \{x_1, \ldots, x_{|X|}\}$, a *partial order* over X is a relation $\preceq$ where for all elements

$x_i, x_j, x_k \in X$

- $x_i \preceq x_i$ (reflexivity),
- $x_i \preceq x_j$ and $x_j \preceq x_i$ implies $x_i = x_j$ (antisymmetry), and
- $x_i \preceq x_j$ and $x_j \preceq x_k$ implies $x_i \preceq x_k$ (transitivity).

A *total order* over X is a relation $\preceq$ that satisfies the conditions for a partial order and, in addition, all elements $x_i, x_j \in X$ are mutually comparable: $x_i \preceq x_j$ or $x_j \preceq x_i$. A *strict total order* over X is a relation $\prec$ that satisfies the conditions for a total order, but either $x_i \prec x_j$ or $x_j \prec x_i$ or $x_i = x_j$ is true for any two $x_i, x_j \in X$ (trichotomy).

An *alphabet* Σ is a set of characters. In the *DNA alphabet* $\Sigma = \{A, C, G, T\}$, each character $\sigma \in \Sigma$ has a complementary character $\overline{\sigma} \in \Sigma$: $\overline{\sigma} = T$ if $\sigma = A$, $\overline{\sigma} = G$ if $\sigma = C$, $\overline{\sigma} = C$ if $\sigma = G$, and $\overline{\sigma} = A$ if $\sigma = T$.

2.2 Biological sequences

A *sequence* of length l over an alphabet Σ is a l-tuple $seq = (\sigma_0, \ldots, \sigma_{l-1})$ of characters $\sigma_i \in \Sigma$ with $0 \leq i < l$. In the following, we write $seq = \sigma_0\sigma_1 \ldots \sigma_{l-1}$ without parentheses and commas for brevity. We use $|seq|$ to denote the *length* l of the sequence seq. A sequence of length $|seq| = 0$ is called an *empty* sequence.

A *chromosome* is a sequence over the *DNA alphabet*. We can read the sequence of a chromosome chr in the *forward orientation* $\sigma_0 \ldots \sigma_{|chr|-1}$ or in the *reverse complemented orientation* $\overline{\sigma}_{|chr|-1} \ldots \overline{\sigma}_0$. Note that the assignment of the forward orientation to one strand of a DNA double helix and the reverse complemented orientation to the other strand is in many cases arbitrary.

A chromosome is either linear or circular. We model a *linear chromosome* to be bounded by zero-length *telomeres* denoted by •. For example, • $TACTG$ • is a linear chromosome of length five. We acknowledge the discrepancy to the biological literature, where a telomere is a non-empty sequence near the ends of a linear chromosome. In a *circular chromosome*, the last character $\sigma_{|chr|-1}$ is followed by the first character σ_0. Thus, circular chromosomes do not have telomeres.

A *genome* is a set of chromosomes $g = \{chr_1, \ldots, chr_K\}$. We define the length of g as $|g| = |chr_1| + \cdots + |chr_K|$. Let $\mathcal{G} = \{g_1, \ldots, g_N\}$ be a set of genomes with $g_n = \{chr_1^n, \ldots, chr_{K_n}^n\}$ for $1 \leq n \leq N$. Given an arbitrary strict total order on the set of genomes $g_1 < \cdots < g_N$ and on the set of chromosomes of each genome $chr_1^n < \cdots < chr_{K_n}^n$, we can refer to the j^{th} character in chromosome chr_k^n of genome g_n, where $1 \leq k \leq K_n$ and $1 \leq n \leq N$, as a *position* $i = |g_1| + \cdots + |g_{n-1}| + |chr_1^n| + \cdots + |chr_{k-1}^n| + j$. Furthermore, we define the set of all positions of the set of genomes $\mathcal{G}$ as $P_{\mathcal{G}} = \{i \mid 0 \leq i < \sum_{n=1}^{N} |g_n|\}$.

A *segment* of a genome in $\mathcal{G}$ is a 3-tuple $s = (i, l, o)$ of a start position $i \in P_{\mathcal{G}}$, a length $l \in \mathbb{Z}$, and a bit $o \in \{+, -\}$ that determines the orientation of the segment. The positions i and $i + l - 1$ have to fall in the same chromosome. If $o = +$, the segment is in the *forward orientation* and reads $\sigma_i \ldots \sigma_{i+l-1}$. If $o = -$, the segment is in the *reverse complemented orientation* and

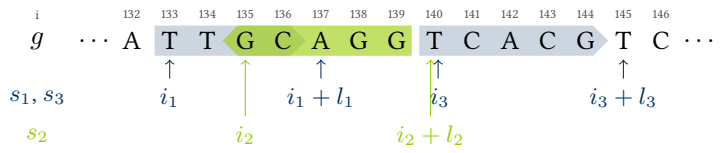

Figure 2.1: Three segments s_1, s_2, and s_3 of a genome g. The segments s_1 and s_3 are in forward orientation (blue). Segment $s_1 = (133, 4, +)$ reads TTGC, and segment $s_3 = (140, 5, +)$ reads TCACG. Segment $s_2 = (135, 5, -)$ is in reverse complemented orientation (green) and reads $\overline{\text{GCAGG}}$ = CCTGC. s_1 and s_2 are partially overlapping, thus, not adjacent. s_1 and s_3 are non-overlapping but not adjacent. s_2 and s_3 are non-overlapping and adjacent at position 140.

reads $\overline{\sigma}_{i+l-1} \dots \overline{\sigma}_i$ (see Fig. 2.1). We call the segment $\overline{s} = (i, l, o')$ the *reverse complement* of the segment $s = (i, l, o)$ if $o' \neq o$. If $l = 0$, the segment is *empty*.

We denote by $p \in s$ that $i \leq p < i + l$, i. e. the position p falls in the segment $s = (i, l, o)$. A *segment s' of the segment s* is again a segment: $s' = (i', l', o')$ with $i' \in s$ and $(i' + l' - 1) \in s$. For a segment in the forward orientation, we refer to the position i as the *tail* and to the position $i + l - 1$ as the *head* of the segment. For a segment in the reverse complemented orientation, we refer to the position i as the *head* and to the position $i + l - 1$ as the *tail* of the segment.

Given a set of segments $\mathcal{S}$ defined on a set of genomes $\mathcal{G}$. Two segments $s_1, s_2 \in \mathcal{S}$ with $s_1 = (i_1, l_1, o_1)$ and $s_2 = (i_2, l_2, o_2)$ can relate to each other in several ways (see also Fig. 2.1). Without loss of generality (w. l. o. g.) let $i_1 \leq i_2$. The two segments s_1 and s_2 are *non-overlapping* if $i_1 + l_1 \leq i_2$, *fully overlapping* if $i_1 = i_2$ and $l_1 = l_2$, and *partially overlapping* otherwise.

Two non-overlapping segments $s_1 = (i_1, l_1, o_1)$ and $s_2 = (i_2, l_2, o_2)$ are *adjacent* if no segment $s_3 \in \mathcal{S}$ exists such that there is a position $p \in s_3$ with $i_1 + l_1 \leq p < i_2$. In other words, two non-overlapping segments are adjacent if no segment between them exists in the set of segments $\mathcal{S}$. The two adjacent segments define an *adjacency* $s_a = (i_1 + l_1, i_2 - (i_1 + l_1))$, which is a segment without orientation. If the adjacency is an empty segment, we can refer to it by a single *adjacency position* $a = i_2 = i_1 + l_1$.

2.3 Sequence alignments

We give the following definitions only for segments. Note that they apply to sequences (and chromosomes and genomes), too, by replacing a sequence seq by the segment $s = (0, |seq|)$.

In the most general case, an *alignment* of a set of segments $\mathcal{S}$ is a set of alignment columns, where an *alignment column* is a non-empty set of positions $A \subseteq P_{\mathcal{S}} = \{p \mid \exists s \in \mathcal{S} : p \in s\}$. We say that positions in the same alignment column are *aligned* to each other. The following three paragraphs describe conditions that an alignment must fulfill to be classified as local or global, pairwise or multiple, and colinear or non-colinear (see also Fig. 2.2 and 2.3).

An alignment is called a *global alignment* if every position $p \in P_{\mathcal{S}}$ is an element of at least one alignment column. In a *local alignment*, only positions from segments of the segments $\mathcal{S}$ are

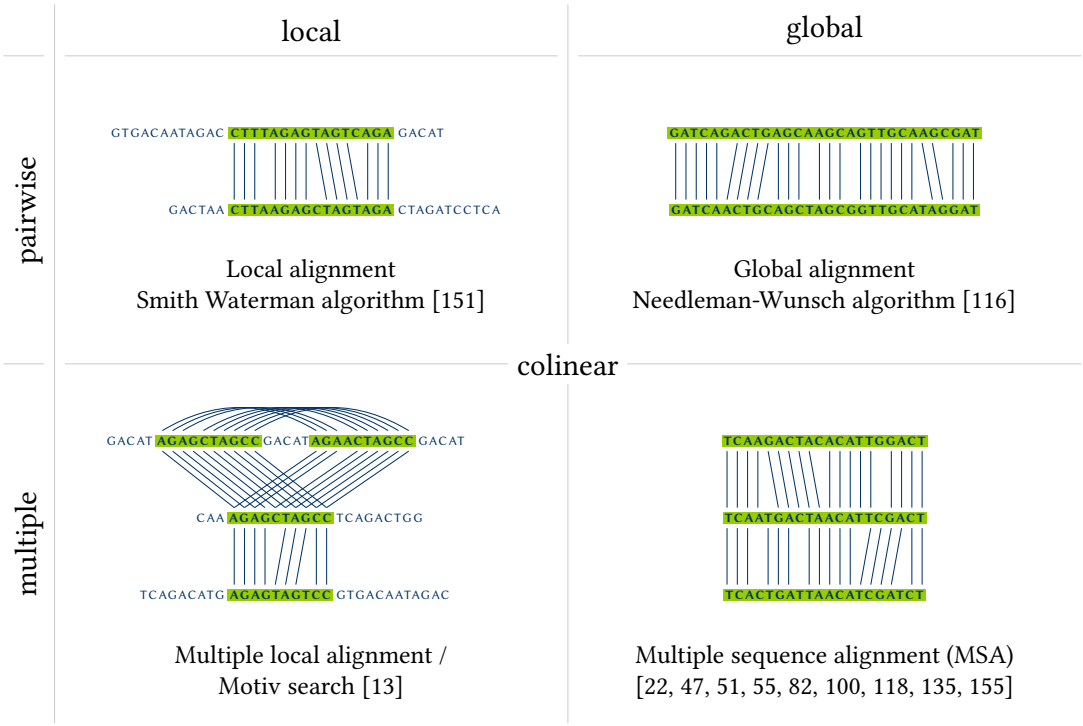

Figure 2.2: The classifications into local or global, pairwise or multiple, and colinear or non-colinear alignments are independent from each other. The figure shows types of colinear alignments, which allow only mismatches, insertions, and deletions as differences among the aligned segments. In addition, the figure lists example approaches to calculate each type of colinear alignment. Chapter 3 addresses the local alignment problem (top left).

elements of alignment columns. A special case is a *semi-global alignment* of two segments where one segment is aligned globally and the other locally. Note that a global alignment does not necessarily align every positions to other positions since alignment columns can consist of only one position.

An alignment is called a *pairwise alignment* if the positions in the alignment columns originate from at most two segments. If the positions originate from more than two segments, the alignment is a *multiple alignment*. If duplications are present, alignment columns of pairwise alignments can contain more than two positions. A pairwise local alignment can align two segments from the same sequence. The number of segments aligned by a multiple local alignment is not upper bounded by the number of segments in $\mathcal{S}$ but by the number of segments of segments in $\mathcal{S}$; the number of aligned segments per input sequence would be an additional condition for local alignments.

To describe the classification into colinear and non-colinear, we define the relation $\prec$ for two alignment columns A_1 and A_2 as follows: $A_1 \prec A_2$ if every pair of two positions $p_1 \in A_1$ and $p_2 \in A_2$ fulfills $p_1 < p_2$ if p_1 and p_2 belong to the same forward oriented segment, and $p_2 < p_1$ if they belong to the same reverse complemented segment. An alignment is *colinear* if the relation

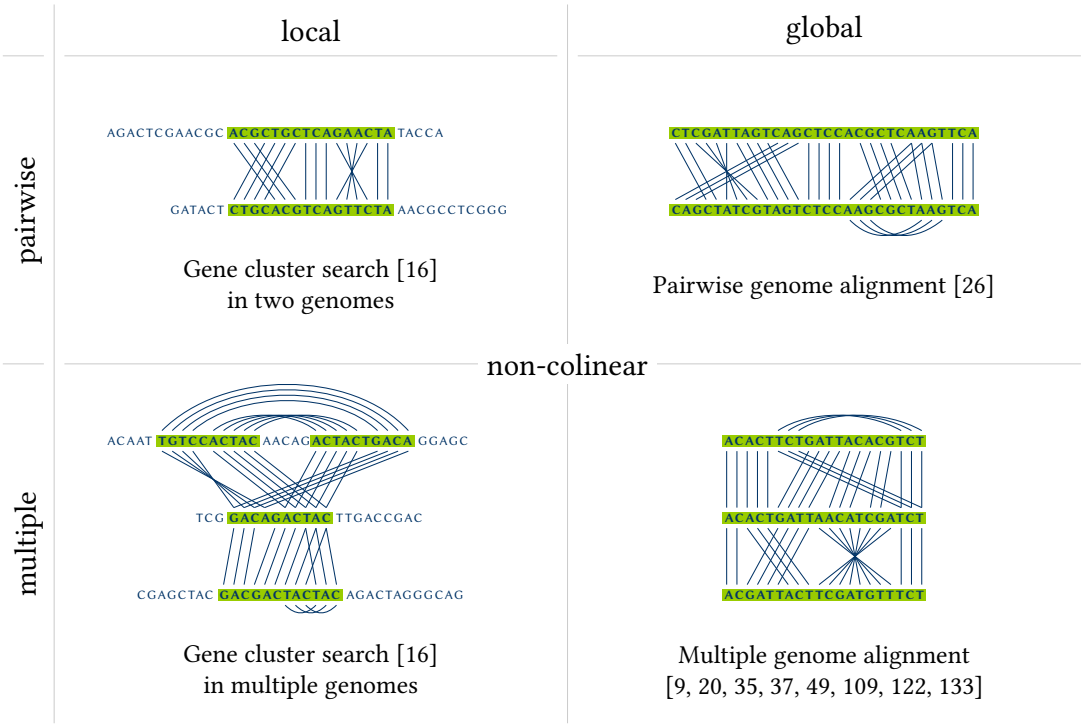

Figure 2.3: The classifications into local or global, pairwise or multiple, and colinear or non-colinear alignments are independent from each other. The figure shows types of non-colinear alignments, which allow inversions, duplications, and translocations among the aligned segments as opposed to colinear alignments. In addition, the figure lists example approaches to calculate each type of non-colinear alignment. The focus of this thesis are multiple genome alignments (bottom right).

$\prec$ imposes a strict total order on its alignment columns and each alignment column contains at most one position from each segment (see Fig. 2.2). If any of the two conditions does not hold, the alignment is *non-colinear* (see Fig. 2.3).

The most common representation of colinear alignments is the *tabular alignment representation*. In this representation, a colinear alignment is written as a matrix where each row corresponds to a segment and each column to an alignment column (see Fig. 2.4). Some matrix entries remain empty because of alignment columns with fewer positions than there are segments. These entries are commonly filled with the *gap character* "$-$".

The alignment columns of a colinear alignment can be classified into match, mismatch, and gap columns (see Fig. 2.4, right). An alignment column A that contains a position from every aligned segment is a *match column* if there is the same character at all positions $p \in A$, and otherwise a *mismatch column*. If A contains fewer positions than there are segments, it is a *gap column*. Both mismatch and gap columns are *error columns*.

Given two segments s_1 and s_2, a *dotplot* is a $|s_1| \times |s_2|$ matrix where each row corresponds to a position from s_1 and each column to a position from s_2 (see Fig. 2.5). A dotplot can visualize similarities of two sequences if matrix entries that represent matching characters are marked,

Figure 2.4: Tabular representation of a colinear multiple alignment of five sequences in a 5×16 matrix (left) and of a colinear pairwise alignment of the two sequences GTTAA and GTGCA in a 2×6 matrix (right). Match columns in the pairwise alignment are shaded in green and marked with vertical lines inbetween the two rows.

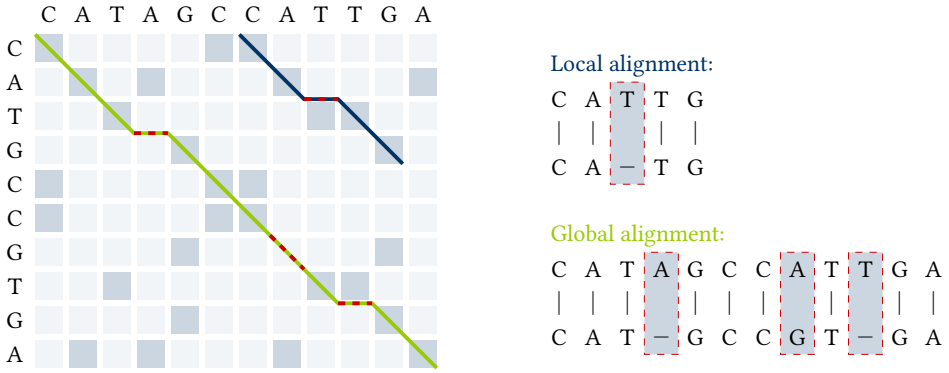

Figure 2.5: A dotplot of the two sequences s_1 = CATAGCCATTGA and s_2 = CATGCCGTGA (left) and the tabular representations of a local and a global alignment. Entries in the dotplot that correspond to matching characters are shaded in darker blue. The traceback of the global alignment (green) reaches from the upper left to the lower right corner. The traceback of local alignments (for example the dark blue one) can start and end at arbitrary positions of the dotplot. Red dashes mark error columns.

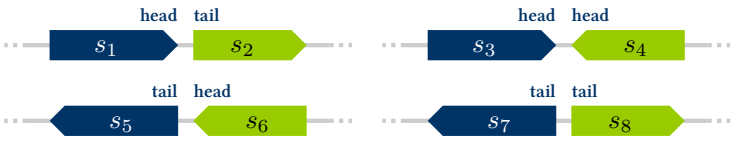

Figure 2.6: Four different adjacencies of two blocks. Both the blue block $B_1 = \{s_1, s_3, s_5, s_7\}$ and the green block $B_2 = \{s_2, s_4, s_6, s_8\}$ have four occurrences each being adjacent to an occurrence of the respective other block. For example, the occurrences $s_1 \in B_1$ and $s_2 \in B_2$ are adjacent at the head of s_1 and the tail of s_2 (top left). Thus, the head of B_1 is adjacent to the tail of B_2. Furthermore, the two heads of the blocks are adjacent (top right), the tail of B_1 and the head of B_2 are adjacent (bottom left), and the two tails of the blocks are adjacent (bottom right).

for example with a dot. A *traceback* is a route through the dotplot that corresponds to a colinear alignment. In a traceback, match and mismatch columns of the alignment appear as diagonals, while gap columns are horizontal or vertical. Horizontal lines correspond to gaps in s_1 and vertical lines to gaps in s_2.

2.4 Genome rearrangement

A *block* B is a set of segments of a given set of genomes. We assume segments in the same block to be "similar" in sequence, although this is no requirement for the following definitions. One may imagine blocks as local colinear alignments.

The *size* $|B|$ denotes the number of segments in the block B. Each segment $s \in B$ is called an *occurrence* of B in the respective chromosome (or genome). A block can occur multiple times per chromosome.

Analogously to segments, blocks have a head and a tail. The *tail* of a block $B = \{s_1, \ldots, s_{|B|}\}$ is the set of tails of all segments $s_j \in B$ with $j = 1, \ldots, |B|$, and the *head* is the set of heads of all $s_j \in B$. We sometimes refer to the head and the tail of a block as its two *ends*.

The *complement* of B is a block $\overline{B} = \{\overline{s}_1, \ldots, \overline{s}_{|B|}\}$, where each segment is in the opposite orientation. The tail of B is the head of $\overline{B}$, and the head of B is the tail of $\overline{B}$. Note that the relative orientation of all segments in a block is significant, but the assignment of B or $\overline{B}$ to one orientation is arbitrary.

Let $\mathcal{B}$ be a set of blocks defined on a set of genomes $\mathcal{G}$. Two blocks $B_1, B_2 \in \mathcal{B}$ are *non-overlapping* if all pairs of segments $s_1 \in B_1$ and $s_2 \in B_2$ are non-overlapping. B_1 and B_2 are *adjacent* if two segments $s_1 \in B_1$ and $s_2 \in B_2$ exist that are adjacent. The two blocks can be adjacent to each other in more than one segment per block and in up to four different ways (see Fig. 2.6). If a block contains two adjacent segments (i. e. two occurrences of the block are adjacent), the block is adjacent to itself.

Two adjacent blocks B_1 and B_2 define a *breakpoint* if an occurrence of one of the blocks exists that is not adjacent to an occurrence of the other block or if the two blocks are adjacent in more than one way. In other words, given two adjacent segments $s_1 \in B_1$ and $s_2 \in B_2$, w. l. o. g. let the tail of s_1 be adjacent to the head of s_2, then, B_1 and B_2 define a breakpoint if a segment $s_1' \in B_1$ exists whose tail is not adjacent to the head of any segment $s_2' \in B_2$. And to put it the other way, two blocks are adjacent without breakpoint if $|B_1| = |B_2|$ and they are adjacent in all segments in the same way.

A set of blocks $\mathcal{B}$ is called a *tiling* of a set of genomes $\mathcal{G}$ if every position $p \in P_{\mathcal{G}}$ in the genomes belongs to exactly one block occurrence: For every $p \in P_{\mathcal{G}}$, there is a $B \in \mathcal{B}$ such that $s \in B$ exists where $p \in s$. In a tiling $\mathcal{B}$, the blocks are non-overlapping. Furthermore, every adjacency is an empty segment and can be referred to by an adjacency position.

Given all adjacencies of segments from a set of non-overlapping blocks $\mathcal{B}$, we can write chromosomes as sequences of the block occurrences, with telomeres bounding linear chromosomes.

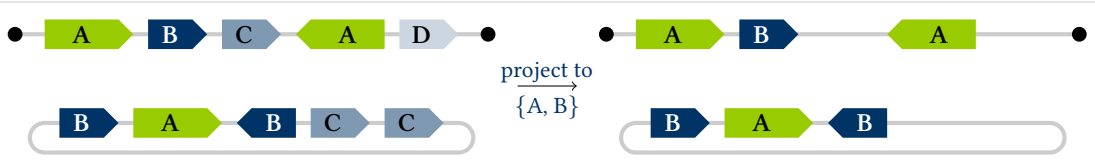

Figure 2.7: A linear chromosome $\bullet ABC\overline{A}D\bullet$ and a circular chromosome $BA\overline{B}CC$ defined on the set of blocks $\mathcal{B} = \{A, B, C, D\}$ (left), and their projection $\bullet AB\overline{A}\bullet$ and $BA\overline{B}$ to the subset $\mathcal{B}' = \{A, B\}$ (right).

Instead of segments, we use the block identifiers to refer to block occurrences in a chromosome. For example, $\bullet ABC\overline{A}D\bullet$ denotes a linear chromosome on the set of blocks $\mathcal{B} = \{A, B, C, D\}$ (see Fig. 2.7).

Using this representation of chromosomes, we may *project* chromosomes to a subset of the blocks $\mathcal{B}$. In the *projection* of a chromosome to a subset of blocks $\mathcal{B}' \subseteq \mathcal{B}$, only the blocks in $\mathcal{B}'$ appear. For example, the projection of the chromosome $\bullet ABC\overline{A}D\bullet$ to the subset of blocks $\mathcal{B}' = \{A, B\}$ is $\bullet AB\overline{A}\bullet$. A projection of a chromosome may create new adjacencies with respect to the chromosome defined on the full set of blocks $\mathcal{B}$. In the above example, the heads of A and B are adjacent in the projection to $\mathcal{B}'$ but not in the chromosome defined on $\mathcal{B}$. (see Fig. 2.7).

2.5 Graph theory

A *graph* is an ordered pair $G = (V, E)$ of a set of vertices V and a set of edges E. We denote an edge $e \in E$ as a pair of vertices $u, v \in V$. An ordered pair $e = (u, v)$ represents a *directed edge*, and an unordered pair $e = \{u, v\}$ an *undirected edge*. If all edges in E are undirected, we call G an *undirected graph*; if all edges are directed, G is a *directed graph*; and if both undirected and directed edges are present, we refer to G as a *mixed graph*.

Two vertices $u, v \in V$ that are connected by an edge $e = (u, v)$ or $e = \{u, v\}$ are called the *endpoints* of e. We say that the edge e is *incident* to the two vertices u and v. If $e = (u, v)$ is directed, the vertex u is called the *source vertex* of the edge and the vertex v is called the *target vertex* of e. We refer to an edge (v, v) or $\{v, v\}$ that is incident to only one vertex as a *loop*.

A graph $G' = (V', E')$ is called a *subgraph* of G if $V' \subseteq V$ and $E' \subseteq E$. We say that G *contains* its subgraphs G'. W. l. o. g. let G be a directed graph. If $E' = \{(u, v) \in E \mid u, v \in V'\}$, we call G' *vertex-induced* by the set of vertices V'. Likewise, if $V' = \{u, v \in V \mid (u, v) \in E'\}$, we call G' *edge-induced* by the set of edges E'.

Let $E_v \subseteq E$ be the set of edges incident to a vertex $v \in V$. We call v a *branching vertex* if the set of vertices in the subgraph induced by E_v is larger than three. Otherwise, v is a *non-branching vertex*. The *degree* of a vertex v is the number of edges incident to v where loops count twice. We denote the degree of v by $d(v)$.

A *path* through a graph G is an edge-induced subgraph $G' = (V', E')$ of G with a strict total order on the edges $e_1 \prec e_2 \prec \cdots \prec e_{|E'|}$ where any two edges $e_i, e_{i+1} \in E'$ are both incident to

a vertex $v_i \in V'$ for all $i = 1, 2, \ldots, |E'| - 1$. We can write a path either as a sequence of edges $e_1 e_2 \ldots e_{|E'|}$ or as a sequence of vertices $v_0 v_1 \ldots v_{|E'|}$ including the first vertex v_0 and the last vertex $v_{|E'|}$ of the path that are incident to the first and the last edge, respectively. We say that the subgraph G' is a path from v_0 to $v_{|E'|}$.

In a *simple path* all vertices $v_0, v_1, \ldots, v_{|E'|}$ are distinct. A path where $v_0 = v_{|E'|}$ is a *cycle*, and a cycle with distinct vertices $v_1, v_2, \ldots, v_{|E'|}$ is a *simple cycle*. In a *mixed path* and in a *mixed cycle* both directed and undirected edges can be present in E'. A path $v_0 v_1 \ldots v_{|E'|}$ is *directed* if the source vertex v_i of any directed edge $e = (v_i, v_j)$ precedes the target vertex v_j in the sequence of vertices, thus $0 \leq i < j \leq |E'|$. An *Eulerian circuit* is a path with $E' = E$ that traverses every edge of the graph G.

A graph is *connected* if for every pair $u, v \in V$ a path exists from u to v. A connected subgraph is *maximal* if it is not contained in any other connected subgraph of G. Maximal connected subgraphs are called *connected components* of the graph. Given an integer $k \in \mathbb{N}$, a graph is *k-edge connected* if the minimal size of a subset of edges that disconnects the graph is greater than $k-1$. The maximal k-edge connected subgraphs are called *k-edge connected components*. We call a graph *E'-connected* if the subgraph induced by the edges $E' \subseteq E$ is connected. Similarly, we refer to maximal E'-connected subgraphs as *E'-connected components* of the graph. A graph is *complete* if every pair of vertices $u, v \in V$ is connected by an edge $e \in E$.

A *tree* is a connected graph without simple cycles. A vertex $v \in V$ is called a *leaf vertex* if it has a degree $d(v) = 1$, and an *inner vertex* if $d(v) \geq 1$. A tree is *binary* if every inner vertex v has a degree $d(v) \leq 3$, and *n-ary* if every inner vertex has a degree $d(v) \leq n + 1$. A tree is *rooted* if it has one designated vertex called the *root* of the tree.

Using *labeling functions*, we can associate the vertices and edges of a graph with additional information. For example, a labeling function $\ell_1 : V \to X$ assigns an element $x \in X$ to each vertex $v \in V$ such that $\ell_1(v) = x$. Similarly, a labeling function $\ell_2 : E \to Y$ assigns an element $y \in Y$ to each edge $e \in E$ such that $\ell(e) = y$. We call x a *vertex label* of v, and y an *edge label* of e.

A graph is *weighted* if its edges or vertices are labeled with numbers, for example $\ell : V \to \mathbb{N}$ or $\ell : E \to \mathbb{Z}$. The numbers are called *weights* of the vertices or edges, respectively.

A *multigraph* is a graph that can have multiple edges with the same pair of endpoints. The number of edges that connect the pair of endpoints is called the *multiplicity*. Multigraphs are often represented as graphs with only one edge per pair of endpoints labeled with the multiplicity. However, we represent multigraphs with multiple edges throughout this thesis.

Chapter 3

Local alignment detection for genome alignments

The local alignment method described in this chapter was developed in collaboration with David Weese and Knut Reinert, presented at RECOMB Satellite Workshop on Comparative Genomics 2011, and published in BMC Bioinformatics [89]:

B. Kehr, D. Weese, and K. Reinert. STELLAR: fast and exact local alignments. *BMC Bioinformatics*, 12 Suppl 9:S15, 2011.

This chapter describes a method for the detection of local pairwise alignments in genomic sequences, which has been implemented in a tool called `STELLAR`. The focus and primary demand during the development of the method was sensitivity on whole genomes. Sensitivity is a key factor for local aligners applied in genome alignment: While a wrong match can be weeded out later, missing local alignments may never be discovered in later genome alignment steps. `STELLAR` has full sensitivity with respect to a general scoring function but, nevertheless, is efficient to be applied on the genomic scale.

The first part of this chapter gives a general overview of algorithmic approaches to local alignment including various objective functions. Next, details of the alignment method implemented in `STELLAR` follow in the second part. The third part presents the results of systematic parameter tests of `STELLAR` and of a performance comparison to other local alignment approaches. The last part concludes the chapter with a summary, a short discussion, and an outlook.

3.1 Background on local alignment

The following describes various objective functions for local alignment and briefly explains the most common algorithmic strategies for finding local alignments efficiently. An overview of tools that implement these and other strategies is given at the end of this part of the chapter.

3.1.1 Objective functions for local alignment

This section describes the most prevalent objectives of local alignment approaches. The biological aim of local alignments is to recognize locally conserved sequence segments. By defining objective functions, the biological aim can be translated into an algorithmic aim: identification of sets of sequence segments that fulfill the criteria of the objective function. Effective local alignment algorithms often use combinations of several objectives. Underlying objective functions is the assumption that conserved segments distinguish themselves from the rest of the sequences by high similarity.

Exact matches. The simplest objectives are identical segments, for example *maximal exact matches (MEMs)* or *maximal unique matches (MUMs)*. MEMs are identical segments that cannot be extended by further matches at both ends. MUMs are MEMs that occur at most once in each input sequence and, thus, avoid highly repeated segments.

Similarity scores. Another class of well-established objectives for local alignments uses similarity scores. A *scoring scheme* assigns scores to the different types of alignment columns such that the sum of scores over all columns of an alignment is the overall alignment similarity score. Match columns get positive scores, error columns 0 or negative penalties. The objective is to find pairs of segments that align with a similarity score above a certain threshold.

Simple scoring functions use the same score for all match columns and the same penalties for mismatch and gap columns. More sophisticated scoring functions use *scoring matrices*, like PAM [38] or BLOSUM [77] for amino acid sequences or HOXD [31] for DNA sequences, which contain separate scores for any pair of characters in an alignment column.

Gap columns are typically penalized with linear or affine gap costs. *Linear gap costs* apply the same penalty for all gap columns, whereas *affine gap costs* have a separate penalty for the first of several consecutive gap columns. Even within the class of objective functions that use similarity scores, the scoring scheme has to be chosen with care since it can strongly influence the resulting set of local alignments [63].

Normalized distance measures. As opposed to similarity scores, distance measures are less informative for local alignments. Long sequence segments that are similar but not identical sometimes have larger distances than completely unrelated short segments. Thus, simple distance measures like the *Hamming distance* [70] and *edit distance* (also called *Levenshtein distance)* [98]

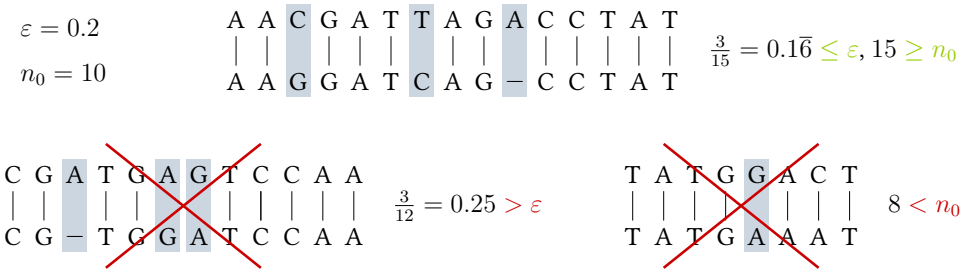

Figure 3.1: An ε-match and two local alignments that are no ε-matches given the maximal error rate $\varepsilon = 0.2$ and the minimal length $n_0 = 10$. The left alignment has an error rate that is too high although it has the same number of errors as the ε-match shown at the top. The right alignment is shorter than the minimal accepted length.

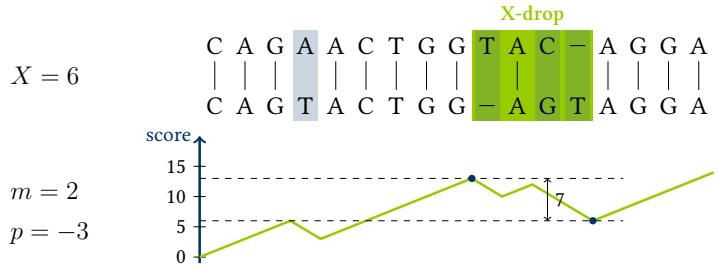

Figure 3.2: An X-drop in a local alignment, where $X = 6$, and the score along the alignment given a scoring scheme where a match scores $m = 2$ and an error $p = -3$.

do not apply to the local alignment problem. The Hamming distance forbids gaps and counts the number of mismatch columns. The edit distance counts gap and mismatch columns. Minimization of distance measures is useful for global and semi-global alignments, where the length of the alignments is given by the length of the input sequences, but not for local alignments.

An alternative for local alignments is the normalization of distance measures by the alignment length. The *normalized edit distance* [105] is the edit distance of an alignment divided by its length. Similarly, the notion of *ε-matches* describes local alignments with an error rate of at most ε: The number of error columns in ε-matches divided by the number of alignment columns has to be less than or equal to ε (see Fig. 3.1). In this form, distance measures are informative also for local alignments. ε-matches just like edit distance alignments use the simpler linear and not an affine gap model. However, in the case of low error rates the difference between the models is minimal since only few consecutive gap columns may occur.

Score drop-offs. To supplement similarity scores or normalized distance measures, objective functions often specify maximal score drop-offs for any part of the local alignments, so called X-drops. An *X-drop* is a low-scoring section of an alignment that has a score less than $-X$ [169] (see Fig. 3.2). An alignment section has a low-score if it contains many error columns, hence, the

probability that the corresponding sequence segments are homologous is low. Therefore, the aim is to identify local alignments without X-drop, where X denotes the maximally allowed score drop-off.

E-values. Finally, e-values [81] have to be mentioned in the context of objective functions for local alignments. An *e-value* describes the expected number of local alignments that reach a given minimal similarity score in random sequences. E-values are universally accepted as probabilistic measure for local alignment significance. They indicate the spuriousness of local alignments. E-values depend on the sequence length and on the scoring scheme used for computing the similarity score. Unfortunately, the calculation of e-values is very complex, and to the author of this thesis no local alignment approach that uses e-values as optimizing function is known. Many local alignment tools, among them `BLAST` [6], compute e-values a posteriori.

3.1.2 Efficient local alignment approaches

The strategies for finding local alignments are diverse. The most noted algorithm was introduced by Smith and Waterman in 1981 [151]. It uses dynamic programming (DP) [33] to compute the best local alignment according to a similarity scoring function. The DP approach remains a standard for short input sequences until today because it guarantees to find the optimal local alignment. For long sequences, its quadratic running time complexity is too slow and more efficient approaches became prevalent. *Seed-and-extend* as well as *filter-and-verify* (see Fig. 3.3 and text below) are efficient strategies that are fast also on long sequences.

Both of these two-step approaches use short exact matches to speed up the computation of local alignments. Already in 1983, short exact matches called *k-tuple matches* were used for alignment of biological sequences [166]. Meanwhile, the names *seed* or *q-hit* are used more frequently. Especially the term 'seed' has brought along a range of possible definitions. Seeds in their original definition consist of a fixed number of consecutive match columns formed by the alignment of two identical sequence segments. The segments are called q*-grams* if they have a length of q (or sometimes k*-mers* or l*-tuples*). To mention only two other seed definitions, there are spaced seeds and adaptive seeds. *Spaced seeds* are predefined patterns of match columns and mismatch or match columns. The definition of *adaptive seeds* leads to varying seed lengths that depend on the occurrence frequency of the matching segments in the sequences [34, 92].

Seed-and-extend. Methods for computing local alignments that apply a seed-and-extend approach initially search for all seeds of a predefined kind. To realize this search efficiently, the methods index at least one of the input sequences. The index depends on the type of seeds and can for example be a q-gram index, a suffix array [104], or a FM index [56]. To increase sensitivity and specificity for local alignments, elaborate approaches use additional criteria such as two smaller seeds in close proximity [7]. In the second phase, the seeds are extended to local alignments. Usually, a DP extension is initiated at both ends of the seeds and continued until an X-drop is reached [170]. We obtain local alignments that are bounded by X-drops and contain at least one seed.

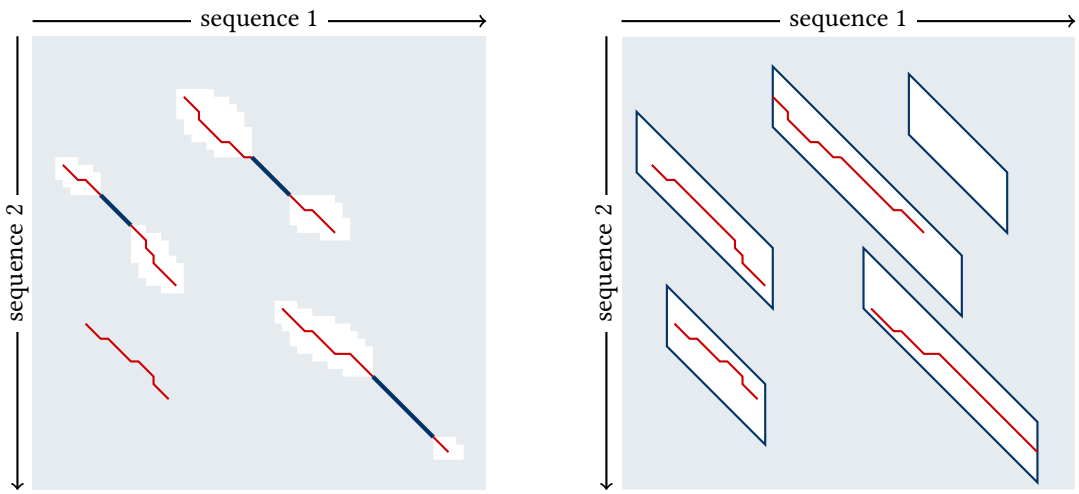

Figure 3.3: Efficient local alignment approaches in the dotplot of two sequences with four local alignments (red). The seed-and-extend approach (left) searches for short exact matches (dark blue) and extends them to both sides (white areas). The local alignment in the bottom left corner does not contain an exact match of the necessary size and will not be identified with a seed-and-extend approach. The filter-and-verify approach (right) reports potentially related areas (white parallelograms) of the dotplot. Some reported areas, such as the one in the top right corner, do not contain local alignments.

Seed-and-extend approaches are heuristics. They identify a local alignment if and only if it contains the specified seed. Seed-and-extend approaches often identify short and insignificant local alignments with high e-values, but sometimes miss significant local alignments with very low e-values. Insignificant alignments are usually filtered with an e-value cutoff before reporting local alignments. The number of missed alignments can only be reduced by using shorter or spaced seeds.

Filter-and-verify. The idea of filter-and-verify approaches is to discard large portions of the dotplot before computing nucleotide-level alignments. Filtering sorts unrelated regions of the dotplot from regions that are potentially related based on observations about the number and distribution of match and error columns in local alignments. Filtering lemmata, for example the q-gram lemma [80] (see also Section 3.2.2 or Lemma 1 on page 37) the pigeonhole principle [113], formalize these observations and provide the minimal number of seeds present in a potentially related region of a defined length. Potentially related regions are subject to verification in the second phase. Here, any local alignment algorithm can be used in principle. We can, for example, use a seed-and-extend method [134] or a DP algorithm [150, 164]. There is the same trade-off between running time and sensitivity for verification as for alignment in general: Heuristic approaches are generally faster but less sensitive than exact approaches.

Filtering can be an alternative to seeding in terms of running time, but requires carefully chosen parameters. In comparison to seed-and-extend approaches, filter-and-verify approaches generally use shorter seeds. The length and number of seeds influence the running time and the

filtering sensitivity. The filtering lemma gives advice on how to set the length of seeds. The usage of sufficiently short seeds guarantees full sensitivity but may trigger a lot of verifications due to random seed matches. If the seeds are longer than the lemmata allow, the approach may miss local alignments and becomes heuristic.

3.1.3 Overview of pairwise local alignment tools

The following section aims at giving an overview of the most relevant tools. It focuses only on tools for computing pairwise local alignments, although tools for the computation of semi-global alignment and multiple local alignment including repeat finding are closely related them. Even for computing pairwise local alignments, the list of tools raises no claim to completeness. Figure 3.4 displays a publication timeline of the listed tools.

The tool SSEARCH from the FASTA package [127] implements the dynamic programming algorithm by Smith and Waterman, which exactly computes similarity scores [151]. FASTA itself is a heuristic local alignment tool [128], which searches for seeds in close proximity and chains them to local alignments. The idea of chaining seeds to local alignments later has been taken up again by the program CHAOS [25].

BLAST [6] is the first in a family of tools that implement the seed-and-extend approach. The original BLAST implementation used only a single seed and X-drop extension without gaps. Its replacement Gapped BLAST [7] extends two shorter seeds and allows gaps. Further developments resulted in the tool MegaBLAST [170]. The tool PatternHunter [99, 102] achieves a sensitivity improvement over earlier seed-and-extend implementations by using spaced seeds while still being fast. In comparison, the focus of the tool BLAT is on efficiency rather than on sensitivity. BLAT sacrifices sensitivity for both speed and memory consumption by indexing only non-overlapping q-grams. It has been developed to align short sequences, for example expressed sequence tags (ESTs) or short reads, to a reference sequence. Here, the loss of sensitivity is smaller due to high similarity between the aligned sequences. Another implementation of the often refined seed-and-extend approach with optimized parameter settings resulted in the tool BLASTZ [148, 149], which has later been replaced by LASTZ [76]. A new seed model and a multiple hit criterion for extension was added by the tool YASS [117]. Seeds in YASS allow positions to match a subset of characters, for example to allow transitions but not transversions. Finally, the recent tool LAST [92] adds adaptive seeds to the seed-and-extend approach. Thereby, LAST can better cope with repeat regions in the sequences. Extension is triggered only for seeds that are long enough to occur less frequently than a predefined threshold. This includes very short seeds if they occur rarely enough.

In addition to the prevalent seed-and-extend approaches, a number of other tools are in use. A smaller family of tools searches for MUMs using suffix trees: Subsequently to MUMmer [40], the tools PROmer for protein-coding DNA and NUCmer for multiple-contig alignment [41] have been developed. Further improvements led to the MUMmer 3 system [94]. The approach implemented in the more recent tool BWT-SW uses the Burrows-Wheeler transformation (BWT) of a suffix array. BWT-SW computes alignments according to a similarity scoring function just like the Smith-Waterman algorithm. Hence, it is an exact alignment tool, which is fast enough for comparing

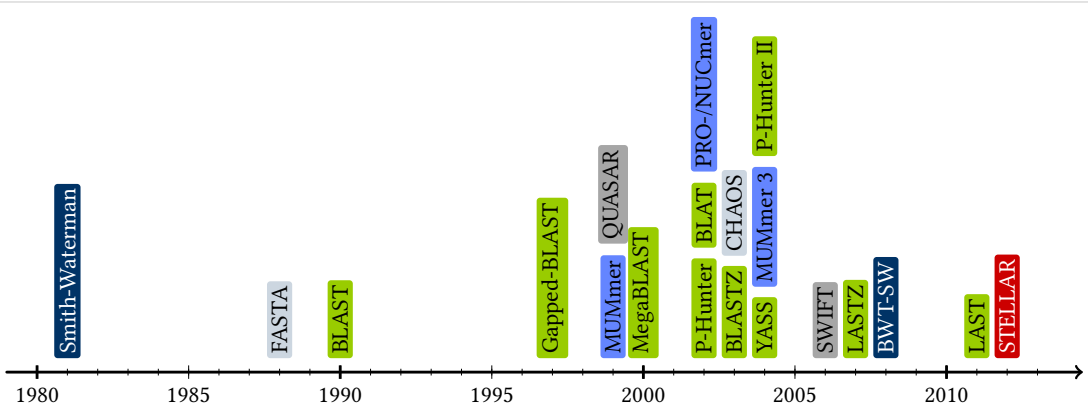

Figure 3.4: Timeline of pairwise local alignment tools. Colors roughly indicate families of the implemented approaches: Tools that implement the seed-and-extend approach are shown in green, filtering tools in gray, MUMs-based tools in middle blue, exact Smith-Waterman alignment methods in dark blue, and chaining tools in light blue. Note that this list of tools does not claim to be comprehensive.

genome-size sequences. Similarly, the filtering approaches in the programs QUASAR [27] and SWIFT [134] are very fast for long sequences while having full sensitivity for ε-matches. The following part of this chapter describes these filtering approaches in more detail and presents an exact verification strategy for SWIFT.

3.2 STELLAR: a lossless filter-and-verify approach

The remainder of this chapter treats a lossless filter-and-verify approach to local alignment, which has been implemented in the SwifT Exact LocaL AligneR (STELLAR). It applies the SWIFT filtering algorithm [134], a q-hit counting method that hugely reduces the search space. Founded on a detailed analysis of the nature of SWIFT hits and ε-matches, STELLAR complements the SWIFT algorithm with an exact verification phase.

This part of the chapter starts with a precise definition of the objective of STELLAR. The second section reviews the filtering algorithm SWIFT followed by a section that describes properties of SWIFT hits and ε-matches including necessary proofs. These properties allow the development of the verification strategy in the fourth section. Finally, the last section of this part provides information about the implementation and availability of STELLAR.

3.2.1 Objective of STELLAR

STELLAR is lossless with respect to a well-defined quality definition of local alignments:

Maximal ε-matches of minimal length n_0 without ε-X-drop.

Figure 3.5: A low scoring section (green) in an ε-match with an error rate of 0.125 (dark blue). The green section is an ε-X-drop if $X = 3$ with $p = -\frac{1}{0.125} + 1 = -7$ and $-pX = 21$. The three match columns are not enough to compensate for one of the four error columns, giving a score drop-off of 25. The light blue boxes show alternative ε-matches that do not span the ε-X-drops.

Figure 3.6: Two ε-matches that share alignment columns. If we require a minimal length of 20 for ε-matches with a maximal error rate $\varepsilon = 0.1$, the intersection and also the union of the blue and green ε-matches are *not* ε-matches themselves.

The basis of this quality definition is a normalized distance measure, the maximal error rate ε. The ε-matches are required to have a minimal length n_0 and are not allowed to span ε-adjusted score drop-offs, which we call *ε-X-drops*. Maximality of ε-matches as defined below avoids redundancy in the final set of local alignments.

ε-X-drops. ε-X-drops are score drop-offs adjusted to the error rate of an ε-match. Using a scoring scheme that scores matches with $m = +1$ and errors with $p = -\frac{1}{\varepsilon} + 1$, an ε-X-drop allows a maximal score drop-off of $-pX$. Then, the value X roughly corresponds to the maximal number of mismatches within a low-scoring region of an ε-match (see Fig. 3.5).

Maximality. Maximality is a criterion that involves more than one local alignment. We here define the term *maximal* to describe local alignments that are longer than any local alignment that shares some alignment columns or have, with respect to all such longer local alignments, a unique portion of length at least n_0 (see Fig. 3.6).

Assume there is an ε-match with an error rate much smaller than ε. Then, shortening this ε-match at one end by one alignment column yields another valid ε-match. In this case, the shorter ε-match provides no additional similarity information. Only if the unique portions of the two ε-match are both large, both ε-match are informative. In Fig. 3.6 only the green ε-match is maximal since it is longer than the dark blue match. The unique portion of the dark blue ε-match has a length of 3, which is shorter than the required minimal length.

3.2.2 Filtering with SWIFT

The filtering algorithm SWIFT identifies areas of the dotplot that intersect with ε-matches of length n_0 or longer. Given the error rate ε, SWIFT looks for local alignments with at most $\lfloor \varepsilon n \rfloor$ errors where $n \geq n_0$ is the length of the local alignment. The errors can be arbitrarily distributed over the length of the ε-match. For each ε-match, the algorithm guarantees to report at least one hit.

The SWIFT algorithm belongs to a class of algorithms that are founded on an observation on the number of q-hits in ε-matches [80]:

Lemma 1 (q-gram lemma). *The minimal number of q-hits in an ε-match of length n is*

$$T(n, q, \varepsilon) = n + 1 - q \cdot (\lfloor \varepsilon n \rfloor + 1) \ .$$

This lemma is proven by the simple fact that every introduction of an error in an exact match of length n eliminates at most q of the $n - q + 1$ q-hits. This fact directly leads to the minimum number of q-hits provided in Lemma 1.

Given the minimal number of q-hits in ε-matches, filter algorithms can search for areas in the dotplot with sufficiently high numbers of q-hits. The closest precursor of SWIFT, the program QUASAR [27], inspects areas that have the shape of a rectangle. SWIFT further improves the filtering by using *parallelograms*, more precisely intersections of consecutive diagonals and consecutive columns (see Fig. 3.7). Parallelograms improve filtering specificity as they are closer in shape to the tracebacks of alignments and hence do not count hits outside the diagonal band.

For describing the general approach of SWIFT, we imagine the dotplot of two sequences S_1 and S_2 to be populated with overlapping parallelograms of a fixed size. Each parallelogram is associated with a counter for q-hits. First, the algorithm builds an index of all q-grams of S_2. Next, it scans

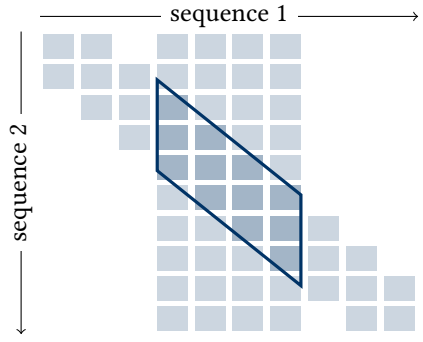

Figure 3.7: A SWIFT hit has the shape of a parallelogram. It is the intersection of consecutive diagonals with consecutive columns in the dotplot. In this example, three diagonals intersect with four columns.

over S_1 inspecting each q-gram of S_1 one by one. Any time a q-gram from S_1 is found in the index of S_2, the algorithm increments all counters that are associated with a parallelogram containing this q-hit. If the counter of a parallelogram reaches a predefined threshold, the parallelogram is reported as a hit.

The q-hit threshold for reporting SWIFT hits and the size of parallelograms affect the sensitivity and specificity of the filtering. An effective filtering algorithm has full sensitivity and aims at high specificity by reporting only as few and as small hits as possible. SWIFT sets the parallelogram size and q-hit threshold to values that guarantee full sensitivity and keep the specificity as high as possible. In the following, we first summarize the theory that SWIFT applies to guarantee full sensitivity and then describe compromises that affect specificity in favor of space and running time efficiency.

Guarantee for full sensitivity. The q-gram lemma already makes some restrictions for the parameter settings of fully sensitive filtering: Given a fixed value for ε, the q-hit length is upper bounded by $q < \frac{1}{\varepsilon}$, or, given a fixed value for q, the error rate is upper bounded by $\varepsilon < \frac{1}{q}$ (see appendix A.1). For other parameter combinations there may exist ε-matches without q-hits.

For finding all ε-matches of length exactly n_0, the function $T(n_0, q, \varepsilon)$ tells us the minimum number of q-hits in a parallelogram of the following size: The length w of the parallelogram (number of consecutive columns) is n_0, and the width e (number of consecutive diagonals) is the maximal number of errors $\lfloor \varepsilon n \rfloor$. However, the aim of the SWIFT filter is to identify all ε-matches of length n_0 or longer.

Obviously, it is not possible to include any ε-match of arbitrary length $n \geq n_0$ in parallelograms of a fixed length. The idea of the SWIFT algorithm is to look for parallelograms that intersect with the tracebacks of ε-matches. For this task, we cannot use the same parameters as above without risking a loss in sensitivity. One reason is that $T(n, q, \varepsilon)$ is not monotonically increasing (see Fig. 3.8, left). To cope with this, SWIFT defines the length n_1 as the next larger length of an ε-match that allows for one more error than the length n_0. n_1 is a function of n_0 that jumps to a higher level at each point $\{\lceil i/\varepsilon \rceil\}_{i \in \mathbb{N}}$ (see Fig. 3.8, right). The following lemma from the SWIFT paper provides bounds for all necessary parameters that ensure full sensitivity:

Lemma 2. *For a given ε and n_0, there exist q, τ, w, and e such that at least τ q-hits of any ε-match of length $n \geq n_0$ reside in a parallelogram of length w and width e. Further, choose $q < \left\lceil \frac{1}{\varepsilon} \right\rceil$ and use the following parameter settings:*

- $\tau \leq \min\{T(n_0, q, \varepsilon), T(n_1, q, \varepsilon)\}$ *where* $n_1 = \left\lceil \frac{\lfloor \varepsilon n_0 \rfloor + 1}{\varepsilon} \right\rceil$
- $e = \left\lfloor \frac{2(\tau-1)+q-1}{1/\varepsilon - q} \right\rfloor$
- $w = \tau - 1 + q(e+1)$.

We omit the proof of this lemma and refer to the original paper [134]. The lemma only guarantees to find parallelograms that intersect with the tracebacks of ε-matches. The parallelograms do not necessarily contain the ε-matches, and also do the ε-matches not necessarily span the full length

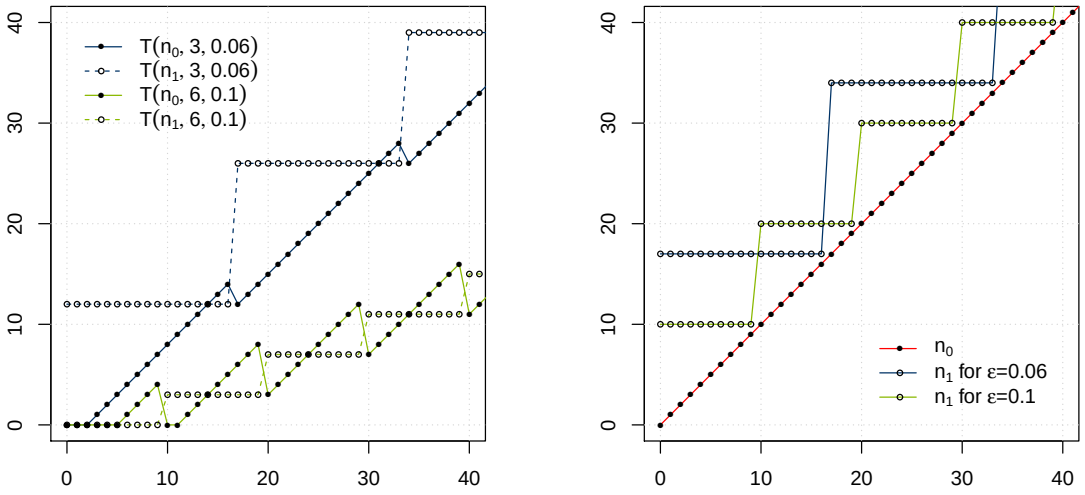

Figure 3.8: The functions $T(n, q, \varepsilon)$ and n_1 for values of n_0 between 0 and 40. n_1 is the next larger length of an ε-match with one error more than an ε-match of length n_0. The parameters q and ε are fixed to $q = 3$ and $\varepsilon = 0.06$ (dark blue) or to $q = 6$ and $\varepsilon = 0.1$ (green). The minima of the sawtooth function $T(n_0, q, \varepsilon)$ co-occur at the same points as the jumps in n_1.

and width of the parallelograms. Section 3.2.3 further investigates possible configurations of ε-matches and parallelograms.

Trading specificity for efficiency. The SWIFT algorithm scans two sequences for parallelograms that fulfill the above stated criteria for the q-gram length, and q-hit threshold. To run more efficiently in practice, the algorithm further relaxes the search to parallelograms larger than the length w and width e specified in Lemma 2, at the expense of specificity.

The number of q-hit counters is critical for memory consumption. Instead of inspecting all possible parallelograms of width e, SWIFT can reduce the search to a set of parallelograms that overlap only by e diagonals if the parallelogram width is set to $e + \Delta$ where $\Delta > 0$. This requires fewer q-hit counters and, thus, less memory. The higher Δ is, the fewer counters are necessary but the lower the specificity.

The algorithm makes a similar specificity trade-off for the length of parallelograms in order to improve running time. Instead of inspecting fixed-length parallelograms, the algorithm keeps only one counter per diagonal bucket of the dotplot. In addition, it tracks the positions of the first and last q-hits per diagonal bucket. If for a bucket the last q-hit occurred more than $w-q$ positions earlier, the algorithm checks for a new SWIFT-hit: If the counter has reached the threshold, and if the first and last q-hit are at least $q+\tau-1$ positions apart, it reports the parallelogram between the first and last q-hit as a SWIFT hit. This parallelogram can be arbitrarily long. After reporting the hit, SWIFT resets the counter as well as the first and last q-hit positions and continues counting. This way, SWIFT gains speed by inspecting counters only when a q-hit falls in the respective buckets and when reaching the end of S_2.

3.2.3 Analysis of SWIFT hits

The verification phase in STELLAR has to separate true positive SWIFT hits from false positive SWIFT hits. As a consequence of SWIFT identifying parallelograms that only intersect with the tracebacks of ε-matches, true positive hits have various possible configurations for ε-matches and parallelograms. This section inspects these configurations in more detail, after briefly looking at reasons for false positive hits.

One reason for false positive hits are the efficiency improvements of the SWIFT algorithm. But also independent from these improvements, the algorithm sometimes reports SWIFT hits that do not overlap with ε-matches. Fig. 3.9 (A-C) shows q-hit configurations that lead to false positive SWIFT hits. In addition, combinations of the shown cases lead to false positive SWIFT hits.

For true positive SWIFT hits, the verification phase in STELLAR identifies the exact tracebacks of ε-matches. The most favorable SWIFT hits start and end at the start and end positions of ε-matches. Unfortunately, this is not the case for all true positive hits. Often, ε-matches only partially overlap with reported parallelograms. The tracebacks of ε-matches in the dotplot can be longer, wider, or shorter than the parallelograms. Also can two ε-matches reside in the same parallelogram, or one ε-match trigger several SWIFT hits. Figure 3.9 (D-I) illustrates possible types of true positive SWIFT hits. Again, combinations of several types can occur.

The verification strategy in STELLAR targets the intersections of SWIFT hits and ε-matches. The idea for verification is to define minimum requirements for this intersection. At the same time, it is advantageous to be able to identify all intersections in one piece, also those intersections that exceed the minimum requirements. Based on this motivation, we introduce *ε-cores* as follows:

Definition 1 (ε-core). *Let n_0 be the minimal length of an ε-match and n_1 be defined as in Lemma 2. Under the simple scoring scheme with a match score $m = +1$ and an error penalty $p = -\frac{1}{\varepsilon} + 1$, an* ε-core *is an alignment with a score of at least*

$$s^{min} := \min_{n \in \{n_0, n_1\}} \left\lceil \frac{n - \lfloor \varepsilon n \rfloor}{\lfloor \varepsilon n \rfloor + 1} \right\rceil .$$

The following two lemmata ensure the presence of ε-cores in the intersection of ε-matches and SWIFT hits for a certain value of the filtering parameter q.

Lemma 3. *Every ε-match contains at least one ε-core.*

Proof. Following the definition of an ε-core, a q-gram of length $q := s^{min}$ is an ε-core. The next two paragraphs prove that at least one such q-gram is present in every ε-match.

The maximum number of errors in an ε-match of length n is $\lfloor \varepsilon n \rfloor$. Consequently, the number of matching positions in the same ε-match is at least $n - \lfloor \varepsilon n \rfloor$. The $\lfloor \varepsilon n \rfloor$ errors divide the $n - \lfloor \varepsilon n \rfloor$ matching positions into at most $\lfloor \varepsilon n \rfloor + 1$ error-free segments. We obtain the minimum length for the longest error-free segment of the ε-match if we distribute the $n - \lfloor \varepsilon n \rfloor$ matching positions evenly between the $\lfloor \varepsilon n \rfloor + 1$ error-free segments. An even distribution results in error-free segments that have a length of $\lfloor l(n, \varepsilon) \rfloor$ and $\lceil l(n, \varepsilon) \rceil$ where $l(n, \varepsilon) := \frac{n - \lfloor \varepsilon n \rfloor}{\lfloor \varepsilon n \rfloor + 1}$. Any other

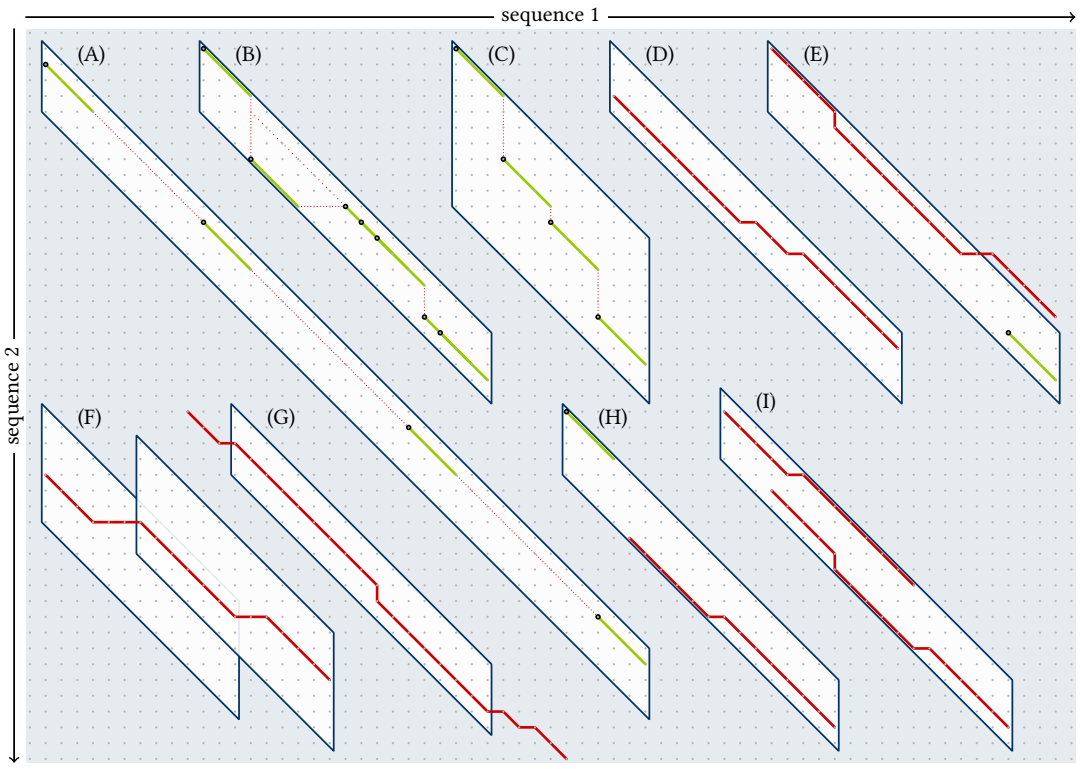

Figure 3.9: Types of false positive (A-C) and true positive (D-I) SWIFT hits and the corresponding q-hit configurations (green) of intersecting ε-matches (red). The figure shows examples for a maximal error rate $\varepsilon = 0.2$, a minimal length $n_0 = 12$, and $q = 3$. Following Lemma 2, $n_1 = 15$, $\tau = 4$, $e = 4$, and $w = 18$.
(A) A SWIFT hit where the τ q-hits required for reporting a SWIFT hit are spread over a long parallelogram with distances between the q-hits just short enough not to exceed the distance cutoff $w - q$. Arbitrary lengths of parallelograms are possible because of the running time improvement in the SWIFT algorithm. (B) q-hits on alternating sides of a parallelogram trigger a false positive hit within a parallelogram of width e. Any match that chains the q-hits has an error rate that exceed ε. (C) The τ q-hits are spread over more than e diagonals as a consequence of using parallelograms of width $\Delta + e$.
(D) A SWIFT hit that spans the same columns of the dotplot as the ε-match. (E) An ε-matches that extends beyond the diagonal border of the parallelogram. Here, an additional q-hit occurs in another parallelogram. (F) If there are sufficient q-hits, the other parallelogram triggers a second SWIFT hit. (G) Error positions close to the ends of ε-matches lead to a shorter SWIFT hit. (H) A random q-hit within a distance of $w-q$ to the ε-match extends the parallelogram beyond the length of the ε-match. (I) Two ε-matches that lie in close proximity lead to only one SWIFT hit.

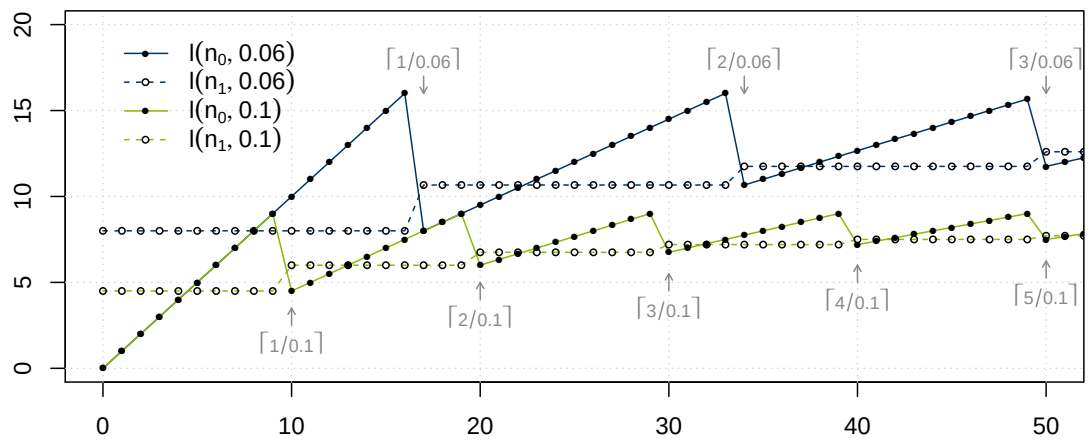

Figure 3.10: The minimal score of an ε-core is the minimum of the functions $\lceil l(n_0, \varepsilon) \rceil$ and $\lceil l(n_1, \varepsilon) \rceil$. The figure shows them for $\varepsilon = 0.06$ (dark blue) and $\varepsilon = 0.1$ (green) for n_0 in a range between 0 and 50. n_1 is calculated as defined in Lemma 2. $l(n_0, \varepsilon)$ drops to a minimum at all points $\{\lceil i/\varepsilon \rceil\}_{i \in \mathbb{N}}$ labeled in gray.

distribution results in at least one longer error-free segment. Thus, an ε-match of length exactly n contains at least one $\lceil l(n, \varepsilon) \rceil$-gram.

Unfortunately, an $\lceil l(n_0, \varepsilon) \rceil$-gram is not an ε-core for all ε-matches of length $n \geq n_0$ because $l(n_0, \varepsilon)$ is not monotonically increasing (see Fig. 3.10). However, the smallest value of $\lceil l(n, \varepsilon) \rceil$ over all $n \geq n_0$ defines the minimum length of an error-free segment. The sawtooth function $l(n, \varepsilon)$ drops to a minimum at each point $\{\lceil i/\varepsilon \rceil\}_{i \in \mathbb{N}}$, the points where n_1 jumps to its next higher value. It is easy to confirm that the minima are monotonically increasing, i. e. each successive minimum of the sawtooth is at least as high as the previous one. Therefore, every ε-match contains at least one q-gram with $q := \min\{\lceil l(n_0, \varepsilon) \rceil, \lceil l(n_1, \varepsilon) \rceil\}$, which is exactly the definition of s^{min}.

□

Lemma 4. *The intersection of a SWIFT hit with an ε-match contains at least one ε-core if $q := s^{min}$.*

Proof. With $q = s^{min}$, each q-hit is an ε-core. However, not every q-hit triggers a SWIFT hit. According to Lemma 2, there are at least $T(n_0, q, \varepsilon)$ q-hits in the intersection of SWIFT hits and ε-matches. Thus, given the requirement of the SWIFT algorithm that $T(n_0, q, \varepsilon) \geq 1$, Lemma 4 trivially follows from Lemma 3. □

With Lemma 4, the presence of ε-cores in the intersections of ε-matches and SWIFT hits is settled. Accordingly, ε-cores are suitable starting points for verification guaranteeing full sensitivity. In addition, Lemma 4 regulates how to set the parameter q in the filtering phase. This, of course, reduces flexibility in the parameter choice, but we expect users to generally welcome such advice.

3.2.4 An exact verification strategy

The lossless verification strategy for SWIFT hits identifies all maximal ε-matches without ε-X-drop on the basis of ε-cores. The strategy has five steps: It begins with the identification of all ε-cores in SWIFT hits. Each ε-core will serve as a seed for a potential ε-match. The second step filters out ε-X-drops from ε-cores to ensure that the resulting ε-matches do not span ε-X-drops. Next, the ε-cores are extended to both sides in a third step. The extension breaks off when hitting an ε-X-drop but not when hitting the borders of SWIFT hits. If the initial ε-core is part of an ε-match, then the extended ε-core spans the ε-match for any configuration of SWIFT hit and ε-match. That is why a fourth step can identify maximal ε-matches in extended ε-cores. Some ε-matches contain more than one ε-core, and therefore the first four steps identify them multiple times (once per ε-core). Similarly, not all of the identified ε-matches are maximal but share large parts with longer ε-matches. The last step filters for maximal ε-matches and ensures that they occur only once in the output. The result of this verification strategy is a set of tracebacks of all maximal ε-matches without ε-X-drops. The following addresses details of all five steps and ends with a theorem stating the exactness of the strategy.

Step 1: ε-core identification. The first step identifies all ε-cores in SWIFT hits with a banded version of a standard local alignment algorithm, the Waterman-Eggert algorithm [160]. The Waterman-Eggert algorithm uses dynamic programming (DP) to compute all non-intersecting local alignments that reach a given minimal score under a linear or affine scoring scheme. The output is a set of local alignments including suboptimal alignments. It extends the Smith-Waterman algorithm [151], which determines only the highest scoring local alignment.

According to their definition, ε-cores are alignments with a minimal score of at least s^{min} under a particular linear scoring scheme. Using this scoring scheme and s^{min} as the minimal score, the Waterman-Eggert algorithm is able to compute all ε-cores in SWIFT hits. In fact, the original Waterman-Eggert algorithm computes a quadratic DP matrix and, hence, requires quadratic running time. For the verification of SWIFT hits we benefit from the parallelogram shape and can reduce running time by computing only a band of the dotplot. As opposed to general banded alignment, the band width is not taken from a sequence distance [58] but is predefined by the parallelogram. Using the parallelogram width as a band reduces the running time by a linear factor.

The scoring parameters $m = +1$ and $p = -\frac{1}{\varepsilon} + 1$ from the definition of ε-cores and the minimal score s^{min} guarantee that the Waterman-Eggert algorithm identifies at least one ε-core for each maximal ε-match, though not necessarily for ε-matches that are not maximal: Because the algorithm computes only non-intersecting local alignments, longer ε-cores sometimes hide shorter ε-cores. The parameters m and p are chosen such that longer ε-cores intersect with shorter ε-cores only if the error rate of the non-intersecting parts have themselves an error rate of at most ε. Then, any ε-match that contains only a shorter ε-core, is not maximal. Only the longer ε-cores belong to maximal ε-matches. Figure 3.11 illustrates the case where an ε-core of a non-maximal ε-match is hidden by a longer ε-core of a maximal ε-match.

Figure 3.11: A non-maximal ε-match (dark blue) and its ε-core (green) with $\varepsilon = 0.1$ and $n_0 = 20$, hence $s^{min} = 6$, $m = +1$, and $p = -9$. The non-maximal ε-match can be extended by one error column and nine match columns to a longer ε-match (indicated by dashed line) without exceeding the maximal error rate. The Waterman-Eggert algorithm only identifies the ε-core of the maximal ε-match (green and light green).

Step 2: ε-X-drop filter. We use ε-cores as seeds for ε-matches without ε-X-drops. Hence, ε-cores should not span ε-X-drops. However, the number of errors in ε-cores and the length of ε-cores has no absolute upper bound and the distribution of errors within ε-cores is arbitrary Thus, ε-X-drops can occur in ε-cores.

In order to eliminate ε-X-drops from ε-cores, we subdivide ε-cores at ε-X-drops into multiple fragments. Since the fragments can be part of distinct maximal ε-matches, we treat them as separate ε-cores in the next steps.

The post-processing algorithm by Zhang et al. [169] detects X-drops in linear time. It builds a tree data structure that represents X-drops of an alignment for any X. In order to identify ε-X-drops, we use the match score and error penalty from definition 1 and the weighted score drop-off value $X \cdot \left(\frac{1}{\varepsilon} - 1\right)$. The general idea is to create a hierarchy of segments with increasing score drop-offs. Segments with larger score drop-offs are the union of several segments with smaller score drop-offs. The hierarchy is represented as a tree and computed from bottom to top. For the verification of `SWIFT` hits, we are interested only in score drop-offs that exceed one particular value. Thus, we can interrupt the algorithm once the hierarchy reaches this value instead of building the entire tree.

Step 3: ε-core extension. All fragments of ε-cores from the previous step serve as seeds for ε-matches. This next step extends ε-cores such that extended ε-cores contain possible, maximal ε-matches. Because the number of errors in the possible ε-matches is unknown, it is impossible to predict the necessary length of an extension that spans an ε-match. ε-X-drops offer a way out and provide a criterion for terminating extension without knowing the length and number of errors.

We extend ε-cores with the gapped X-drop algorithm by Zhang et al. [170] and use the same ε-adjusted parameters as before. This DP algorithm computes the score of a maximal alignment extension under the given scoring scheme. Different to other alignment algorithms it computes only entries of the DP matrix that can be reached without X-drop. The algorithm proceeds antidiagonal-wise and stores only the current antidiagonal of the DP matrix and an overall maximal score in order to detect X-drops. In addition, it can track the lowest and uppermost diagonals, which will be useful for computing tracebacks afterwards. The output of this algorithm are then bounds for the lower and upper diagonal of a traceback and the score and end position of a maximal extension.

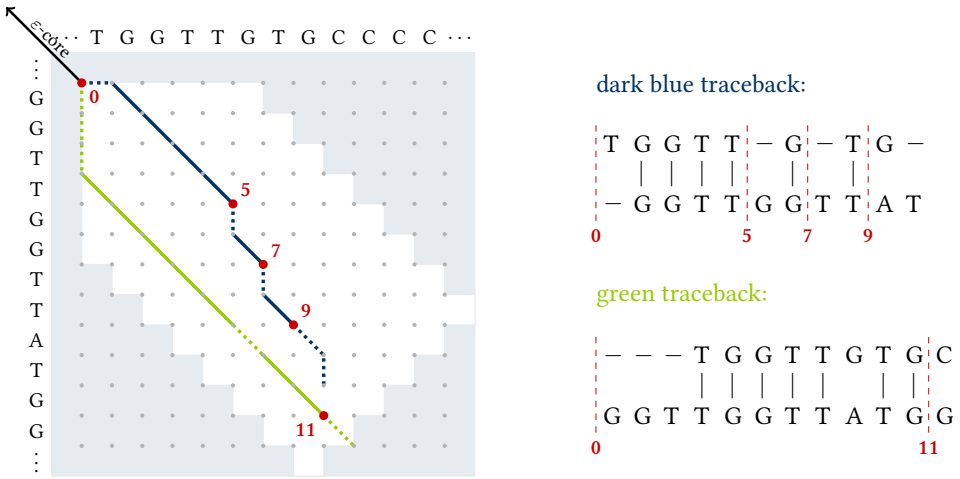

Figure 3.12: Optimal tracebacks of the right extension of an ε-core. The optimal traceback depends on the length of the extension. The dark blue traceback has a smaller error rate for a length of up to nine. For length eleven, the green traceback has a smaller error rate.

Generally, we extend all ε-cores at both ends. An exception are sets of ε-cores that originate from the same ε-core before division in the ε-X-drop filter step. In such sets of ε-cores, it is sufficient to extend only the first ε-core to the left and the last ε-core to the right. ε-X-drop extension at the previously detected ε-X-drops would immediately break off and is clearly not necessary.

Step 4: Maximal ε-match identification. Having extended ε-cores such that the extension spans possible ε-matches, the remaining task is to determine tracebacks of maximal ε-matches in extended ε-cores. More precisely, within an extended ε-core, we search for all maximal ε-matches that contain the corresponding ε-core.

We can immediately discard extended ε-cores that are shorter than n_0. For the remaining extended ε-cores we make the following two observations: First, it is not possible to determine the lengths of the left and the right extensions of an ε-core independently; the error rate allowed in the right extension depends on the error rate of the left extension and vice versa. Second, the optimal traceback in the dotplot sometimes depends on the length of the extension (see Fig. 3.12).

Based on these observations, we divide the identification of maximal ε-matches into three smaller steps that operate on dotplots of the extensions. The first step computes for any possible length of an extension the optimal end position of an ε-match. We obtain two sets of possible end positions, one for each side of the ε-core. The next step determines the length of a maximal ε-match by taking into account all pairs of possible end positions from the two sets. Finally, the last step starts computing a traceback from the optimal end positions.

Step 4a: Possible extension end positions. For a fixed length, the optimal traceback of an extension has the smallest possible number of errors. We compute the end positions of optimal tracebacks for any possible length and store them in a list. Each list entry consists of the length

Algorithm 1: Combine end positions to longest ε-match.

Input : *left*[], *right*[], *core*, ε, n_0. `// Lists of optimal end positions,`
`// the ε-core, error rate, minimal length.`
Output: *leftEnd*, *rightEnd*. `// End positions of the longest ε-match.`

```
1  minimumLength ← n_0
2  for l ← size(left) − 1 down to 0 do
3      for r ← size(right) − 1 down to 0 do
4          length ← length(left[l]) + length(core) + length(right[r])
5          errors ← errors(left[l]) + errors(core) + errors(right[r])
6          if length < minimumLength then
7              break                          // Continue with l = l − 1.
8          end
9          if errors/length ≤ ε then
10             leftEnd ← coordinate(left[l])
11             rightEnd ← coordinate(right[r])
12             minimumLength ← length
13             break                          // Continue with l = l − 1.
14         end
15     end
16 end
17 return leftEnd, rightEnd
```

of the extension, the score (or number of errors[1]), and coordinates of the corresponding entry in the dotplot. This computation is still independent for the left and right extension of an ε-core resulting in two lists.

We fill the lists during dynamic programming, when re-computing the alignment scores of the extensions. Now that we know the maximal extension lengths, we store all entries of the DP matrix. More precisely, we compute and store all entries that lie in the band defined by the upper and lower diagonal provided by the ε-core extension algorithm. When computing a score entry, we also check and if necessary update the list of optimal end positions. If the new score is higher than the score of the respective entry in the list, we update the list entry.

Unfortunately, tracebacks of different lengths may end in the same entry of the DP matrix. Only one of these tracebacks is considered by the standard DP alignment. To determine optimal end positions for all extension lengths it is necessary to add another dimension to the DP matrix for all possible lengths similar to algorithms that compute normalized alignment scores [10, 105].

After computing the full lists of possible end positions, the lists can subsequently be reduced by some extension lengths according to the following observation: If the score of the entry for length $l + 1$ differs from the score of entry l only by +1 (the match score in our scoring scheme), then an ε-match that ends at the coordinate of entry l is not maximal. Therefore, we can reduce the list by the entry of length l.

[1] We can calculate the number of errors e given the length L and score S as $e = \frac{S - mL}{p - m}$ where m is the match score and p is the error penalty of a linear scoring scheme.

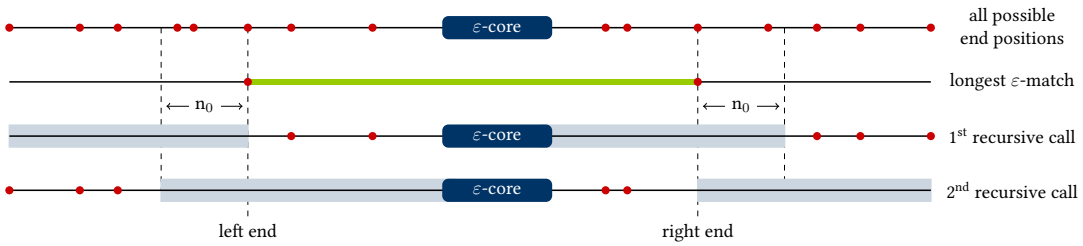

Figure 3.13: Two recursive calls of Algorithm 1 can identify further maximal ε-matches that contain the same ε-core (dark blue). From the lists of possible end positions (red dots), we may exclude several entries (light blue) after having identified a first maximal ε-match (green).

Step 4b: Maximal extension lengths. As a result of step 4a we obtain two list of possible traceback starting positions, one for the left extension and another for the right extension. All maximal ε-matches that contain the corresponding ε-core are combinations of one position from the left extension with one position from the right extension. We can find one such combination with an exhaustive search algorithm. Fig. A.1 in the appendix demonstrates that a simple greedy approach may fail.

The pseudocode for the exhaustive search algorithm is given as pseudocode in Algorithm 1. The functions `length`(), `errors`(), and `coordinate`() return the length, the number of errors, and the end coordinates of the extension for one entry in the lists of possible end positions *left*[] and *right*[], or the length and the number of errors of an ε-core, respectively. The function `size`() returns the number of entries in a list. Starting with the longest possible extensions, the algorithm calculates the error rate of the extended ε-core and, if the error rate is below ε, updates the left and right end positions of the longest ε-match. If the error rate is too large, it continues with the next shorter right position, otherwise it continues with the next shorter left extension and the longest right extension. In addition, the algorithm checks the length of each combination and continues with other combinations as soon as the length falls below n_0 or below the length of a previously found ε-match (*minimumLength*).

For the case that several maximal ε-matches are present, Algorithm 1 needs to be called recursively. We reduce the lists of possible end positions for two recursive calls until none of the combinations of a left and right end position is an ε-match. The first recursive call uses a list for the left extension without end positions left of the previously identified *leftEnd* and without *leftEnd*, and a list for the right extension without end positions left of *rightEnd*$+n_0$. The second recursive call uses a list for the left extension without end positions right of *leftEnd*$-n_0$ and a list for the right extension without end positions right of *rightEnd* and without *rightEnd* (see Fig. 3.13).

Step 4c: Traceback. Having identified the end positions of the longest ε-matches, we can use a standard traceback procedure to obtain alignments for the extensions. These alignments appended to the ends of the ε-core form an ε-match.

Step 5: Removal of non-maximal ε-matches. Often, the first verification step identifies several ε-cores of one ε-match that are all extended. Step 4 identifies the longest ε-match that contains a specific ε-core. However, the longest ε-match is not necessarily maximal; there may be a longer ε-match that shares alignment columns and contains another ε-core. In addition, two ε-cores can result in the same ε-match. Similarly, we obtain duplicate ε-matches if there are several SWIFT hits for one ε-match. To ensure that the verification output consists only of unique and maximal ε-matches, this last step is necessary.

Pairwise alignments intersect if the same positions from one sequence are assigned to the same positions in the second sequence. We sort the set of ε-matches according to their begin positions in one sequence. In the sorted set we can compare all alignment columns of two consecutive ε-matches. If two ε-matches are found to be identical or if the shorter one has no unique part of length n_0, this ε-match is not maximal and we remove it from the output.

Theorem 1. *The algorithm that uses SWIFT for filtering and the five steps for verification detects exactly all maximal ε-matches without ε-X-drop of two sequences.*

Proof. Let $\mathcal{M}$ be the set of maximal ε-matches without ε-X-drop of two sequences. For each ε-match $m \in \mathcal{M}$, there is at least one SWIFT hit in the output of the SWIFT filter algorithm that intersects with m because of Lemma 2. The first verification step identifies all ε-cores $c \in C'$ in SWIFT hits. Let C be the set of ε-cores that are present in the intersections of SWIFT hits with all $m \in \mathcal{M}$ according to Lemma 4. Because of false positive SWIFT hits, $C \subseteq C'$. If an identified ε-core $c \in C'$ is also in C, then the ε-X-drop extension in step 3 spans the corresponding ε-match, since all $m \in \mathcal{M}$ do not contain ε-X-drops. Thus, the set of extended ε-cores spans all $m \in \mathcal{M}$. Step 4 extracts ε-matches from extended ε-cores such that there are no longer ε-matches that contain the ε-core, and Step 5 removes non-maximal ε-matches. Since we do not loose maximal ε-matches in these last two steps, the output of the algorithm is exactly $\mathcal{M}$. □

3.2.5 Implementation and Availability

The program STELLAR implements the filtering with SWIFT and all five verification steps using the SeqAn C++ Library [48, 64]. The code is freely accessible as one of the SeqAn core applications at `http://svn.seqan.de/seqan/trunk/core/apps/stellar`. Binaries for the command line tool are available from the SeqAn project webpage at `http://www.seqan.de/projects/stellar` together with a detailed usage description. Being a SeqAn tool, STELLAR is also available as a KNIME node [19].

The following paragraph gives an overview of how parameter selection and all steps of the method are implemented. See Section A.2 in the appendix for an outline of the organization and main functions in the source code.

The two main parameters of the tool STELLAR are the maximal error rate and minimal length of ε-matches. In the implementation, all other parameters are calculated or are preset with default values. Still, it is possible to specify many of them as optional parameters on the command line.

STELLAR calculates the filter parameters e, w, and q-hit threshold as defined in Lemma 2, follows Lemma 4 by setting q to s^{min}, and sets Δ to the smallest power of two that is larger than e as recommended in the SWIFT paper [134]. The q-hit threshold and Δ have minimum values, i. e. STELLAR sets the q-hit threshold to at least 1 and Δ to at least 16. s^{min}, the match score m, and the error penalty p used in many steps of the verification are set according to Definition 1. The implementation differs from the described verification strategy only in step 4a where the implementation does not iterate over all lengths to compute possible end positions. Thereby we risk a loss in sensitivity, which is, however, not observable in the tests (see Section 3.3.4).

3.3 Performance evaluation of STELLAR

While the previous part provided a theoretical proof of STELLAR's exactness, we now address its practical performance. We test a wide range of parameter settings of STELLAR on bacterial genomes and study the influence of the parameter choice on the performance. In addition, we compare STELLAR to a number of established local alignment tools on simulated data sets with random sequences and on a real data set from fly genomes.

The first section of this part provides details on the setup of the parameter study and the second section describes the results of the parameter study. The last two sections provide details about the comparison of local alignment tools: the evaluation setup and the results on simulated and fly data.

3.3.1 Systematic parameter testing – Setup

The local alignment approach implemented in STELLAR has three main parameters: the error rate ε, the minimal length n_0, and the maximal score drop off X. In addition, the SWIFT filter algorithm depends on the parameter q. In STELLAR, q is automatically set to the value s^{min} calculated from ε and n_0, but the implementation allows the user to manually change its value. The next two sections examine the influence of ε, n_0, and q on the performance of STELLAR on two bacterial genomes. The values of these parameters not only affect how much of the input genomes is covered by the output set of ε-matches, but also have a strong influence on the running time. This section describes the data set, the parameter settings, and the evaluation method for the tests.

Data set. The data set in the parameter tests consists of the two bacterial genomes of *Escherichia coli* str. K-12 substr. MG1655 (NC 000913) and *Shigella boydii* Sb227 (NC 007613). The two genomes have about the same lengths with 4.7 Mbp in the *E. coli* genome and almost 4.6 Mbp in the *S. boydii* genome.

Parameter settings. We examine ranges of parameter values for the minimal length, the maximal error rate, and the q-gram length separately.

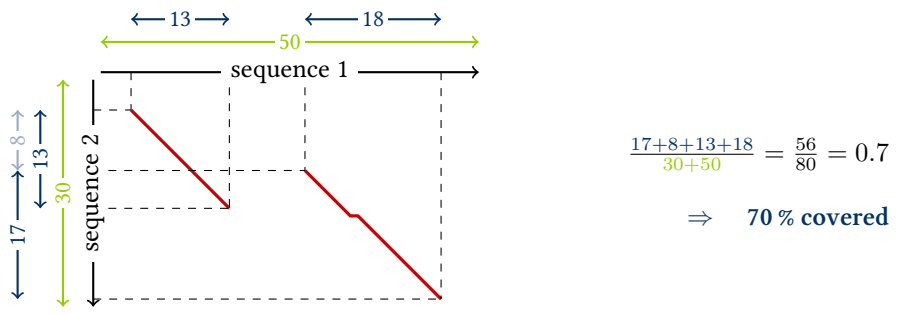

Figure 3.14: Percentage of sequence covered by local alignments. The figures shows a dotplot of two sequences with two local alignments projected onto the sequences. One alignment covers 13 positions in both sequences and the other alignment covers 18 positions in sequence 1 and 17 positions in sequence 2. Since the two alignments overlap in sequence 2 by 5 positions, they cover in total 56 positions of the two sequences. The coverage, in this example 70 %, is the number of characters that are aligned in at least one local alignment multiplied by 100 and divided by the sum of the two sequence's length.

For evaluating the influence of the minimal length, we ran STELLAR with values of n_0 between 30 and 1000 in steps of 10. We repeated this test for the five error rate values 0.001, 0.025, 0.05, 0.075, and 0.1 with the default q-gram length $q = s^{min}$ that depends on ε and n_0. In addition, we repeated the tests with error rates of 0.001, 0.025, and 0.05 for a fixed q-gram length manually set to the value 15. For $\varepsilon = 0.075$ and $\varepsilon = 0.1$, the maximal possible value of q for the SWIFT algorithm is below 15.

For testing the effect of the error rate, we ran STELLAR with ε set to values between 0 and 0.1 in steps of 0.001. We repeated this test four times with the minimal length n_0 set to 50, 100, 150, and 200 using the default q-gram length $q = s^{min}$. Furthermore, we repeated the test for $q = 15$ with all four values of the minimal length n_0. Note that with $q = 15$, the maximal possible error rate is $\varepsilon = 1/15 \approx 0.066$ for the SWIFT algorithm.

For examining the influence of a user-specified q-gram length, we tested all values of q in the range from 7 to 32. We repeated the test 16 times for all combinations of the four minimal lengths 50, 100, 150, and 200 with the four error rates 0.005, 0.025, 0.05, and 0.075.

Evaluation method. For each test run of STELLAR, we measured the total running time, and the time needed for filtering. The measurements were done on a 2.66 GHz Intel Xeon X5550 with 72 GB of RAM running Linux. In addition, we report the *sequence coverage* as the percentage of sequence characters that is part of at least one ε-match in the output (see Fig. 3.14). We also report the number of SWIFT hits as one aspect of the filtering specificity.

Furthermore, we report values of the calculated parameters of SWIFT and STELLAR: The parameters e, Δ, and w determine the size of SWIFT hits, and the q-hit threshold τ determines how many q-hits are necessary to report a SWIFT hit. The minimal score of an ε-core s^{min} can affect the verification phase as well as the filtering phase if STELLAR's default settings are used, where $q = s^{min}$.

3.3.2 Systematic parameter testing – Results

As a basis for understanding the results of the parameter tests, this section starts with a paragraph that addresses expectations on the influence of the parameters. Next, it presents the results of the tests with different minimal lengths, with different maximal error rates, and finally, with different q-gram lengths.

Expectations on parameter influence. The trend of how the maximal error rate ε and the minimal length n_0 influence the percentage of covered input sequence by ε-matches is intuitively clear: Given two maximal error rates $\varepsilon_1 < \varepsilon_2$, the set of positions covered by ε_2-matches includes all positions covered by ε_1-matches. Similarly, given two minimal lengths $n_0 < m_0$, the set of ε-matches longer than n_0 includes all ε-matches longer than m_0. Thus, an increase of ε and a decrease of n_0 lead to an increase of the coverage.

For the running time, however, the first intuition is misleading. The running time of STELLAR is *not* generally growing with the sequence coverage by ε-matches. A careful analysis of the algorithmic parameters of STELLAR reveals complex dependencies. The running time consists of the time for filtering and the time for verification. The fraction for verification highly depends on the specificity of filtering, which is determined by the size and number of reported SWIFT hits. This is independent from the number and size of ε-matches. If the filtering specificity is high, the fraction of the running time for verification is low. Thus, the parameters that determine the specificity greatly influence the running time.

Both the size and the number of SWIFT hits can be associated with individual parameters of the filtering algorithm. The size of SWIFT hits depends on the parameters e, Δ, and w, and the number of SWIFT hits on the parameters q and τ. The parameters e and Δ determine the width of SWIFT hits, whereas w has an effect on the length. The value of q influences the probability for spurious q-hits, thus, the number of SWIFT hits. Finally, the q-hit threshold τ also affects the number of SWIFT hits since, for example, the probability for two spurious q-hits is much smaller than for one. Thus, we expect that low values of e, Δ, and w and high values of q and τ lead to a high filtering specificity and shorter time for verification.

All of these five filtering parameters depend on the maximal error rate ε and the minimal length n_0. Their dependency is not linear such that more and less beneficial combinations of n_0 and ε exist. The following results of the parameter tests may assist to decide which combinations are most beneficial and identifies the factors that have the strongest influence on the running time in this data set.

Dependency on the minimal length. The two figures 3.15 and 3.16 display the results of the tests for minimal lengths between 30 and 1000.

As expected, the percentage of covered sequence drops for increasing minimal lengths in all tests with five different error rates. The filtering time is constantly low with a slight increase only for $\varepsilon = 0.075$ and $\varepsilon = 0.1$ at very small values of n_0. The time for verification dominates over the filtering time across the tested range of minimal lengths. Note that the figures display the

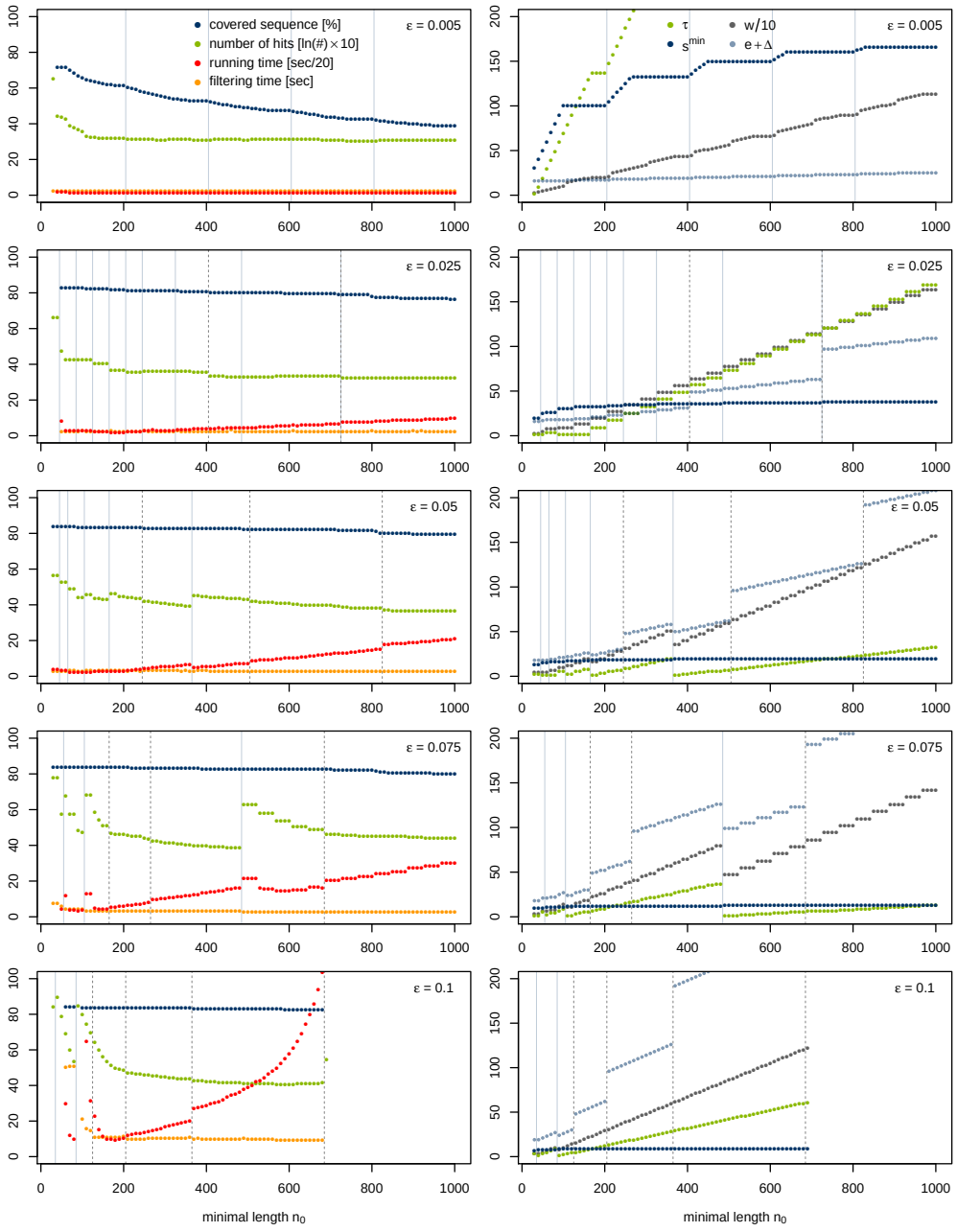

Figure 3.15: Systematic testing of the minimal length parameter of STELLAR with default $q = s^{min}$. The five rows display the results for error rates of 0.005, 0.025, 0.05, 0.075, and 0.1. The plots on the left show the measured filtering time, total running time, number of SWIFT hits, and sequence coverage by ε-matches. The plots on the right show the calculated algorithm parameters, the q-hit threshold τ, the hit width $e + \Delta$, the minimal score for ε-cores s^{min}, and the parameter w, which affects the length of SWIFT hits. Light blue vertical lines denote values of n_0 where s^{min} starts climbing to the next level. Dotted vertical lines indicate values of n_0 where Δ jumps to a higher value.

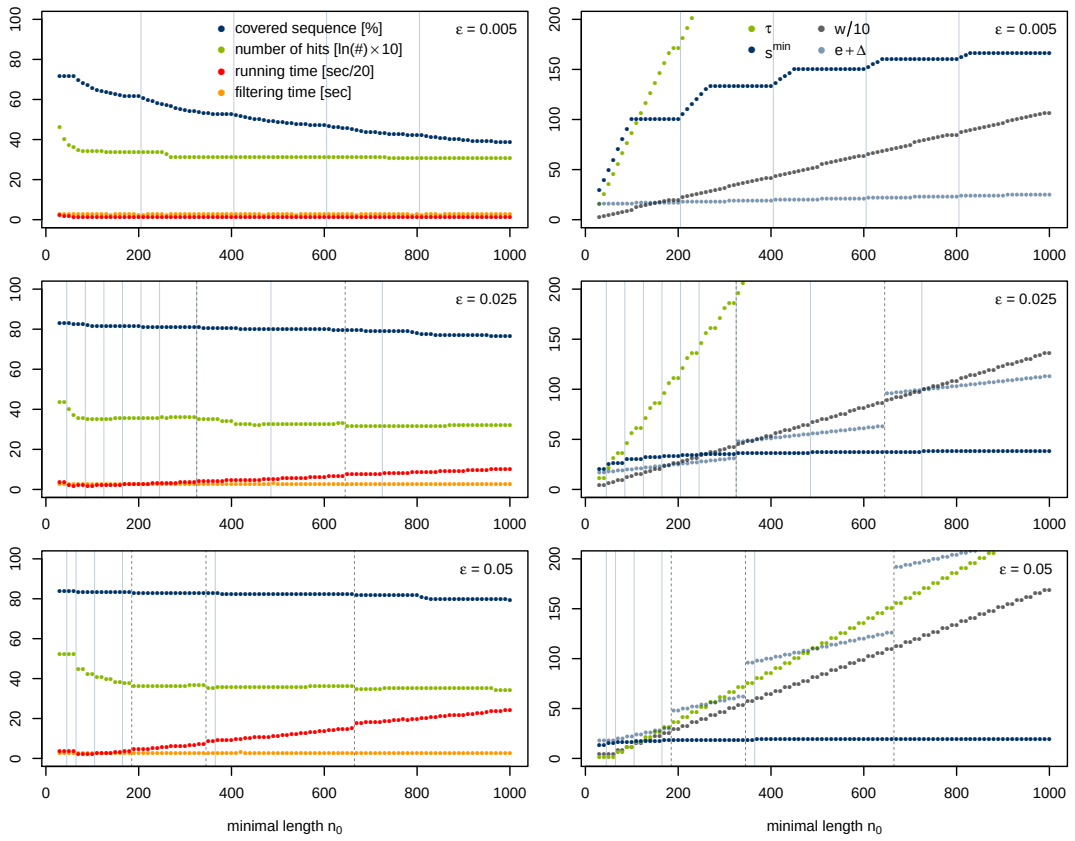

Figure 3.16: Systematic testing of the minimal length parameter of STELLAR with the q-gram length fixed to 15. The three rows display the results for error rates of 0.005, 0.025, and 0.05. STELLAR requires the q-gram length to be greater than $1/\varepsilon$, thus, for $q = 15$ only error rates up to $0.0\overline{6}$ are possible. The rows display the results for the error rates, 0.005, 0.025, and 0.05. See Fig. 3.15 for details on the displayed values and vertical lines.

filtering time in seconds, but the total running time in seconds divided by 20. The total running time shows several jumps and confirms a complex dependency on the algorithm parameters.

A subset of the jumps can be explained with the algorithm parameters that influence the number of SWIFT hits. In the tests of Fig. 3.15, both q and τ depend on s^{min}, since $q = s^{min}$ and τ depends on q. s^{min} grows with increasing n_0 but converges to a maximum. The increase is not contiguous but the value rests at several levels, e.g. for $\varepsilon = 0.005$ at $s^{min} = 100, 133, 150$, etc. It starts climbing to the next level only at minimal lengths $n_0 = \lceil i/\varepsilon \rceil$ with $i \in \mathbb{N}$. The q-hit threshold τ falls to 1 at values of n_0 where s^{min} steps to the next level. In Fig. 3.16, where q is set to a fixed value, the q-hit threshold τ increases monotonically.

The jumps of τ to 1 co-occur with jumps in the number of SWIFT hits (light blue vertical lines in Fig. 3.15). This confirms a negative correlation of the number of hits with τ. Significant increases in the number of SWIFT hits in turn affect the running time (e. g. for $\varepsilon = 0.075$ at $n_0 = 55, 108$, and 481 or for $\varepsilon = 0.1$ at $n_0 = 32$ and 81). However, the negative correlation of the running

time with τ is not visible between the jumps and also not in the tests with fixed q. Furthermore, there are jumps in the running time that do not co-occur with jumps in the number of SWIFT hits.

The remaining jumps can be explained with the algorithm parameters that determine the width of SWIFT hits. Since STELLAR sets the width to $e + \Delta$ and $\Delta \in \mathbb{N}$ to the minimal value where $2^{\Delta} > e$, the width jumps to higher levels where e exceeds powers of two. These jumps occur at the same values of n_0 as the remaining jumps in the running time (dashed vertical lines in Fig. 3.15 and 3.16) confirming an influence of the width of SWIFT hits on the verification time. In addition, the behavior of the running time between the jumps conforms with the expectations regarding the values of e and w and increases with growing e and w. Thus, the influence of the size of SWIFT hits appears to dominate over the influence of the number of SWIFT hits except for values of τ close to 1, where the number of hits is largest.

Dependency on the maximal error rate. The two figures 3.17 and 3.18 display the results of the tests for maximal error rates between 0 and 0.1. In all four tests with different minimal lengths, the percentage of covered sequence grows as expected for increasing error rates. The filtering time is again low compared to the total running time with a slight increase close to $\varepsilon = 0.1$. The two figures display the natural logarithm of the total running time to visualize the extreme variation in the verification time.

The running time variation is caused by peaks in the number of SWIFT hits. Like in the tests with different minimal lengths, the number of hits increases when τ falls to 1. With increasing error rate, s^{min} converges towards 1 in integer steps. With $q = s^{min}$ (see Fig. 3.17), τ falls to many minima in the range of ε between 0 and 0.1 leading to many peaks in the number of hits and also in the running time. The peaks become larger with growing ε, where $q = s^{min}$ decreases. At values of ε where s^{min} falls to the next smaller integer value, τ jumps to a value larger than 1. Figure 3.18 illustrates how the number of hits gets larger for values of τ close to 1. The figure also demonstrates that differences of τ at large values (e.g. $\tau > 50$) have a minor effect.

The running time depends less on the size of SWIFT hits across the tested maximal error rates than on the size of hits across the tested minimal lengths. Only in Fig. 3.18 at values of ε where $q > s^{min}$, the running time visibly grows together with the width and length determining parameters e, Δ, and w.

Dependency on the q-gram length. Figure 3.19 displays the results of the tests for q-gram lengths between 7 and 32 and Fig. 3.20 the corresponding values of the filtering parameters. In all 16 fixed combinations of ε and n_0, the coverage stays mostly constant with few decreases of less than $1\,\%$ right and left of $q = s^{min}$. We did not verify whether this indicates a loss of sensitivity or is a side effect. Lemma 4 only guarantees full sensitivity for STELLAR if $q = s^{min}$.

The filtering time is low (around 2.5 s) for most values of q. Values of q below 10 are the only tested parameter combinations where the filtering time is much higher and has a significant influence on the total running time in this data set. For all other values, the verification time dominates over the filtering time. The total running time reaches a local maximum at $q = s^{min}$

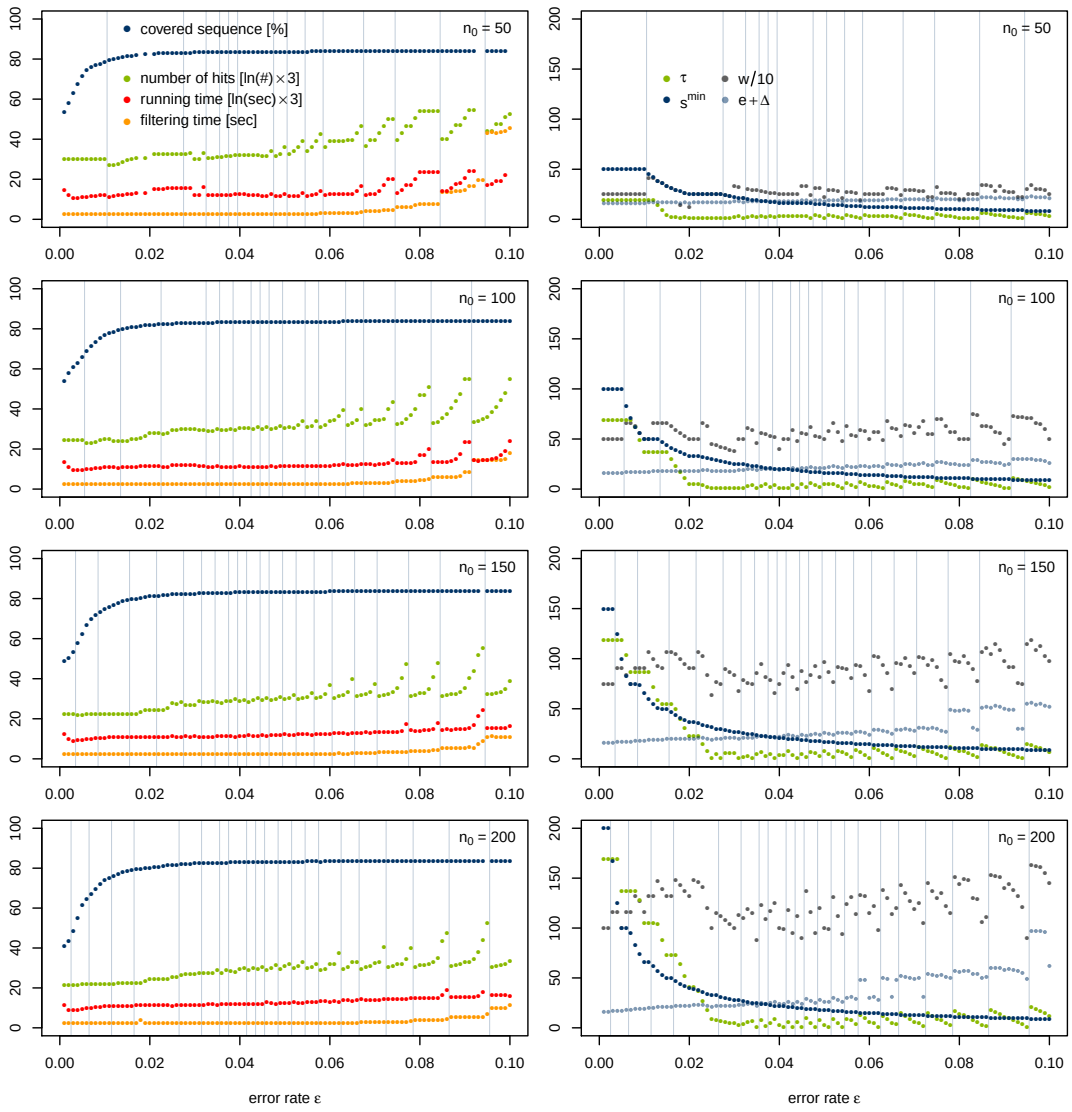

Figure 3.17: Systematic testing of the maximal error rate parameter of STELLAR with default $q = s^{min}$. The four rows display the results for different minimal lengths of 50, 100, 150, and 200. See Fig. 3.15 for details on the displayed values. Note that the natural logarithm of the running time is shown here. Vertical lines denote values of n_0 where s^{min} starts dropping to the next level.

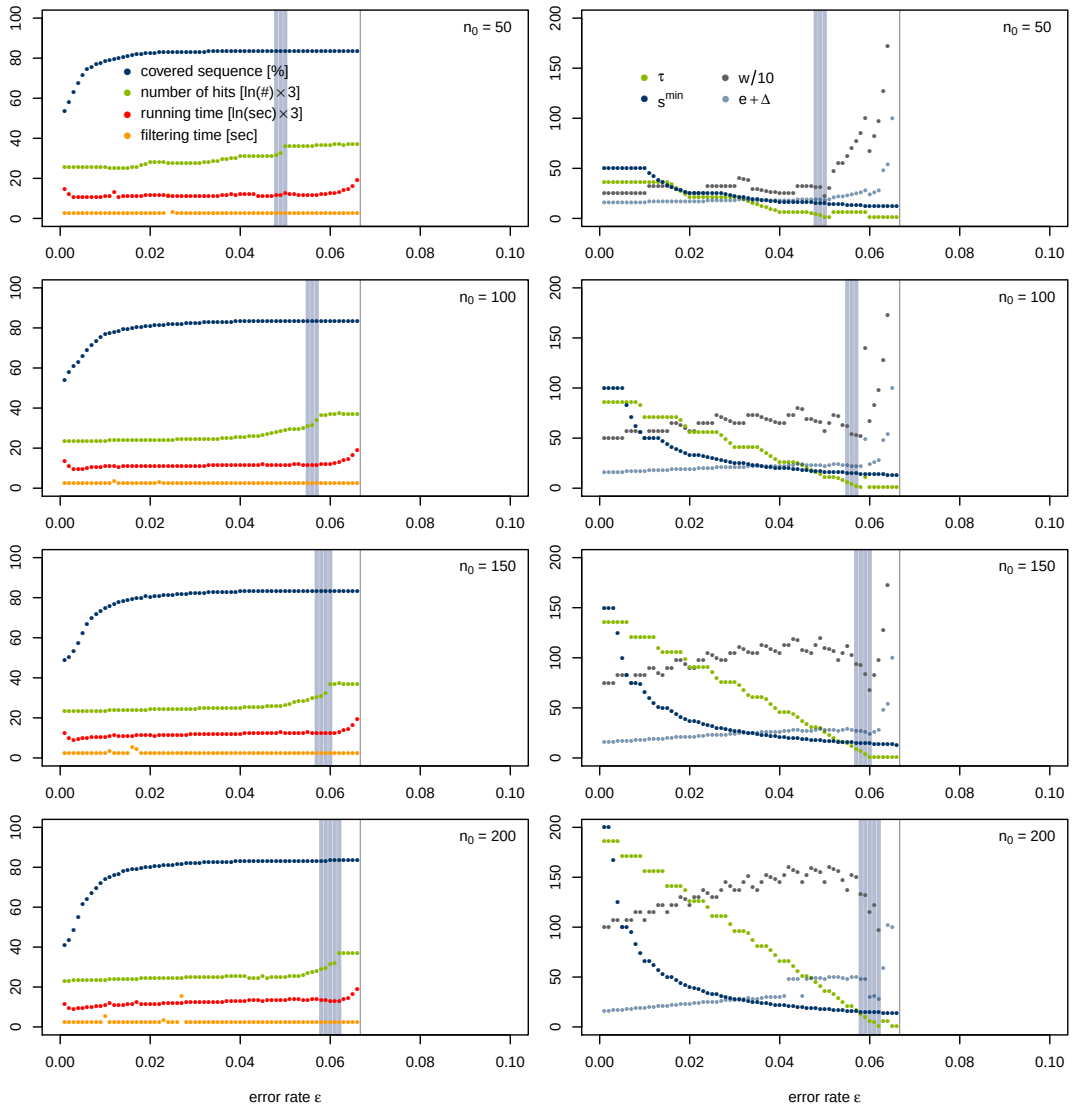

Figure 3.18: Systematic testing of the maximal error rate parameter of STELLAR with the q-gram length fixed to 15. The four rows display the results for different minimal lengths of 50, 100, 150, and 200. Note that for $q = 15$ only error rates up to $0.0\overline{6}$ are possible because STELLAR requires the q-gram length to be greater than $1/\varepsilon$. See Fig. 3.15 for details on the displayed values. Analogously to Fig. 3.17, the natural logarithm of the running time is shown here. The light blue bars denote the range of ε where $s^{min} = 15$.

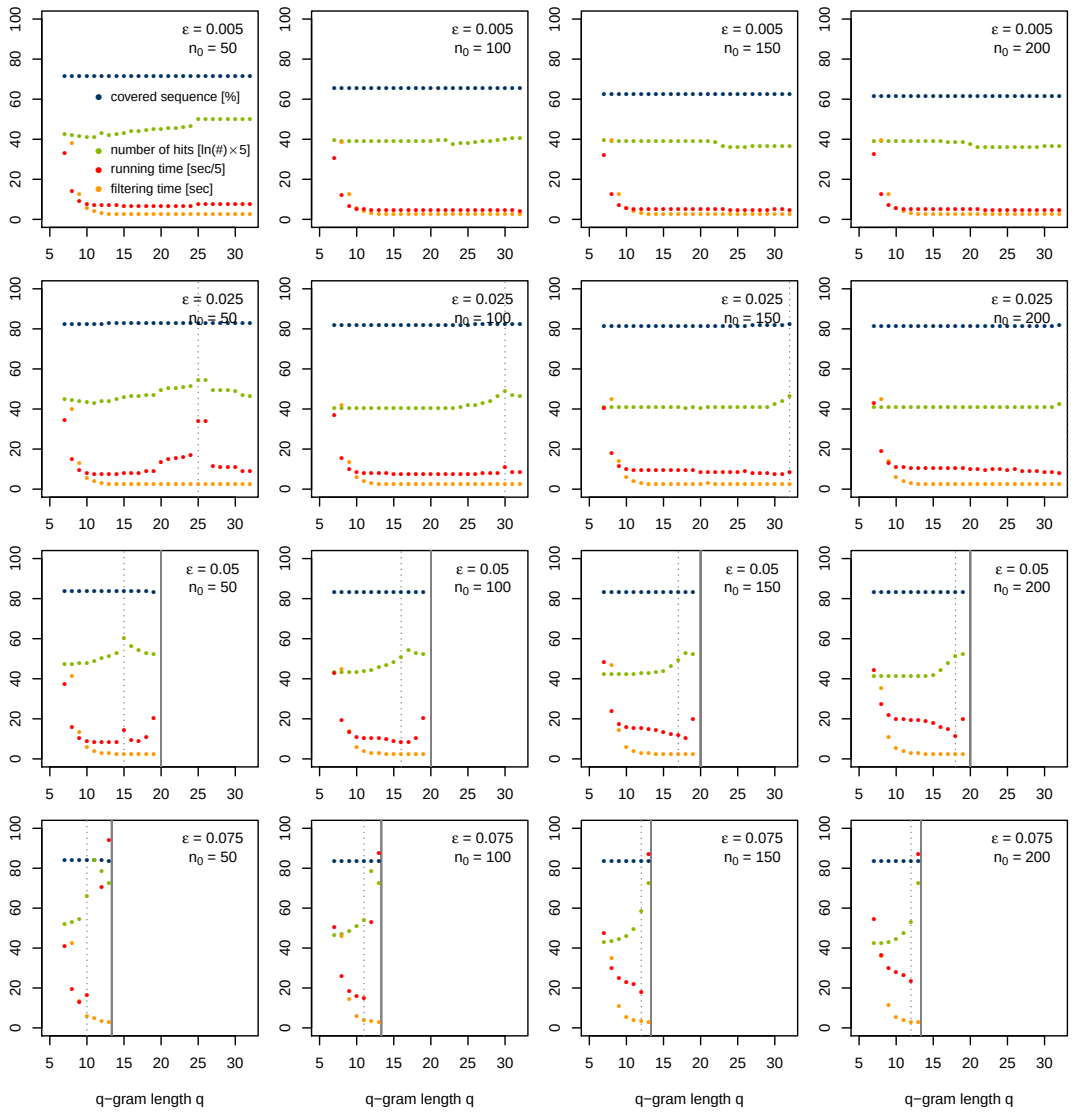

Figure 3.19: Systematic testing of the q-gram length parameter of STELLAR for 16 combinations of ε and n_0. The rows show four different error rates and the columns four different minimal lengths. Each plot displays the filtering time, the total running time, the number of SWIFT hits, and the percentage of covered sequence. The dotted vertical lines denote $q = s^{min}$, and the solid lines indicate the maximum possible value $q = 1/\varepsilon$.

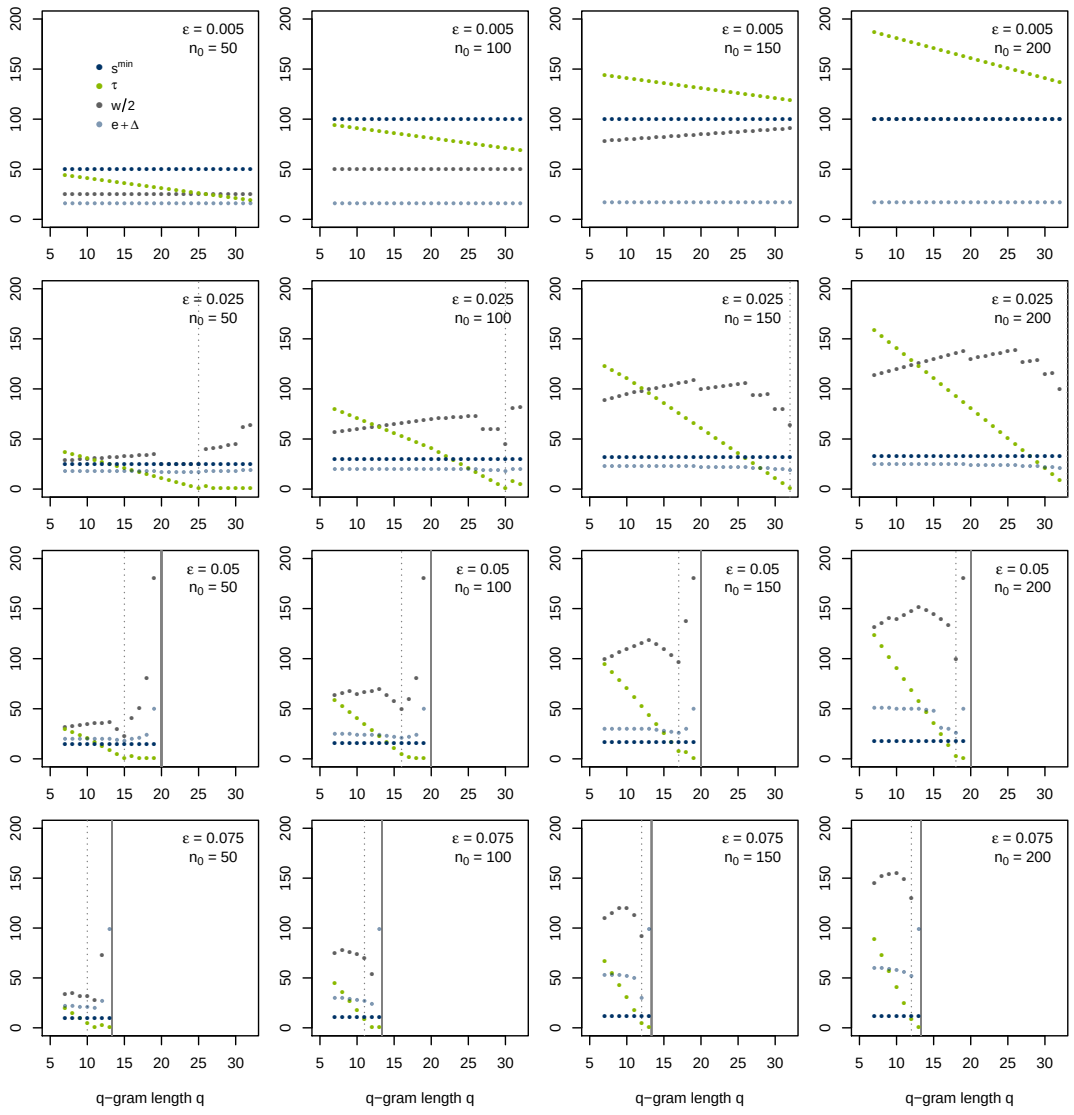

Figure 3.20: Systematic testing of the q-gram length parameter of STELLAR for 16 combinations of ε and n_0. The rows show four different error rates and the columns four different minimal lengths. Each plot displays the filtering parameters τ, $e+\Delta$, s^{min}, and w. Analogously to Fig. 3.19, the dotted vertical lines denote $q = s^{min}$, and the solid lines indicate the maximum possible value $q = 1/\varepsilon$.

for some combinations of ε and n_0 and it increases when q climbs towards $\frac{1}{\varepsilon}$. This variation in the running time is again due to the filtering specificity expressed as both the number and size of SWIFT hits.

The q-hit threshold τ drops from $q = 7$ to $q = s^{min}$ and reaches $\tau = 1$ at or shortly behind $q = s^{min}$. This correlates with the number of hits, which climbs to a maximum at $q = s^{min}$ for most parameter combinations. However, the size parameters of SWIFT hits fall to a minimum at $q = s^{min}$ for most parameter combinations and counteract the effect on the running time. For $\varepsilon = 0.025$ and for $n_0 = 50$ there is a peak in the running time at $q = s^{min}$ caused by the increase in the number of hits. For larger ε or n_0, the minimum in the hit size keeps the running time low. The increase of the running time towards $q = \frac{1}{\varepsilon}$ can be explained with a large increase in w and $e + \Delta$.

3.3.3 Comparison of local aligners – Setup

The aim of the comparison of local aligners is to investigate STELLAR's gain in sensitivity compared to seed-based methods and to compare its speed to the other full-sensitivity methods. The simulated data provides the opportunity to find the range of error rates and sequence lengths where we can benefit from using STELLAR instead of other tools. This section describes the setup of the comparison including the data sets, the evaluation method that measures sensitivity according to our objective of finding ε-matches, a list of compared tools, and tested parameter values for STELLAR.

Data sets. We compare the performance of STELLAR on two simulated data sets, *sim-err* and *sim-len*, and on one real data set from fly genomes. Simulated data is useful for systematic testing of sensitivity, but can have biases or artifacts in comparison to real data. Our simulations generated random sequences with randomly implanted ε-matches. As alternative with fewer biases one could implant ε-matches into real sequences at random positions. The advantage of random sequences is that similarities occur by chance only with a very small probability. Thus, the simulated data sets allow for examination of the dependence of the performance on one parameter independently from other factors.

The data set *sim-err* tests different error rates and the data set *sim-len* several sequence lengths. Both simulated data sets consist of pairs of random sequences with local alignments implanted at random positions. All sequences have uniformly distributed characters from the DNA alphabet. The implanted local alignments have lengths between 50 and 200 bp and contain substitutions, insertions, and deletions of single characters with uniform probabilities. The number of implanted local alignments n per sequence pair varies depending on the lengths of the sequences, with $n = \lceil \text{length}/2000 \rceil$. For example, for a pair of two 1 Mbp sequences $n = 500$.

In the data set *sim-err*, which tests different error rates, all sequences have a length of 1 Mbp. The implanted local alignments have error rates of 0 %, 2.5 %, 5 %, 7.5 %, or 10 %. For each error rate, the data set has 5 pairs of sequences, making in total 25 alignment instances with 12,500 implanted local alignments.

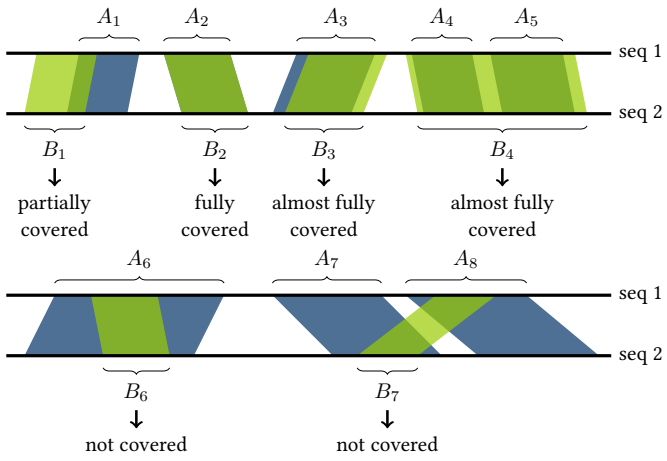

Figure 3.21: Coverage of green local alignments B_i by dark blue local alignments A_i. A green local alignment is only covered if the same positions from sequence 1 are aligned to the same positions from sequence 2 in a dark blue local alignment.

In the data set *sim-len*, the sequences have a length of 1 kbp, 10 kbp, 100 kbp, 1 Mbp, or 10 Mbp. The error rates of the implanted local alignments are uniformly distributed in the range from 0 to 10 %. To partially compensate for the different numbers of local alignments in sequences of different lengths, the data set has 50 pairs of 1 kbp sequences, 10 pairs of 10 kbp and 100 kbp sequences each, and one pair of 1 Mbp and 10 Mbp sequences each. This gives in total 72 alignment instances with 6,100 implanted local alignments.

The data set with real genomic data consists of sequences from fly genomes of the species *Drosophila melanogaster* (release 5.26) and *Drosophila pseudoobscura* (release 2.14) available from FlyBase [157]. We selected chromosome 2L from *D. melanogaster*, which has a length of ∼23.5 Mbp, and the ∼11.7 Mbp group 3 from the chromosome 4 assembly of *D. pseudoobscura* [137] for our comparison.

Evaluation method. For all three data sets, we measured the running times of `STELLAR` and the compared programs on the same Intel machine as used in the comparison of different local aligners (see Section 3.3.1), a 2.66 GHz Intel Xeon X5550 with 72 GB of RAM running Linux. For all tools we include indexing of input sequences in the running times. We omit details on memory usage as it was below 1 GB in all test runs.

To measure sensitivity, we compare computed local alignments to those that were implanted into the simulated sequences, or we compare local alignments of different programs among each other for the fly data. The search strategies of the different programs often result in slightly different local alignments. Therefore, we say that a reference local alignment (e. g. an implanted local alignment) is covered by the set of computed local alignments if at least 10 % of all positions of the reference alignment are aligned to the same positions in any of the computed alignments (see Fig. 3.21). Only below 10 % we consider the reference alignment to be missed. This is a very

general check whether significant similarities are found by a tool or not, which is in favor of the compared programs. We report the percentage of covered local alignments as the sensitivity.

Compared tools. In comparison to STELLAR, we tested the two exact programs SSEARCH [126] and BWT-SW [95] that compute Smith-Waterman alignments, and the seed-based methods implemented in the programs BLAST [6, 7], LAST [92], LASTZ [76], and BLAT [90].

The program SSEARCH is provided with the FASTA package [128] and we used version 36. BWT-SW is available from github[2]; we used version 20080713. For the BLAST search we used the NCBI blastn implementation [28] version 2.2.23+. Furthermore, we downloaded version 278 of LAST[3]. LASTZ[4] is the replacement of BLASTZ [148] and we used version 1.03.02. Finally, we used version 34 of BLAT from the UCSC Genome Bioinformatics Site[5].

We ran all programs with default settings. In addition, we ran BLAST with the word size parameter set to s^{min} and denote this with BLAST*. According to Lemma 3, this parameter setting enables full sensitivity for the BLAST method. For SSEARCH, we used an e-value cut-off of 0.01.

Parameter settings for STELLAR. For the test runs on simulated data, the minimal length parameter in STELLAR was set to 50 and the maximal error rate was set according to the error rates in the data sets. Applying the findings from the parameter study (see Section 3.3.2 and Fig. 3.19), a q-gram length of $s^{min} - 1$ was used.

On the fly data, we tested three combinations of parameters: A maximal error rate of 10 % with a minimal length of 100, a maximal error rate of 10 % with a minimal length of 200, and a maximal error rate of 5 % with a minimal length of 200. In all three combinations, the x-drop parameter was set to 20. For the runs with 10 % errors, we used a q-gram length of 8, and for the run with 5 % errors a q-gram length of 17, which is again $q = s^{min} - 1$.

3.3.4 Comparison of local aligners – Results

Simulated data. The running times and sensitivity of STELLAR and all compared tools are shown in Table 3.1 for the data set *sim-err* and in Table 3.2 for the data set *sim-len*.

Table 3.1 demonstrates that STELLAR clearly outperforms the other full sensitivity tools SSEARCH and BWT-SW in terms of running time for the tested range of error rates. In addition, STELLAR is faster than LASTZ and BLAT and in most cases LAST, which use the heuristic seed-based approach and miss significant percentages of local alignments. Only BLAST is faster than STELLAR, but it also misses considerable amounts of local alignments with larger error rates. The BLAST* run is much more sensitive than BLAST but still misses a small fraction of matches while its running time is similar to STELLAR at high error rates. Figure 3.22 displays a simulated local alignment that only STELLAR, SSEARCH, and BWT-SW identify but not the seed-based approaches.

[2] https://github.com/mruffalo/bwt-sw
[3] http://last.cbrc.jp/
[4] http://www.bx.psu.edu/~rsharris/lastz/
[5] http://genome.ucsc.edu/FAQ/FAQblat.html

Table 3.1: Running times and sensitivity on simulated sequences containing local similarities of different error rates.

error rate	0 %		2.5 %		5 %		7.5 %		10 %	
	time	missed	time	missed	time	missed	time	missed	time	missed
SSEARCH	133:17 h	0.00 %	132:55 h	0.00 %	133:24 h	0.00 %	132:25 h	0.00 %	132:52 h	0.00 %
BWT-SW	14.38 s	0.00 %	14.28 s	0.00 %	14.18 s	0.00 %	14.09 s	0.00 %	14.02 s	0.04 %
STELLAR	**0.55 s**	**0.00 %**	**0.52 s**	**0.00 %**	**0.57 s**	**0.00 %**	**0.88 s**	**0.00 %**	**3.81 s**	**0.00 %**
BLAST	0.25 s	0.00 %	0.25 s	0.16 %	0.26 s	5.36 %	0.25 s	17.00 %	0.24 s	38.80 %
BLAST*	0.24 s	0.16 %	0.25 s	0.00 %	0.27 s	0.04 %	0.49 s	0.28 %	3.56 s	2.60 %
LAST	1.84 s	0.00 %	1.85 s	0.00 %	1.85 s	1.12 %	1.86 s	4.60 %	1.86 s	10.16 %
LASTZ	6.14 s	0.00 %	6.13 s	0.72 %	5.90 s	5.56 %	5.58 s	12.68 %	5.02 s	24.92 %
BLAT	13.05 s	29.36 %	10.53 s	29.64 %	12.81 s	28.88 %	13.42 s	31.44 %	13.38 s	34.32 %

Sequences have a length of 1 Mbp and contain local alignments of lengths between 50 and 200 bp. The simulations were repeated five times, the displayed values are the average of all runs except for SSEARCH which was run only once. Sensitivity is measured by the percentage of missed local alignments (Fig. 3.21). BWT-SW, BLAST, LAST, LASTZ, and BLAT were run with default parameter settings. BLAST* stands for a more sensitive run of BLAST with the word size parameter set to the minimal length of an ε-core s^{min}.

```
          .     :     .     :     .     :     .     :     .     :
628613  AAGGATTGTCCATCTACAGCGCTTTATTTAAGCTGGGCATAGCGAACTGC
        ||||||||||||||||||  ||||||||||||||  ||||||||||||  ||
407615  AAGGATTGTCCATCTACA-CGCTTTATTTAAGCT-GGCATAGCGAAC-GC

          .     :     .     :     .     :     .     :
628663  CCGCTCCATAGACTTAAATGCTTTCCTCTAC-ATACAAGTCG
        ||||||||||||||  |||  ||||||||||||||  ||   ||  |||
407662  CCGCTCCATAGAC-TAAGTGCTTTCCTCTACGAT-TAA-TCG
```

Figure 3.22: This local alignment from the simulated data set *sim-err* was only found by STELLAR, SSEARCH, and BWT-SW, but not by the seed-based methods. It has an e-value of 7×10^{-26}.

Table 3.2 indicates that STELLAR's running time scales less well to 10 Mbp sequences than all of the other tested tools. This limitation on long sequences is due to a high number of verifications in STELLAR at as high error rates as 10 %. The filtering specificity of SWIFT is low for high error rates since the q-gram length has to be set to a small value ($s^{min} = 8$ for $\varepsilon = 0.1$ and $n_0 = 50$) in order to guarantee full sensitivity. Thus, spurious q-hits and false positive SWIFT hits become probable. The resulting large number of verifications are the reason for an increase in running time.

However, except for the 10 Mbp sequences, STELLAR is faster than most other tools on the data set *sim-len* just as observed on the data set *sim-err*. The data set *sim-len* also confirms full sensitivity for STELLAR, SSEARCH, and BWT-SW. BLAST and LASTZ show up to 14 % and 10 % of missed local alignments and also LAST misses up to 4 %. Among the seed-based approaches with default settings, LAST is the most sensitive and only slower than BLAST.

For BLAT, the test results suggest a dependency of the sensitivity on the sequence length: While it is fully sensitive on sequences of up to 10 kbp, the number of missed local alignments almost reaches 70 % on the 10 Mbp sequences. A possible explanation for this lossy performance is that

Table 3.2: Running times and sensitivity on simulated sequences of different lengths.

seq length	1 kbp		10 kbp		100 kbp		1 Mbp		10 Mbp	
	time	missed	time	missed	time	missed	time	missed	time	missed
SSEARCH	2.45 s	0.00 %	259.5 s	0.00 %	7:16 h	0.00 %	136:16 h	0.00 %	–	–
BWT-SW	–	–	67 ms	0.00 %	0.62 s	0.00 %	14.17 s	0.00 %	465.60 s	0.10 %
STELLAR	**<1 ms**	**0.00 %**	**<1 ms**	**0.00 %**	**0.06 s**	**0.00 %**	**3.76 s**	**0.00 %**	**733.16 s**	**0.00 %**
BLAST	3 ms	14.00 %	8 ms	6.00 %	0.03 s	11.40 %	0.25 s	13.40 %	3.23 s	12.64 %
BLAST*	3 ms	0.00 %	8 ms	0.00 %	0.07 s	0.00 %	3.54 s	0.60 %	471.38 s	1.46 %
LAST	<1 ms	4.00 %	10 ms	0.00 %	0.17 s	2.60 %	1.85 s	3.40 %	37.75 s	2.70 %
LASTZ	10 ms	10.00 %	57 ms	2.00 %	0.53 s	7.80 %	5.71 s	9.40 %	116.87 s	9.12 %
BLAT	22 ms	4.00 %	34 ms	0.00 %	0.36 s	1.00 %	13.98 s	33.00 %	349.22 s	69.30 %

Sequences contain $\lceil \text{length}/2000 \rceil$ local alignments with a maximal error rate of 10 % and lengths between 50 and 200 bp. The simulation of the 1 kbp sequences was repeated 50 times, the simulation of the 10 kbp and 100 kbp sequences ten times. The displayed values are the average of all runs except for SSEARCH which was run only once. Sensitivity is measured by the percentage of missed local alignments (Fig. 3.21). BWT-SW, BLAST, LAST, LASTZ, and BLAT were run with default parameter settings. BLAST* stands for a more sensitive run of BLAST with the word size parameter set to the minimal length of an ε-core s^{min}.

Table 3.3: Results of STELLAR on fly chromosomes.

error rate	min. length	running time	num. of ε-matches	overlap BLAST[1]
10 %	100	2077 s	3940	100 %
10 %	200	1300 s	341	100 %
5 %	200	18 s	44	100 %

We used chromosome arm 2L from *D. melanogaster* ($\sim$ 23.5 Mb) and group 3 of chromosome 4 from *D. pseudoobscura* ($\sim$ 11.7 Mb). STELLAR was run with the X-drop parameter set to 20.
[1] Percentage of covered ε-matches from filtered BLAST output.

BLAT was originally designed for the comparison of many short sequences (ESTs or sequencing reads) against one long reference sequence, and not for the comparison of two long sequences.

Both tables demonstrate STELLAR's gain of sensitivity in comparison to the seed-based methods, and a gain of running time efficiency in comparison to other full-sensitivity methods. The results from the data set *sim-err* (Table 3.1) suggests that we benefit most from using STELLAR when we compare closely related sequences that still have significant differences. The data set *sim-len* (Table 3.2) indicates that STELLAR is especially fast on short and moderately long sequences.

Fly chromosomes. The tests on real data compare STELLAR only to BLAST because BLAST is the standard tool for local alignment searches and, in addition, performed well on simulated data. The results are shown in tables 3.3 and 3.4.

The set of local alignments computed by BLAST contains some alignments with low e-values that are shorter than the minimal length for ε-matches or that have more errors than allowed by the maximal error rate for ε-matches. Since STELLAR does not find such alignments and also does not intend to find them, these alignments were filtered from the BLAST output. Some long filtered

Table 3.4: Results of BLAST on fly chromosomes.

word size	running time	num. of hits	overlap STELLAR 200[1]	overlap STELLAR 100[1]
default	9 s	9504	95.01 %	89.24 %
9	792 s	28434	100.00 %	99.26 %

We used chromosome arm 2L from *D. melanogaster* ($\sim$ 23.5 Mb) and group 3 of chromosome 4 from *D. pseudoobscura* ($\sim$ 11.7 Mb).
[1] Percentage of covered STELLAR matches (error rate 10 %, minimal length 200 or 100).

```
          .    :    .    :    .    :    .    :    .    :    .    :    .    :    .    :
17794720 TTTCAGTTGCAGCGAGGGTCGCATCATGTCCTCGGACACCAAGGGATGTGTGGACCGCAACGAGTGCCTGGACCTGCCAT
         |||||||||||||||||| |||||||||| |||||||||||||| ||||| || |||||||| |||||||| ||||| |
10600229 TTTCAGTTGCAGCGAGGGACGCATCATGTCGTCGGACACCAAGGGCTGTGTCGATCGCAACGAATGCCTGGATCTGCCCT

          .    :    .    :    .    :    .    :    .    :    .    :    .    :    .    :
17794640 GTCTGAACGGAGCCACCTGCATCAATCTGGAGCCCCGGCTGCGGTACCGATGCATTTGCCCGGAGGGCTACTGGGGCGAA
         | |||||||| ||||| |||||||||||||||||||| |||||||| || ||||| ||||||||||||||||||||||||
10600309 GCCTGAACGGGGCCACGTGCATCAATCTGGAGCCCCGTCTGCGGTATCGCTGCATCTGCCCGGAGGGCTACTGGGGCGAG

          .    :    .    :    .    :    .    :    .    :    .    :    .    :    .    :
17794560 AACTGCGAGCTGGTGCAGGAGGGACAGCGCCTGAAGCTGAGCATGGGCGCCCTGGGGGCCATATTCGTTTGCCTGATTAT
         ||||||||||||||| |||||||| |||||||||||||||||||||| ||||| || ||||||||||||||||||||||||
10600389 AACTGCGAGCTGGTCCAGGAGGGCCAGCGCCTGAAGCTGAGCATGGGGGCCCTCGGCGCCATATTCGTTTGCCTGATTAT

          .    :    .    :    .
17794480 CATACTGAGTAAGTA-G-AGTGATGG
         |||| |||||||||| | | || |||
10600469 CATATTGAGTAAGTACGAAATG-TGG
```

Figure 3.23: This local alignment of the fly chromosomes was found by STELLAR with $\varepsilon = 10$ % and $n_0 = 200$, but not by BLAST with default parameters. It has an e-value of 6×10^{-84}.

BLAST hits with an error rate above ε may contain shorter ε-matches, thus valid ε-matches from long filtered BLAST hits were extracted and added to the set of BLAST ε-matches. This results in a set of ε-matches identified by BLAST. The output of STELLAR was compared to this set of ε-matches (last column of Tab. 3.3).

As expected, STELLAR covers all of the BLAST ε-matches. In accordance with the findings from the simulated data and the parameter tests, the running time of STELLAR is smaller for a minimal length of 200 and again much smaller for an error rate of 5 %. However, the number of ε-matches also shows that real sequences can have less conserved similarities, which we only find when allowing high error rates and short minimal lengths.

Table 3.4 shows that BLAST with default parameters is much faster than STELLAR. At the same time, BLAST identifies a very large number of local alignments. Although all these BLAST hits have low e-values, there are 17 ε-matches with a minimal length of 200, and 424 ε-matches with a minimal length of 100 that BLAST misses if ε is set to 10 %. One of these matches is displayed in Fig. 3.23. While BLAST identifies a large number of similarities, it misses some significant local alignments. With the word size parameter set to s^{min} in order to improve the sensitivity

of BLAST, the output is an even larger number of local alignments (many of them short) but with hardly any false negatives. The running time becomes comparable to that of STELLAR. Depending on the parameter settings of STELLAR, the one or the other program is faster.

3.4 Conclusion to the chapter

The chapter concludes with a summary, a short discussion, and an outlook on how it may be possible to further improve STELLAR's running time through sophisticated parameter selection.

Summary. STELLAR is an efficient method to solve the pairwise local alignment problem with full sensitivity according to a clear quality definition for local alignments. This quality definition includes a minimal alignment length as well as the established measures of a maximal error rate ε and a maximal score drop-off. With its guarantee for full sensitivity, the method is well suited to be applied in genome alignment approaches.

The novel methodological contribution of STELLAR is an exact verification strategy for the previously published filtering algorithm SWIFT [134]. The filtering algorithm is lossless. However, its hits were verified only by heuristics until now. For STELLAR, a thorough analysis of the hits allowed the development of a fully sensitive verification strategy. The key to full sensitivity is the definition of ε-cores, which serve as starting point in the verification strategy. A theoretical proof substantiates full sensitivity according to the quality definition.

Systematic parameter tests for the two most important parameters of STELLAR, ε and n_0, as well as for the filtering parameter q indicate the existence of more and less beneficial parameter combinations. The tests suggest at first glance a general trend of the running time to increase both with growing minimal lengths and growing maximal error rates. A careful analysis of the test results, though, reveals a non-linear dependency of STELLAR's running time on the maximal error rate and minimal length, attributable to the filtering specificity.

Finally, a performance comparison to widely used and established local alignment tools confirms efficiency and full sensitivity of STELLAR. On simulated data, STELLAR is the fastest of three fully sensitive programs. Seed-based approaches can be faster but often miss a considerable amount of local alignments. A comparison against BLAST on fly genomes indicates that the performance is similar on real genomic data. The results suggest that STELLAR is an attractive alternative if full sensitivity is sought especially for moderate error rates below 10 %.

Discussion. STELLAR is an exact local alignment tool from the algorithmic point of view. From the biological point of view, it models homology like all alignment tools with an objective function and formalizes properties that we think probable for homologous segments. Identifying homologous segments with absolute certainty is impossible since the evolutionary history of extant biological sequences is unknown. Thus, STELLAR accepts the risk of missing homologous segments that have other properties than formalized by the objective function.

Among the objective functions introduced in the first part of this chapter, the most widely accepted indicator of homology is the e-value. Local alignments with an e-value below a threshold are called significant. For feasible values of the maximal error rate and minimal length parameters, alignments reported by STELLAR are generally significant with very low e-values. The implementation of STELLAR reports the maximal possible e-value for the user specified parameters.

Nevertheless, it is possible that there are significant local alignments that STELLAR does not identify: for example, a very long alignment with an error rate above ε. Seed-and-extend approaches like BLAST put no constraint on the length and error rate and, hence, may identify these alignments. At the same time, they miss some significant local alignments even if these alignments have much lower e-values than many of the reported alignments. Depending on the application, the approach of STELLAR or BLAST may be more advantageous.

Related to this, the performance evaluation described in Section 3.3.3 and 3.3.4 may seem unfair. It compares tools that apply different quality definitions for local alignments on a data set with implanted local alignments that match the quality definition used by STELLAR: ε-matches without X-drops. Implanting local alignments that match another quality definition would lead to different results. For example, the seed-and-extend approach implemented in BLAST promises to be fully sensitive for local alignments without X-drops that contain a seed of a certain length. However, BLAST uses an e-value cutoff in addition, because not all of the seed-containing alignments are significant for homology prediction. In contrast, ε-matches of a minimal length are guaranteed to have good (low) e-values. Consequently, the implanted local alignments in the simulated data sets are all significant and, hence, form a reasonable reference set for sensitivity measurements.

Outlook. A disadvantage of STELLAR is that the complex parameter dependencies make the running time unpredictable for an average user of local alignment tools. Addressing this issue, the results of the parameter tests suggest a simple change to the method: Instead of using the user specified maximal error rate and minimal length for filtering, STELLAR could achieve higher filtering specificity with a higher maximal error rate or smaller minimal length. Eventually, only the local alignments that conform with the user specified parameters will pass verification. The goal is to avoid the peaks in the running time.

A method to determine the next larger maximal error rate and next smaller minimal length that result in a better filtering specificity is left for future work. What we already observed in the parameter tests of one data set is that a combination of ε and n_0 is typically beneficial for the specificity if

- τ is as large as possible but at least greater than 1,
- s^{min} has the same or a lower value than for surrounding values of ε and n_0,
- e and w are smaller than for surrounding values of ε and n_0, and
- q has a value between about 10 and s^{min}.

It remains open if it these observations can be generalized for all genomic sequences. One task

will be to test other data sets with various degrees of similarity and different sequence lengths.

Furthermore, it remains open if a q-gram length smaller than s^{min} affects the sensitivity of STELLAR. It is possible that Lemma 4 provides the only valid value of q for full sensitivity, or maybe all $q \leq s^{min}$ are valid. A detailed analysis of the results of the parameter test for smaller q might answer this question. Values of q larger than s^{min} and manual changes in the q-hit threshold τ lead to lossy filtering, but at the same time decrease the running time. One could test whether such settings make STELLAR competitive to the seed-based approaches.

Finally, it might be worth to examine thoroughly the parameter Δ introduced in the SWIFT algorithm for saving memory. The memory consumption was never an issue in the tests, but the value of Δ has an influence on the running time. Possibly, smaller values of Δ improve the running time without exceeding reasonable amounts of memory. Ideally, Δ and the memory consumption are automatically adapted to the available amount of memory.

All in all, there is potential to improve the performance of STELLAR, but already now it provides an alternative to other local aligners especially if sensitivity is sought. As a further step, it will be interesting to test the impact of missing local alignments on the accuracy of genome alignments with a tool like STELLAR.

Chapter 4

Graph representations for genome alignments

The comparison of graph data structures described in this chapter was realized in collaboration with Kathrin Trappe, Manuel Holtgrewe, and Knut Reinert and has been submitted for publication [88]:

B. Kehr, K. Trappe, M. Holtgrewe, and K.Reinert. Graph representations for genome alignment: a comparison. *Submitted*, 2013.

The focus of this chapter is the representation of genome alignments in graph structures. Within the last decade, several graph-based genome alignment approaches have been published [121, 124, 132, 133]. These approaches build a graph from a set of local alignments and exploit the graph structure in order to remove inconsistencies from the set of local alignments, eventually yielding a valid genome alignment.

The graph structures used for genome alignment have many similarities. However, the ability of a graph structure to represent non-colinearity often depends on small differences. Unfortunately, the differences among the used graph structures and advantages over each other remain rather elusive. In addition, labels on vertices and edges are often not described in detail. In some cases, the implementation of the graph data structure even slightly deviates from the description in the publication.

This chapter examines alignment graphs [83], A-Bruijn graphs [131], Enredo graphs [124], and cactus graphs [122] in detail. The first part of the chapter provides formal definitions of the four graph structures and of labels. Differences of the structures become apparent in the second part: It compares the graph structures by describing transformations among them. The idea is that a graph structure provides at least as much information about a genome alignment as all graph structures that it can be transformed into. The third part addresses the appearance

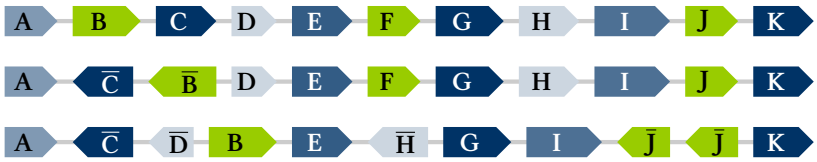

Figure 4.1: Three example genomes as sequences of eleven blocks (A-K). Blocks B, C, and D have different orders and orientations in all three genomes, and blocks G and H changed with respect to the third genome. Block F does not appear in the third genome and the reverse complement of block J appears twice. Figures 4.2, 4.3, 4.6, and 4.8 show the representation of this set of blocks as alignment graph, A-Bruijn graph, Enredo graph, and cactus graph.

of substructures in the graphs. Substructures indicate inconsistencies in the initial set of local alignments and are subject to removal in graph-based genome alignment approaches. The fourth part of the chapter describes modifications for the removal of substructures and their effects on the graphs. Finally, the conclusion in the last part of the chapter summarizes and discusses this comparison of graph data structures for genome alignments.

4.1 Definitions of selected graph representations

This part of the chapter unifies definitions of the four graph representations by assuming the same input for building the graphs and by using the same terminology, namely the terms segment, block, and adjacency (see Chapter 2). Given is always a set of non-overlapping blocks $\mathcal{B}$ defined on a set of genomes $\mathcal{G}$. We assume the blocks to be a tiling of $\mathcal{G}$. A tiling (see definition in Section 2.4) can be obtained from an arbitrary set of non-overlapping blocks by adding blocks of size one for all non-empty adjacencies.

For all four graphs, we define a *graph model* $M = (G, \ell)$ as an ordered pair of a *graph structure* $G = (V, E)$ and a *labeling function* ℓ. The function ℓ labels the vertices V or the edges E of the structure G. We define ℓ such that the set of blocks $\mathcal{B}$ can be recovered from the model M. The following sections display example structures of each graph for the three genomes given in Fig. 4.1.

Furthermore, this part of the chapter addresses the capabilities of the graph structures to model non-colinear changes among the genomes. We discuss whether duplications, translocations, and inversions are visible in the graph structures, i. e. without using labels. Some non-colinear changes are ambiguous or not visible in several graph structures, which is proven by examples. In addition, each section suggests sparse labeling functions for the respective graph that provide only as much information as necessary to model all non-colinear changes. The next part of the chapter (Part 4.2) uses the sparse labeling functions to describe transformations.

4.1.1 Alignment graphs

Alignment graphs were originally defined for colinear multiple alignments [83]. Still, they allow the modeling of non-colinear changes (see below). In Kececioglu's original definition of the alignment graph structure [83], vertices represent single sequence characters, and edges connect vertices of aligned characters. The alignment graph has since been used in various versions, e. g. with additional sequence edges [136] and with vertices representing genes [62] or segments [135] instead of single characters. We use the more general segment-based version with additional sequence edges.

Definition of graph model and structure. In the following section, let $M = (G, \ell)$ be an alignment graph model defined by an alignment graph structure $G = (V, E)$ and a labeling function ℓ for the vertices V of the structure G. The vertices of the alignment graph structure G represent segments (occurrences of the blocks $\mathcal{B}$) and are connected by two sets of edges $E = E_A \cup E_B$. Directed adjacency edges E_A connect vertices that represent adjacent segments, and undirected block edges E_B connect vertices that represent occurrences of the same block. Thus, alignment graphs are mixed graphs. See Fig. 4.2 for an example alignment graph.

Vertices. The alignment graph structure has a vertex $v \in V$ for each segment $s \in S_{\mathcal{B}}$ from the set of all occurrences $S_{\mathcal{B}} = \bigcup_{B \in \mathcal{B}} B$ of the blocks $\mathcal{B}$. The labeling function $\ell : V \to S_{\mathcal{B}}$ of the alignment graph model M associates each vertex $v \in V$ of the alignment graph structure with the corresponding segment $s \in S_{\mathcal{B}}$ such that $\ell(v) = s$.

Directed edges. The set of directed adjacency edges E_A represents adjacencies between pairs of segments $s_1, s_2 \in S_{\mathcal{B}}$ in the alignment graph structure. A directed edge $e \in E_A$ exists for every

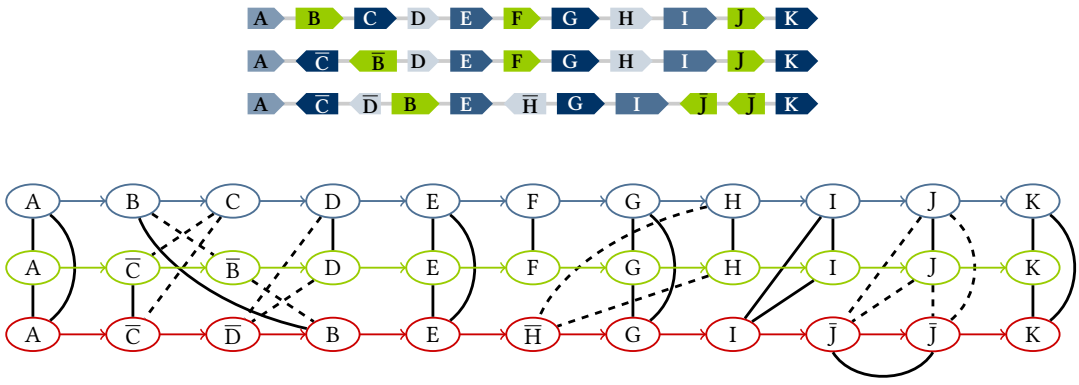

Figure 4.2: An example alignment graph (bottom) for the same set of blocks as shown in Fig. 4.1 (top). Vertices and adjacency edges are colored by genome. Block edges are shown in black. Dashed block edges connect vertices that represent segments in opposite orientation. Note that colors, line styles, and vertex labels (e.g. A or $\overline{\text{C}}$) do not belong to the structure G of an alignment graph but are labels.

two vertices $v_1, v_2 \in V$ that are labeled with adjacent segments $s_1 = \ell(v_1)$ and $s_2 = \ell(v_2)$. Let $s_1 = (i_1, l_1, o_1)$ and $s_2 = (i_2, l_2, o_2)$, then $e = (v_1, v_2)$ if $i_1 < i_2$, and $e = (v_2, v_1)$ if $i_2 < i_1$.

Undirected edges. In the alignment graph structure, there is an undirected block edge $e \in E_B$ between any two vertices $v_1, v_2 \in V$ that are labeled with two occurrences $s_1, s_2 \in B$ of the same block $B \in \mathcal{B}$. As a consequence, each block forms an E_B-connected component.

Recovering $\mathcal{B}$. By determining the set of all E_B-connected components $\mathcal{C}$, we may recover the set of blocks $\mathcal{B}$ from the alignment graph model. Given that $\mathcal{B}$ is a tiling of $\mathcal{G}$, each connected component $C \in \mathcal{C}$ corresponds to one block $B \in \mathcal{B}$. Let $V_C \subseteq V$ be the set of vertices of the component C, then $B = \{\ell(v) \mid v \in V_C\}$ is the corresponding block.

Representation of non-colinearity in the graph structure. The above definition of the alignment graph structure models duplications and translocations but no inversions. Duplications appear in G as block edges connecting vertices that represent segments of the same genome[1]. Translocations appear in G as mixed cycles. Inversions are not visible in the alignment graph structure because the orientation of segments remains unclear without labels (see also Fig. 4.2).

To model inversions in the alignment graph without using full labels, we define the sparse labeling function $\ell^{inv} : V \to \{+, -\}$. The function ℓ^{inv} associates each vertex $v \in V$ only with an orientation bit. If $\ell(v) = (i, l, o)$, the sparse label of v is $\ell^{inv}(v) = o$. As an alternative to sparse vertex labels, it is possible to model inversions by block edge labels: bits that indicate equal or opposite orientation of the two segments represented by the endpoints of the edge (visualized as dashed and solid lines in Fig. 4.2 or red and black edges in [14]).

4.1.2 A-Bruijn graphs

Pevzner et al. introduced A-Bruijn graphs originally for *de novo* repeat classification [131] as a generalization of de Bruijn graphs [32, 39]. The structure of A-Bruijn graphs revisits the idea of merging aligned vertices in the alignment graph structure, which was already briefly mentioned by Kececioglu [83].

Definition of graph model and structure. In this section, let $M = (G, \ell)$ be an A-Bruijn graph model defined by an A-Bruijn graph structure $G = (V, E)$ and a labeling function ℓ for the vertices V of the structure G. A-Bruijn graph structures have one vertex per block. All edges are directed and represent sequence adjacencies. See Fig. 4.3 for an example A-Bruijn graph.

Vertices. In the A-Bruijn graph structure, there is a vertex $v \in V$ for every block $B \in \mathcal{B}$. There is only one vertex per block regardless of the block's size and of duplications. The labeling function $\ell : V \to \mathcal{B}$ of the A-Bruijn graph model associates each vertex $v \in V$ of the A-Bruijn graph structure with the corresponding block $B \in \mathcal{B}$ such that $\ell(v) = B$.

[1] Note that G is not n-partite as in the original definition of an alignment graph [83] because of these edges.

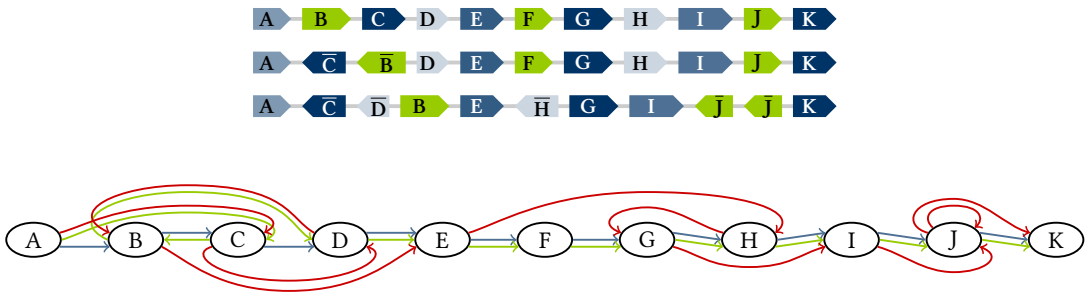

Figure 4.3: An example A-Bruijn graph (bottom) for the same set of blocks as shown in Fig. 4.1 (top). Edges are colored by genome. Note that colors and vertex labels are not present in the structure G of an A-Bruijn graph.

Figure 4.4: A section of an A-Bruijn graph structure (left) representing five blocks each of size two. The example demonstrates the ambiguity of A-Bruijn graph structures. The structure results either from duplications (top right) or from translocations (bottom right) of block B and D.

Edges. The A-Bruijn graph structure has an edge $e \in E$ for every pair of adjacent segments $s_1 \in B_1$ and $s_2 \in B_2$, where $B_1, B_2 \in \mathcal{B}$. The edge e connects the two vertices $v_1, v_2 \in V$ that represent the blocks $B_1 = \ell(v_1)$ and $B_2 = \ell(v_2)$. If two adjacent segments are occurrences of the same block $B_1 = B_2$, the edge is a loop. Let $s_1 = (i_1, l_1, o_1)$ and $s_2 = (i_2, l_2, o_2)$, then $e = (v_1, v_2)$ if $i_1 < i_2$, and $e = (v_2, v_1)$ if $i_2 < i_1$. If multiple pairs of adjacent segments exist in the same two blocks, then there are multiple edges in E that connect the same two vertices. Thus, G is a multigraph.

Recovering $\mathcal{B}$. Given the labeling function ℓ of the A-Bruijn graph model M, which labels each vertex $v \in V$ of an A-Bruijn graph structure G with a block $B \in \mathcal{B}$, recovering the set of blocks $\mathcal{B}$ from M is straightforward: The set of blocks $\mathcal{B}$ is simply the set of labels $\ell(v)$ of all vertices $v \in V$.

Representation of non-colinearity in the graph structure. The A-Bruijn graph structure models translocations and duplications although they create ambiguity. Inversions are not visible in the A-Bruijn graph structure. Fig. 4.4 shows an example A-Bruijn graph structure that represents either duplications or translocations.

The ambiguity of duplications and translocations in the A-Bruijn graph structure results from the fact that a vertex represents multiple segments (all block occurrences). Hence, a vertex can

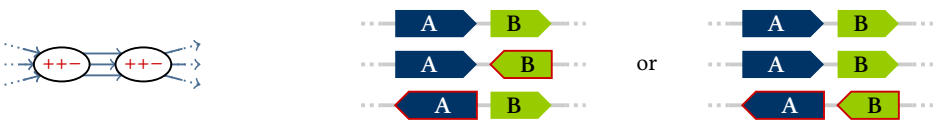

Figure 4.5: A section of an A-Bruijn graph structure representing two blocks each of size three where the vertices are labeled with sets of orientation bits (left). According to the labels, one segment per block is in the reverse complemented orientation. The labels do not resolve ambiguity since, the two reverse complemented segments can belong to two different genomes (middle) or to the same genome (right).

have multiple incoming and outgoing edges – one for each occurrence of the block. Therefore, labels that allow a mapping of incoming edges to outgoing edges resolve the ambiguity.

We define the sparse labeling function $\ell^{dup} : E \to \mathbb{N}$ as a strict total order on the edges E to resolve ambiguity. The function ℓ^{dup} assigns numbers to the edges such that $\ell^{dup}(e_1) < \ell^{dup}(e_2)$ for two edges $e_1, e_2 \in E$ if $a_1 < a_2$, where a_1 is the adjacency position represented by e_1 and a_2 is the adjacency positions represented by e_2. Note that the full labels ℓ of the graph model are necessary for determining the adjacency positions.

To model inversions in the A-Bruijn graph without providing the full labels ℓ, we define the sparse labeling function $\ell^{inv} : E \to \{+,-\} \times \{+,-\}$. The function ℓ^{inv} associates each edge E with two bits indicating the orientations of the two adjacent segments that define the edge. Then, the sparse label of the edge $e = (v_1, v_2)$ is $\ell^{inv}(e) = (o_1, o_2)$ where $(i_1, l_1, o_1) \in \ell(v_1)$ and $(i_2, l_2, o_2) \in \ell(v_2)$ are the two adjacent segments. Only one orientation bit per block occurrence as label on the vertices is not sufficient as the example in Fig. 4.5 proves.

4.1.3 Enredo graphs

Enredo graphs have been introduced in the context of a pipeline for genome alignment that consists of the programs Enredo and Pecan [124]. The program Enredo partitions genomes into segments with the help of Enredo graphs, and the program Pecan computes nucleotide-level co-linear alignments of the segments. The structure of Enredo graphs resembles breakpoint graphs from rearrangement studies [3, 12, 86].

Since we describe Enredo graphs separately from the Enredo method, the definitions below differ slightly from the original definition [124]. See Section B of the appendix for a description of the differences.

Definition of graph model and structure. Let now $M = (G, \ell)$ be an Enredo graph model defined by the Enredo graph structure $G = (V, E)$ and a labeling function ℓ for undirected edges of the structure G. Enredo graphs have two vertices per block that are connected by two sets of edges $E = E_A \cup E_B$. Directed adjacency edges E_A represent segment adjacencies, and undirected block edges E_B connect the head and tail vertex of blocks. Thus, Enredo graphs are mixed graphs. See Fig. 4.6 for an example Enredo graph.

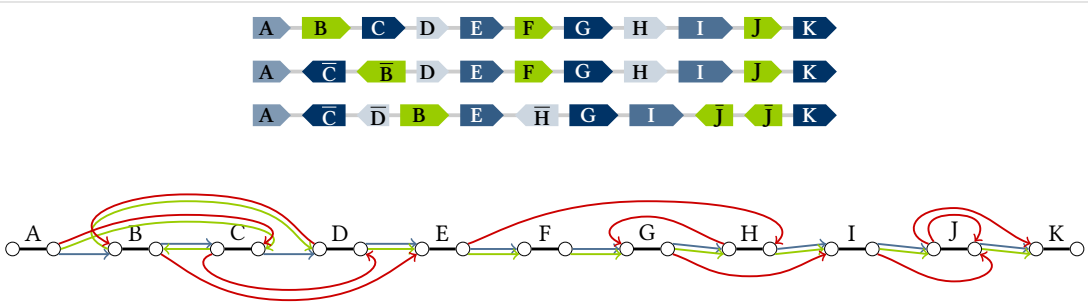

Figure 4.6: An example Enredo graph (bottom) for the same set of blocks as shown in Fig. 4.1 (top). Adjacency edges are colored by genome, block edges are shown in black. Note that colors and block edge labels are not present in the structure G of an Enredo graph.

Vertices. An Enredo graph structure has two vertices for each block $B \in \mathcal{B}$, a tail vertex $v_t \in V$ and head vertex $v_h \in V$. The two vertices represent the ends of B. Given only the structure and not the model, the tail vertex and the head vertex are often not distinguishable from each other.

Undirected edges. In the Enredo graph structure, there is an undirected block edge $e \in E_B$ for every pair of tail and head vertex $v_t, v_h \in V$ that represents the two ends of a block $B \in \mathcal{B}$. The labeling function $\ell : E_B \to \mathcal{B}$ of the Enredo graph model associates each block edge $e \in E_B$ of the Enredo graph structure with the corresponding block $B \in \mathcal{B}$ such that $\ell(e) = B$.

Directed edges. The Enredo graph structure has a directed adjacency edge $e \in E_A$ for every pair of adjacent segments $s_1 \in B_1$ and $s_2 \in B_2$, where $B_1, B_2 \in \mathcal{B}$. Like in A-Bruijn graphs, the orientation of e depends on the start positions i_1 and i_2 of the segments $s_1 = (i_1, l_1, o_1)$ and $s_2 = (i_2, l_2, o_2)$, the edge is a loop if $B_1 = B_2$, and G is a multigraph. W. l. o. g. let $i_1 < i_2$. Then, the source vertex of the edge e is the head vertex of block B_1 if $o_1 = +$, and the tail vertex of B_1 if $o_1 = -$. Further, the target vertex of the edge e is the tail vertex of block B_2 if $o_2 = +$, and the head vertex of B_2 if $o_2 = -$.

Recovering $\mathcal{B}$. Given the labeling function ℓ of the Enredo graph model M, which labels each block edge $e \in E_B$ of the Enredo graph structure G with a block $B \in \mathcal{B}$, recovering the set of blocks $\mathcal{B}$ from M is straightforward: The set of blocks $\mathcal{B}$ is simply the set of labels $\ell(e)$ of all block edges $e \in E_B$.

Representation of non-colinearity in the graph structure. All the types of non-colinear changes are visible in the Enredo graph structure but duplications and translocations create ambiguity. Fig. 4.7 shows the same example for ambiguity in the Enredo graph structure as Fig. 4.4 shows in the A-Bruijn graph structure. The representation of blocks in two vertices that are connected by a block edge does not resolve this ambiguity but visualizes the relative orientations of block occurrences, thus models inversions.

To resolve ambiguity of duplications and translocations, we define the sparse labeling function $\ell^{dup} : E_A \to \mathbb{N}$ as a strict total order on the edges E. The function ℓ^{dup} numbers the edges such

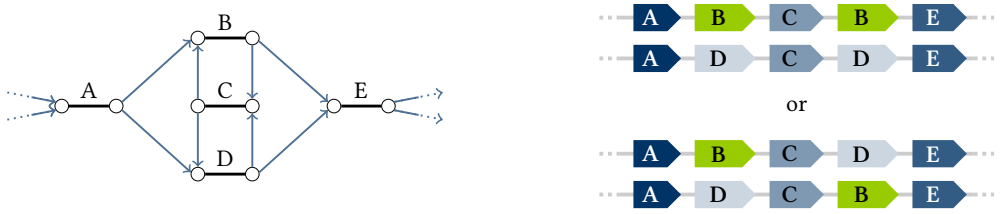

Figure 4.7: A section of an Enredo graph structure (left) representing five blocks each of size two. The example demonstrates the ambiguity of Enredo graph structures. The structure results either from duplications (top right) or from translocations (bottom right) of block B and D.

that $\ell^{dup}(e_1) < \ell^{dup}(e_2)$ for two edges $e_1, e_2 \in E_A$ if $a_1 < a_2$, where a_1 is the adjacency position represented by e_1 and a_2 is the adjacency position represented by e_2. This is analogous to the definition of ℓ^{dup} in A-Bruijn graphs. Again, the full labels ℓ of the graph model are necessary to determine the adjacency positions.

4.1.4 Cactus graphs

Paten et al. introduced cactus graphs for whole-genome alignments in 2011 [121, 122]. Cactus graphs in general date back to 1953 [75]. They have the property that every edge belongs to at most one simple cycle, or equivalently, any two simple cycles share at most one vertex. This is called the cactus property. Cactus graphs for whole-genome alignments have a second property, the existence of an Eulerian circuit. As a consequence, every edge belongs to exactly one simple cycle. In the following, we only refer to cactus graphs that fulfill both properties.

The two properties of cactus graphs are favorable for representing genome alignments: The cactus property subdivides the graph (and genomes) into independent units, and the Eulerian circuit conveniently provides a consensus genome.

Definition of graph model and structure. In the following, let $M = (G, \ell)$ be a cactus graph model defined by a cactus graph structure $G = (V, E)$ and a labeling function ℓ for the edges of the structure G. The vertices of cactus graph structures represent sets of adjacencies. All edges are undirected and represent blocks. See Fig. 4.8 for an example cactus graph.

Vertices. The vertices V of the cactus graph structure G partition the set of all segment adjacencies into a set of pairwisely disjoint subsets Ω. There is a vertex $v \in V$ for each subset $\nu \in \Omega$. The exact steps to determine Ω are described in Section 4.2.3 along with the transformation from Enredo graphs.

Edges. The cactus graph structure has an undirected edge $e \in E$ for every block $B \in \mathcal{B}$. The edge e connects the two vertices $u, v \in V$ that represent two sets of adjacencies $\mu, \nu \in \Omega$, where μ contains all adjacencies of one end of B and ν contains all adjacencies of the other end of B. The edge e is a loop if $\mu = \nu$. The labeling function $\ell : E \to \mathcal{B}$ of the cactus graph model

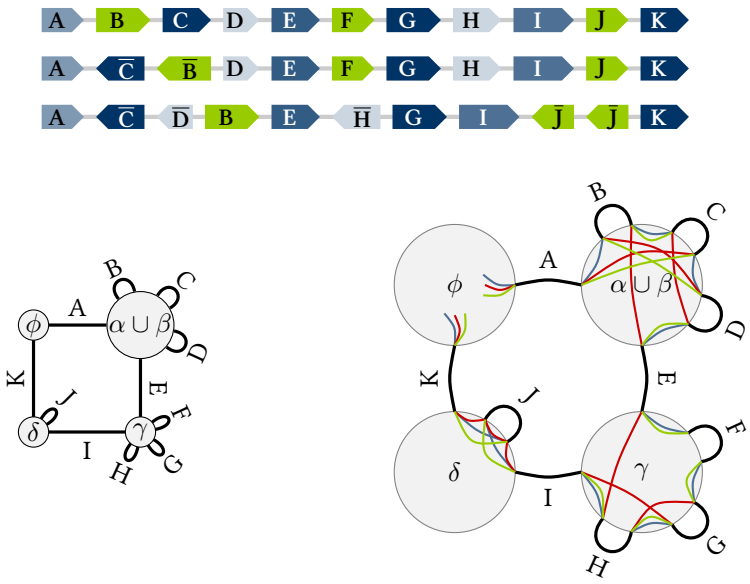

Figure 4.8: An example cactus graph (bottom left) for the same set of blocks as shown in Fig. 4.1 (top). Adjacencies are hidden in the vertices of cactus graphs. On the bottom right, all adjacencies are shown in enlarged vertices colored by genome.

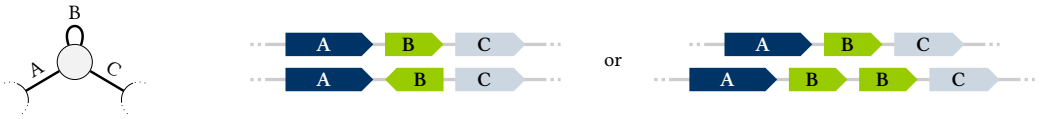

Figure 4.9: A section of a cactus graph (left) representing a set of three blocks. The example demonstrates the ambiguity of cactus graph structures. The structure results either from an inversion (middle) or from a duplication (right) of block B.

associates each edge $e \in E$ of the cactus graph structure with the corresponding block $B \in \mathcal{B}$ such that $\ell(e) = B$.

Recovering $\mathcal{B}$. Given the labeling function ℓ of the cactus graph model M, which labels each edge $e \in E$ of the cactus graph structure G with a block $B \in \mathcal{B}$, recovering the set of blocks $\mathcal{B}$ from M is straightforward: The set of blocks $\mathcal{B}$ is simply the set of labels $\ell(e)$ of all edges $e \in E$.

Representation of non-colinearity in the graph structure. All the types of non-colinear changes can be visible in the cactus graph structure and appear as cycles, but they all create ambiguity. Fig. 4.9 shows an example cactus graph structure that either represents an inversion or a duplication. The figure demonstrates that it is not possible to unambiguously recognize single duplications or inversions in the cactus graph structure G, and the example in Fig. 4.14 demonstrates the same for translocations. Inverted tandem duplications may not be visible at all

Figure 4.10: A section of a cactus graph structure (left) representing a set of three blocks, and two possible sections of genomes (right) that result in the structure. The inverted tandem duplication in the first genome is not visible in the cactus graph structure.

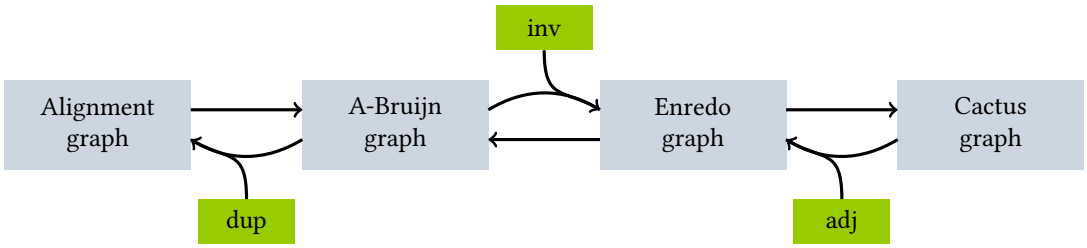

Figure 4.11: The transformations between the four graph structures (light blue boxes) described in 4.2. Bent arrows indicate that a transformation is ambiguous without labels and green boxes name the sparse labeling functions used as additional input for describing these transformations.

as Fig. 4.10 shows.

To resolve ambiguity in the cactus graph structure, we define the sparse labeling function $\ell^{adj} : E \to 2^{\{+,-\}\times\mathbb{N}}$. The function ℓ^{adj} associates an edge $e \in E$ of the cactus graph structure with a set of pairs of orientation bits and positive numbers, one pair for each occurrence of the block $\ell(e) = B$. The orientation bits in the labels ℓ^{adj} indicate the relative orientation of the block occurrences. The numbers impose a strict total order $\prec$ on all block occurrences $s \in S_{\mathcal{B}}$: The relation $s_1 \prec s_2$ holds for two block occurrences $s_1, s_2 \in S_{\mathcal{B}}$ if $i_1 < i_2$, where $s_1 = (i_1, l_1, o_1)$ and $s_2 = (i_2, l_2, o_2)$. The total order on the edges resolves ambiguity for duplications and translocations as well as for inversions of blocks that are not represented by loop edges. To resolve ambiguity for all inversions, the orientation bits are necessary.

4.2 Transformations between the graph structures

A *transformation* from a graph structure G to another graph structure G' is an operation that outputs G' given G, where both G and G' represent the same genome alignment. This part of the chapter describes how it is possible to transform the four graph structures described in 4.1 into each other. More precisely, it addresses the transformations depicted as arrows in Fig. 4.11.

Transformations between graph *models* are trivial. The labeling functions ℓ are defined such that it is possible to recover the set of blocks $\mathcal{B}$ from the graph models $M = (G, \ell)$. The set $\mathcal{B}$ is input for building all four graph models as described in 4.1. Thus, a transformation from a graph model M to another graph model M' is always possible via $\mathcal{B}$.

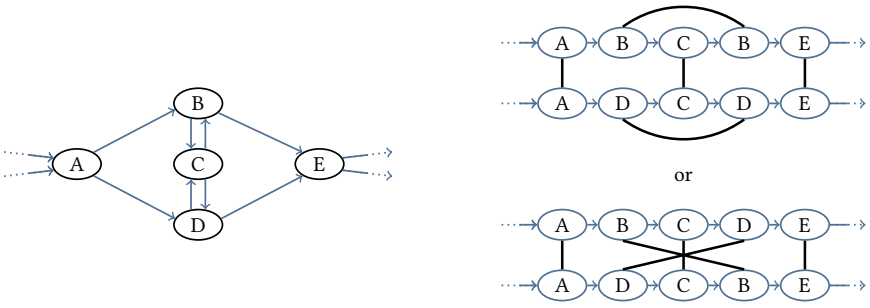

Figure 4.12: A section of an A-Bruijn graph (left) that can be transformed into two alternative sections of alignment graphs (right). The example demonstrates that a transformation from A-Bruijn graph structures to alignment graph structures is ambiguous.

Since the graph structures differ in their representation of non-colinear changes, some transformations between the graph structures are impossible (bent arrows in Fig. 4.11). More precisely, a transformation from G to G' is impossible if G comprises less information than G'. If this is the case, the sections below provide examples for ambiguity that prove differences in the information content of the graph structures. The corresponding transformations are described using the sparse labeling functions as additional input.

4.2.1 Transformations between alignment and A-Bruijn graphs

The alignment graph structure and the A-Bruijn graph structure both do not model inversions (see Section 4.1.1 and 4.1.2). Thus, additional labels that recover information about the orientation of segments are not necessary for a transformation between the two graph structures. However, the A-Bruijn graph structure is ambiguous for duplications and translocations, whereas the alignment graph structure is not. For this reason, the transformation from A-Bruijn graph structures to alignment graph structures requires additional labels that resolve this ambiguity (see also Fig. 4.12).

From alignment graphs to A-Bruijn graphs. Given only the structure of an alignment graph $G = (V, E_A \cup E_B)$, the following describes the transformation from G to an A-Bruijn graph structure $G' = (V', E')$. The description follows the construction of A-Bruijn graphs in [131] and uses a many-to-one mapping m from alignment graph vertices V to A-Bruijn graph vertices V'.

First, compute the set of E_B-connected components $\mathcal{C}$ of the alignment graph structure. Next, add a vertex v' to the initially empty set of A-Bruijn graph vertices V' for each E_B-connected component $C \in \mathcal{C}$. Let $V_C \subseteq V$ be the set of vertices of the E_B-connected component C. Keep the mappings of all vertices $v \in V_C$ to the vertex v' as labels $m[v] = v'$.

Using this mapping, transfer the adjacency edges E_A to the A-Bruijn graph structure as follows: For the source and target vertices $u, v \in V$ of each edge $e = (u, v)$ in the set of alignment

graph adjacency edges E_A, determine the corresponding A-Bruijn graph vertices $u' = m[u]$ and $v' = m[v]$. Finally, add an edge $e' = (u', v')$ to the set of A-Bruijn graph edges E'.

From A-Bruijn graphs to alignment graphs. Given the structure of an A-Bruijn graph $G = (V, E)$ and the sparse labeling function $\ell^{dup} : E \to \mathbb{N}$, this section describes the transformation from G with the labels ℓ^{dup} to an alignment graph structure $G' = (V', E'_A \cup E'_B)$. The transformation successively creates a one-to-many mapping m from A-Bruijn graph vertices V to alignment graph vertices V'.

Initially, set $m[v] = \{\}$ for all $v \in V$. Every time when adding a vertex v' to the set of alignment graph vertices V' in the following steps, update the mapping $m[v]$ of the corresponding A-Bruijn graph vertex $v \in V$. Furthermore, connect v' with all other vertices $u' \in m[v]$ by block edges E'_B. Then, the mapping $m[v] = \{v'_1, v'_2, \ldots, v'_{|V'_C|}\}$ holds all vertices V'_C of a E'_B-connected component C of G' at the end of the transformation.

Iterate over the edges $E = \{e_1, e_2, \ldots, e_{|E|}\}$ of the A-Bruijn graph in increasing order of labels: $\ell^{dup}(e_1) < \ell^{dup}(e_2) < \cdots < \ell^{dup}(e_{|E|})$. For each edge $e_i \in E$, $i = 1, \ldots, |E|$ apply the following steps: If $e_i = (u, v)$ represents the adjacency of the first block in a chromosome, add a vertex u' to the set of alignment graph vertices V'. Add block edges between u' and all vertices in the set $m[u]$ to the set E'_B, and update $m[u]$ by adding u'. Next, add a vertex v' to the set of alignment graph vertices V'. Add block edges between v' and all vertices in the set $m[v]$ to the set E'_B, and then update $m[u]$ by adding v'. Finally, add an edge $e' = (u', v')$ to the set of alignment graph adjacency edges E'_A. If the edge represents not the last adjacency in a chromosome, keep v' for the next edge $e_{i+1} \in E$ as u'.

4.2.2 Transformations between A-Bruijn and Enredo graphs

In both the A-Bruijn graph structure and the Enredo graph structure duplications and translocations create ambiguity. The two structures have the same information content with respect to duplications and translocations as the transformations below show. However, the A-Bruijn graph structure does not model inversions, whereas the Enredo graph structure does. Thus, the transformation from A-Bruijn graph structures to Enredo graph structures requires additional labels that indicate the orientation of block occurrences in the A-Bruijn graph (see also Fig. 4.13).

The following transformations are based on the definition of Enredo graphs from Section 4.1.3 with the differences to the original definition as explained in Section B of the appendix.

From A-Bruijn graphs to Enredo graphs. Given an A-Bruijn graph structure $G = (V, E)$ and the sparse labeling function $\ell^{inv} : E \to \{+, -\} \times \{+, -\}$, the following describes the transformation from G with the labels ℓ^{inv} to an Enredo graph structure $G' = (V', E'_A \cup E'_B)$. The transformation uses a one-to-one mapping m from A-Bruijn graph vertices V to Enredo graph block edges E'_B. Furthermore, the Enredo graph vertices V' need to be labeled as head or tail vertices to correctly add the adjacency edges E'_A.

Figure 4.13: A section of an A-Bruijn graph structure (left) that can be transformed into two alternative sections of Enredo graph structures (right). The example demonstrates that a transformation from A-Bruijn graph structures to Enredo graph structures is ambiguous.

First, add a tail vertex v'_t and a head vertex v'_h to the set of Enredo graph vertices V' and a block edge $e' = \{v'_t, v'_h\}$ to the set of Enredo graph block edges E'_B for each A-Bruijn graph vertex $v \in V$. Keep the mappings of all vertices $v \in V$ to the block edges $e' \in E'_B$ as labels $m[v] = e'$ for the next steps.

Transfer the A-Bruijn graph edges E to the Enredo graph structure using ℓ^{inv} and m as follows: For the source and target vertices $u, v \in V$ of every edge $e = (u, v)$ in the set of A-Bruijn graph edges E, determine the corresponding Enredo graph block edges $m[u] = \{u'_t, u'_h\}$ and $m[v] = \{v'_t, v'_h\}$. Add an edge $e' = (u'_x, v'_y)$ to the set of Enredo graph adjacency edges E'_A and choose u'_x and v'_y according to the label $\ell^{inv}(e) = (o_1, o_2)$. If $o_1 = +$, the source vertex $u'_x = u'_h$ and otherwise $u'_x = u'_t$. If $o_2 = +$, the target vertex $v'_y = v'_t$ and otherwise $v'_y = v'_h$.

From Enredo graphs to A-Bruijn graphs. Given only the structure of an Enredo graph $G = (V, E_A \cup E_B)$, this section describes the transformation from G to an A-Bruijn graph structure $G' = (V', E')$. The description uses a one-to-one mapping m from Enredo graph block edges E_B to A-Bruijn graph vertices V'.

As a first step, add a vertex v' to the set of A-Bruijn graph vertices V' for each Enredo graph block edge $e \in E_B$. Keep the mapping of the edge e to the vertex v' as a label $m[e] = v'$.

Transfer the Enredo graph adjacency edges E_A to the A-Bruijn graph structure as follows: For each adjacency edge $e = (u, v)$, find the two block edges $e_u, e_v \in E_B$ that are incident to the the two endpoints $u, v \in V$ of e. Determine the corresponding vertices $u' = m[e_u]$ and $v' = m[e_v]$ in the set of A-Bruijn graph vertices V'. Finally, add an edge $e' = (u', v')$ to the set of A-Bruijn graph edges E'.

4.2.3 Transformations between Enredo and cactus graphs

In both the Enredo graph structure and the cactus graph structure all types of non-colinear changes can be visible but both structures are ambiguous. However, the Enredo graph structure is only ambiguous for duplications and translocations, whereas the cactus graph structure is also ambiguous for inversions. Furthermore, the transformation from Enredo graph structures to cactus graph structures can be ambiguous even if no inversions are present as the example in

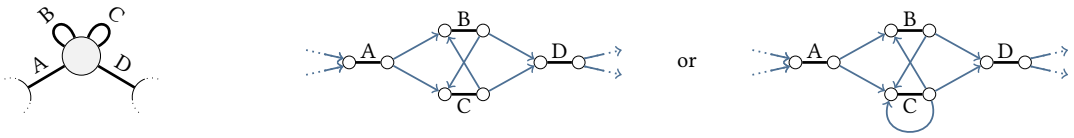

Figure 4.14: A section of a cactus graph structure (left) that can be transformed into two alternative sections of Enredo graph structures (middle and right). The example demonstrates that a transformation from cactus graph structures to Enredo graph structures is ambiguous.

Fig. 4.14 proves. The cactus graph structure hides information about adjacencies in vertices that is visible in the Enredo graph structure. For this reason, the transformation from cactus graph structures to Enredo graph structures requires additional labels that indicate the orientation and adjacencies of block occurrences in cactus graphs.

From Enredo graphs to cactus graphs. Given only the structure of an Enredo graph $G = (V, E_A \cup E_B)$, this section describes the transformation from G to a cactus graph structure $G' = (V', E')$. The description follows the construction of cactus graphs in [121, 122]. It uses a one-to-many mapping m from Enredo graph vertices V to cactus graph vertices V' for constructing a precursor cactus graph. This precursor cactus graph is subsequently modified in two steps to yield the final cactus graph G'.

First, add additional adjacency edges to the Enredo graph that connect the ends of each genome. Compute the set of E_A-connected components $\mathcal{C}_A$ of the Enredo graph structure. Next, add a vertex v' to the set of cactus graph vertices V' for each $C \in \mathcal{C}_A$. Let $V_C \subseteq V$ be the set of Enredo graph vertices of C. Keep the mappings of all vertices $v \in V_C$ to the vertex v' as labels $m[v] = v'$. The additional adjacency edges in the Enredo graph ensure that the ends of all genomes are represented by a single vertex in the cactus graph, the *origin vertex* ϕ.

Using the mapping, transfer the block edges E_B to the cactus graph structure as follows: For the source and the target vertices $u, v \in V$ of each edge $e = \{u, v\}$ in the set of Enredo graph block edges E_B, determine the corresponding cactus graph vertices $u' = m[u]$ and $v' = m[v]$. Add an edge $e' = \{u', v'\}$ to the set of cactus graph edges E'. This yields the precursor cactus graph (see Fig. 4.15 for an example).

Compute the set of 3-edge connected components $\mathcal{C}_3$ of the precursor cactus graph. Add a vertex v' to the set of cactus graph vertices V' for each 3-edge connected component $C \in \mathcal{C}_3$. Let $V'_C \subseteq V'$ be the set of vertices of C. For each cactus graph vertex $w' \in V'_C$, determine the set of precursor cactus graph edges $E'_w \subseteq E'$ incident to w'. For each edge $e' = \{u', w'\}$ in the set E'_w, add another edge $e' = \{u', v'\}$ to the set of cactus graph edges E'. Finally, remove w' and all incident edges $e' \in E'_w$ from the precursor cactus graph.

In a last step, identify the set of edges $E'_2 \subseteq E'$ that disconnect the modified precursor cactus graph. Compute the set of connected components $\mathcal{C}_2$ of the E'_2-induced subgraph of the modified precursor cactus graph. Replace the vertices $V_C \subseteq V$ of each connected component $C \in \mathcal{C}_2$ and the incident edges by a single vertex v' and corresponding edges as described for 3-edge

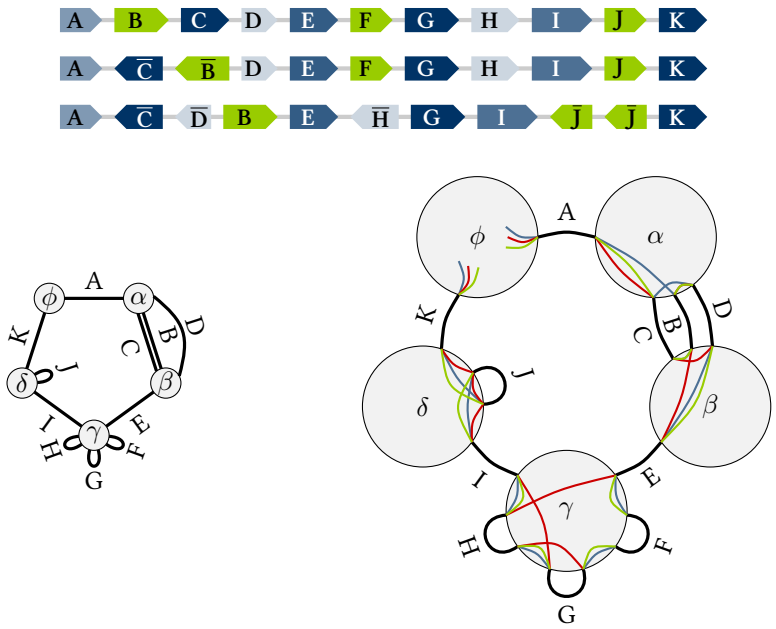

Figure 4.15: An example precursor cactus graph (bottom left) for the same set of blocks as shown in Fig. 4.1 (top) and the final cactus graph shown in Fig. 4.8. Adjacencies are already hidden in the vertices of the precursor cactus graph. On the bottom right, all adjacencies are shown in enlarged vertices colored by genome.

connected components in the previous step. This concludes the transformation to the cactus graph $G' = (V', E')$.

From cactus graphs to Enredo graphs. Given the structure of a cactus graph $G = (V, E)$ and the sparse labeling function $\ell^{adj} : E \to 2^{\{+,-\}\times\mathbb{N}}$, the following describes the transformation from G with the labels ℓ^{adj} to an Enredo graph structure $G' = (V', E'_A \cup E'_B)$. The transformation uses a one-to-one mapping m from cactus graph edges E to Enredo graph block edges E'_B with a direction. The direction of the block edges in the mapping is arbitrary but fixed and allows to distinguish between tail and head vertex.

Iterate over the cactus graph edges E visiting each edge $e \in E$ as many times as there are pairs in the label $\ell^{adj}(e)$ (this is the number of occurrences of the represented block): Start with the edge $e_0 \in E$ that contains the smallest number n_0 in one of the pairs in its label $\ell^{adj}(e_0)$. This edge $e_0 = \{\phi, \alpha\}$ is incident to the origin vertex $\phi \in V$ and another vertex $\alpha \in V$. It is possible that $\alpha = \phi$. Continue with the edge $e_1 \in E$ that contains the next larger number $n_1 > n_0$ in one of the pairs in its label $\ell^{adj}(e_1)$. The edge $e_1 = \{\alpha, \beta\}$ is incident to the vertex α and to another vertex $\beta \in V$. Note that $e_0 = e_1$ is possible if $\phi = \beta$. Repeat the same until reaching the end of all genomes $\mathcal{G}$.

During the iteration, add vertices and edges to the Enredo graph. For e_0, add a pair of vertices u'_t, u'_h to the set of Enredo graph vertices V' and an edge $e' = \{u'_t, u'_h\}$ to the set of Enredo graph

block edges E'_B. Map e_0 to e' as a label $m[e_0] = (u'_t, u'_h)$, and keep u'_h for the next edge as u'_x.

For every further edge $e_i \in E$ visited during the iteration, add a pair of vertices v'_t, v'_h and a block edge $e' = \{v'_t, v'_h\}$ to the Enredo graph if the mapping of the edge e_i is unspecified. Otherwise, determine the block edge $e' \in E'_B$ using the label $m[e_i] = (v'_t, v'_h)$. In both cases, add a directed edge $e'_a = (u'_x, v'_y)$ to the set of Enredo graph adjacency edges E'_A. The vertex u'_x is from the previous step. The orientation bit o associated with the number in the label $\ell^{adj}(e_i)$ determines v'_y. If $o = +$, then $v'_y = v'_t$ and keep v'_h for the next step as u'_x. If $o = -$, then $v'_y = v'_h$ and keep v'_t for the next step as u'_x.

4.3 Substructures in the alignments

Substructures in genome alignments provide information about the rearrangement of the genomes with respect to each other and about inconsistencies in the alignment. This part of the chapter describes classes of substructures that we compiled from several graph-based genome alignment approaches, namely from `ABA` [133], `DRIMM-Synteny` [132], `Enredo` [124], and `Cactus` [121].

The following three sections provide definitions for colinear paths, visiting paths, and small cycles on the set of blocks $\mathcal{B}$. In addition, the sections describe the appearance of these substructures in the graph structures, and discuss whether it is possible to unambiguously identify them in the graph structures. Identification in the graph models is always possible given that $\mathcal{B}$ can be recovered from M_G. Finally, the last section briefly addresses two somewhat different substructures used by the `Cactus` method, chains and groups.

4.3.1 Colinear paths

The set of maximal colinear paths of a genome alignment defines the segmentation of the genomes. Although only the `Enredo` method explicitly addresses colinear paths (see also Section 4.4.3), all graph-based genome alignment approaches aim at maximizing colinear paths in size and length.

Definition of maximal colinear paths. Given a set of blocks $\mathcal{B}$ defined on a set of genomes $\mathcal{G}$. A *colinear path* is a set of $k \geq 1$ blocks $B_1, \ldots, B_k \in \mathcal{B}$ with the following property for all B_i with $i = 2, \ldots, k-1$: One end of block B_i and one end of block B_{i-1} are adjacent without breakpoint, and the other end of B_i and one end of block B_{i+1} are adjacent without breakpoint. For the remainder of this chapter, we assume w. l. o. g. that the head of B_1 is adjacent to B_2 and the tail of B_k is adjacent to B_{k-1} for any colinear path. A colinear path is *maximal* if it is bounded by breakpoints, that is, any block that is adjacent to B_1 or B_k other than B_2 and B_{k-1} is adjacent with breakpoint.

All blocks of a colinear path have the same size b, which is also the *size* of the colinear path. The concatenations of the adjacent segments in a colinear path give longer segments $s_1, \ldots, s_b$ of

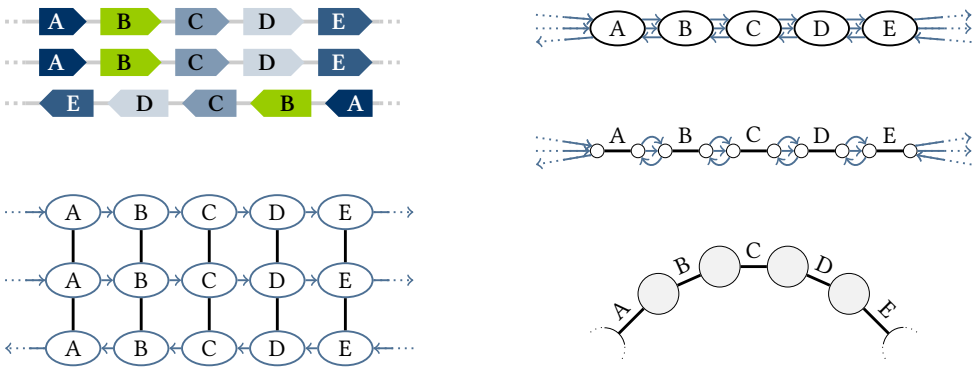

Figure 4.16: A colinear path formed by three segments each being the concatenation of occurrences of blocks A, B, C, D, and E. The figure displays the relevant parts of genomes as sequences of blocks (top left) as well as the appearance of this colinear path in an alignment graph (bottom left), in an A-Bruijn graph (top right), in an Enredo graph (middle right), and in a cactus graph (bottom right).

the genomes $\mathcal{G}$ reaching from the tails of the segments in B_1 to the heads of the segments in B_k. For every $j = 1, \ldots, b$, the segment s_j can be in the forward orientation or in the reverse complemented orientation. The *length* of the colinear path is the length of the segments $s_1, \ldots, s_b$. If the segments have varying lengths, we suggest to use the median of $|s_1|, \ldots, |s_b|$ as the length of the colinear path.

Appearance in graph structures. Figure 4.16 shows an example colinear path and its representation in the four graph structures.

In an alignment graph structure $G = (V, E_A \cup E_B)$, a colinear path appears as a set of E_B-connected components $C_1, \ldots, C_k$. In this set, all E_B-connected components C_i with $i = 1, \ldots, k$ have the same number of vertices b. Furthermore, exactly b adjacency edges connect the vertices of two consecutive E_B-connected components C_i and C_{i+1} for all $i = 1, \ldots, k-1$.

In an A-Bruijn, Enredo, and cactus graph structure, colinear paths appear as simple paths without branching vertices except for the first and last vertex. More formally, let V be the set of vertices of any of the three graph structures. Then, a colinear path appears as a simple path $v_1 v_2 \ldots v_{k-1} v_k$ where $v_1, \ldots, v_k \in V$ and where $v_2, \ldots, v_{k-1}$ are non-branching vertices.

Identification in graph structures. In order to identify a colinear path unambiguously, a graph structure needs to model all inversions. Therefore, the identification of colinear paths in alignment, A-Bruijn, and cactus graph structures is not possible.

The Enredo graph structure is the only structure that allows to identify colinear paths unambiguously. Inversions are visible and do not create ambiguity. In addition, duplications and translocations introduce branching vertices. Thus, every simple path in Enredo graphs without branching vertices as described above is a colinear path.

4.3.2 Visiting paths

We here suggest to summarize a number of similar substructures as visiting paths. These substructures include "microblocks" from `DRIMM-Synteny`, the blocks split in the "joining" operation of the `Enredo` method, and the first type of "aberrant homologies" from the `Enredo` method as well as retrotransposed pseudogenes from the `Enredo` method.

Definition of visiting paths. *Visiting paths* are maximal colinear paths with an additional property of the adjacencies of the first and last block. Given again a set of blocks $\mathcal{B}$ defined on a set of genomes $\mathcal{G}$. Let the set of blocks $B_1, \ldots, B_k \in \mathcal{B}$ be a maximal colinear path of size b and let $s_1, \ldots, s_b$ be the concatenations of the adjacent block occurrences along the colinear path. Then, $B_1 \ldots B_k$ is also a visiting path if there is a pair of blocks $B_0, B_{k+1} \in \mathcal{B}$ where B_0 is adjacent to B_1 and B_{k+1} is adjacent to B_k such that each segment s_j, $j \in \{1, \ldots, b\}$, that is adjacent to a segment in B_k is also adjacent to a segment in B_0, and each segment s_j that is adjacent to B_0 is also adjacent to B_k.

Appearance in graph structures. Figure 4.17 shows an example visiting path and its representation in the four graph structures. The appearance of visiting paths in the graph structures is similar to the appearance of colinear paths. In cactus graph structures, it is even identical. The following describes the differences for alignment, A-Bruijn, and Enredo graph structures.

For every visiting path in alignment graph structures, there are E_B-connected components C_0 and C_{k+1} that are connected by adjacency edges to the first and last E_B-connected components C_1 and C_k of the visiting path, respectively. Furthermore, there is a path without block edges from C_0 to C_{k+1} for every segment in C_0 that is adjacent to a segment in C_1 and for every segment in C_{k+1} that is adjacent to a segment in C_k.

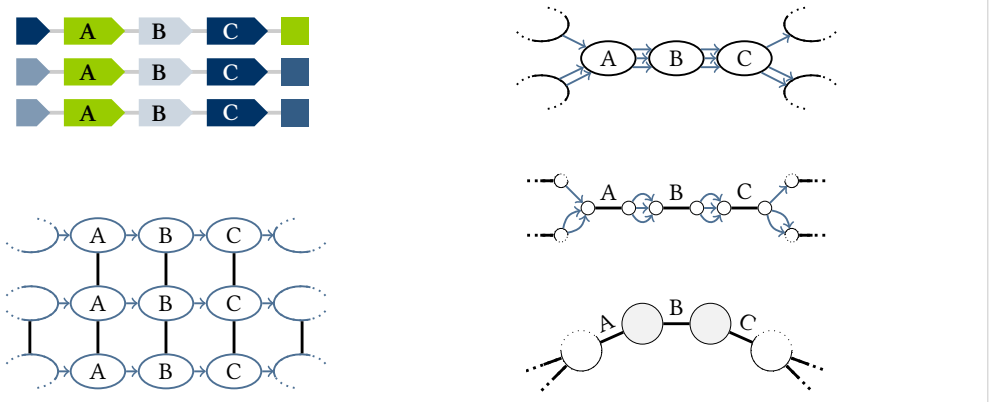

Figure 4.17: A visiting path formed by three segments each being the concatenation of occurrences of blocks A, B, and C. The figure displays the relevant parts of genomes as sequences of blocks (top left) as well as the appearance of this visiting path in an alignment graph (bottom left), in an A-Bruijn graph (top right), in an Enredo graph (middle right), and in a cactus graph (bottom right). Dotted and incomplete vertices indicate the possibility of more incident edges.

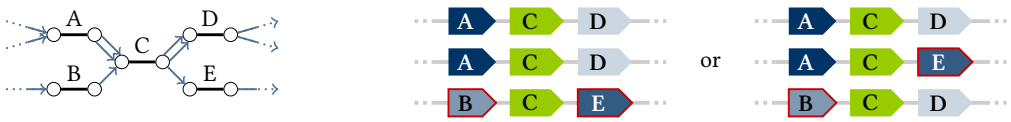

Figure 4.18: A piece of an Enredo graph structure (left) representing five blocks A-E, where block C is potentially a visiting path. Block C is a visiting path if the two occurrences of block A are adjacent to the two occurrences of C that are adjacent to block D (middle). However, the Enredo graph structure shown on the left can as well represent the genome segments where one occurrence of block C that is adjacent to block A is adjacent to block E (right).

For every vising path in A-Bruijn graph structures and in Enredo graph structures, there are two vertices $v_0, v_{k+1} \in V$ connected to the branching vertices v_1 and v_k of the visiting path. The vertex v_0 is connected to v_1 by the same number of directed edges as v_{k+1} is connected to v_k.

Identification in graph structures. In order to identify a visiting path unambiguously, a graph structure needs to model inversions and it must be possible to pair the two adjacencies of each block occurrence. In alignment, A-Bruijn, and cactus graph structures, not all inversions are visible just as described above for colinear paths. In A-Bruijn, Enredo, and cactus graph structures, only the labels ℓ, ℓ^{dup}, or ℓ^{adj} provide the pairings of adjacencies of each block occurrence. Therefore, the identification of visiting paths is not possible in any of the four graph structures. Fig. 4.18 shows an example for the ambiguity of the Enredo graph structure for visiting paths.

4.3.3 Short cycles

Cycles are characteristic for non-colinear alignments. Graph structures without cycles represent colinear alignments [97, 136]. Therefore, cycles play an important role for graph-based genome alignment approaches. The A-Bruijn graph approaches `ABA` and `DRIMM-Synteny` mainly focus on cycles, and also the `Enredo` method handles cycles.

Definition of short cycles. Given a set of blocks $\mathcal{B}$ defined on a set of genomes $\mathcal{G}$. A *cycle* is a set of $k \geq 1$ blocks $B_1, \ldots, B_k \in \mathcal{B}$ where B_k and B_1 are adjacent and all B_i and B_{i+1} are adjacent for $i = 1, \ldots, k-1$. The set of block adjacencies along a cycle has exactly k different elements. Any of the adjacencies can be breakpoints and the blocks may have different sizes.

The *length* of a cycle is the total length of all blocks forming the cycle. If the segments in a block have varying length, we suggest to use the median. Given a length threshold t, a cycle is *short* if its length is at most t.

Appearance in graph structures. Figure 4.19 shows three example cycles and their representation in the four graph structures. In alignment graph structures, most cycles appear as mixed simple cycles with at least one adjacency edge. In A-Bruijn graph structures, they appear as

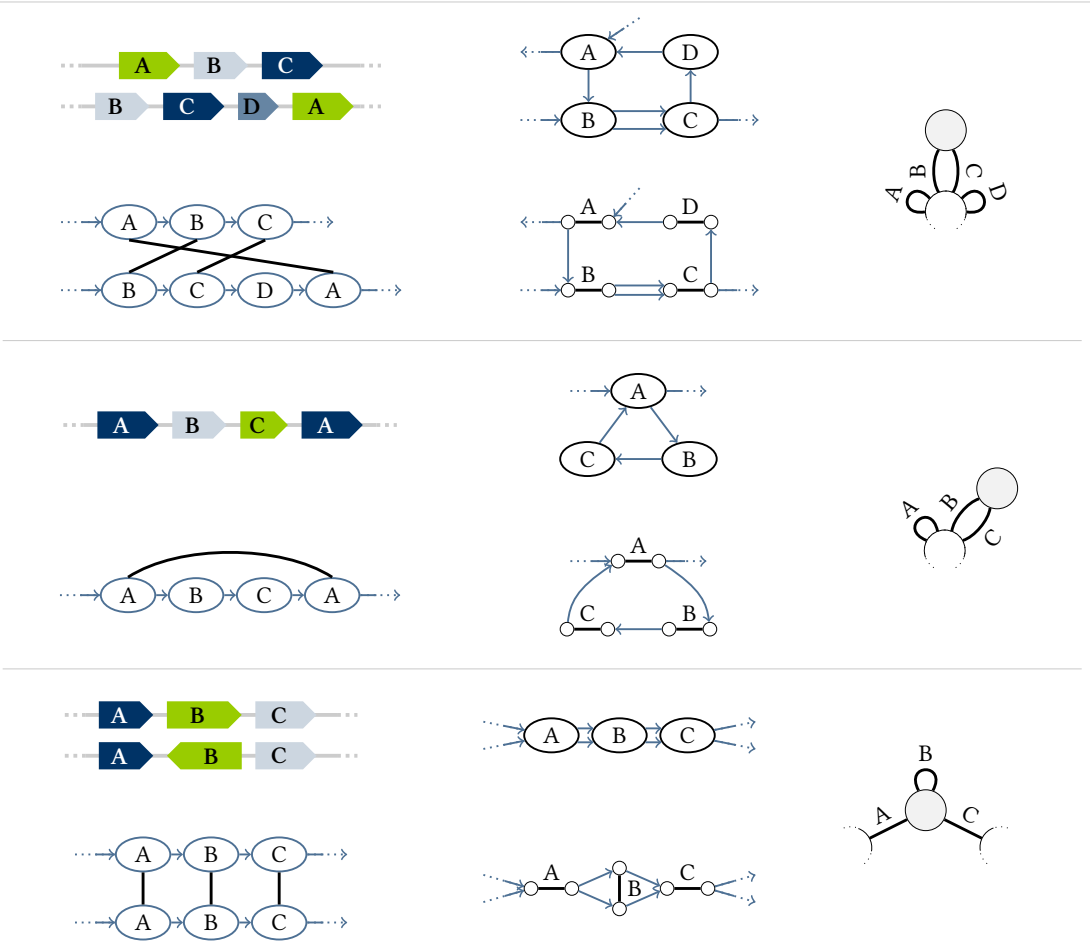

Figure 4.19: Three cycles that result from a translocation (top panel), a duplication (middle panel), and an inversion (bottom panel). Each panel displays the relevant parts of genomes as sequences of blocks (top left) as well as the appearance in an alignment graph (bottom left), in an A-Bruijn graph (top middle), in an Enredo graph (bottom middle), and in a cactus graph (right).

simple cycles. However, since alignment and A-Bruijn graph structures do not model inversions, some cycles are not visible in the two structures. In Enredo graph structures, all cycles appear as mixed simple cycles. A single cycle in the set of blocks can appear as several cycles in all three graph structures. Finally in cactus graph structures, most cycles appear as cycles that are not necessarily simple. Note that the direction of edges does not matter for the definition of a cycle in a graph structure (see Section 2.5).

Identification in graph structures. A cycle in the alignment, A-Bruijn and Enredo graph structures always corresponds to a cycle in the set of blocks though several cycles in the graph structures may correspond to the same cycle in the set of blocks. It is possible to detect cycles in the graph structure with established algorithms [60].

However, since not all cycles in the set of blocks appear as cycles in alignment and A-Bruijn

graph structures, it is not possible to identify all cycles given only the structures of these two graphs. Similarly, not all cycles in the set of blocks appear as cycles in the cactus graph structure. Thus, only the Enredo graph structure allows identification of all cycles in the set of blocks.

4.3.4 Substructures in cactus graphs

As a result of the complex construction steps, the cactus graph structure exhibits distinct substructures compared to the other graph structures. In an iterative procedure, the `Cactus` method addresses sets of blocks, called *chains* and sets of adjacencies called *groups*.

Chains. Chains are sets of blocks that form simple cycles in the cactus graph structure. They do not necessarily represent continuous genome segments, but they represent conserved orders of blocks (for example blocks A, E, I, K in Fig. 4.8 on page 77). Chains appear as a substructure only in the cactus graph structure.

Groups. Groups are sets of adjacencies that form E_A-connected components in the Enredo graph structure. In the publication of the `Cactus` method [121], adjacencies are segments and blocks have at least two occurrences. To account for the difference when blocks with one occurrence are present and all adjacencies are single positions, we include in groups all blocks with one occurrence along chains that start incident to an adjacency of the E_A-connected component. Note that in the original definition, each group is represented by one vertex of the cactus graph, but a vertex of the cactus graph can represent several groups. In the strict sense, identification of groups is only possible in Enredo graph structures but not in cactus graph structures without labels.

4.4 Alignment modifications

This part of the chapter describes operations that modify the set of blocks. Through modifications, genome alignment approaches decide which of the local alignments are kept for the final genome alignment. Along with this selection of local alignments, the modifications determine breakpoint positions and the resulting genome segmentation.

The following sections define the splitting and merging of blocks as well as the segmentation of genomes. These modifications are small operation entities from which more complex operations like the “joining” or “annealing” operation in the `Enredo` method can be assembled. In addition to the definitions on the set of blocks, the sections describe the effects that the modifications have on the four graph structures. Generally, the aim is to eliminate visiting paths and small cycles.

The algorithmic goal of alignment modifications is typically to maximize both the size and length of colinear paths. This part of the chapter does not address algorithms that find the best trade-off between the sizes and lengths of colinear paths. Nevertheless, we acknowledge that the choice

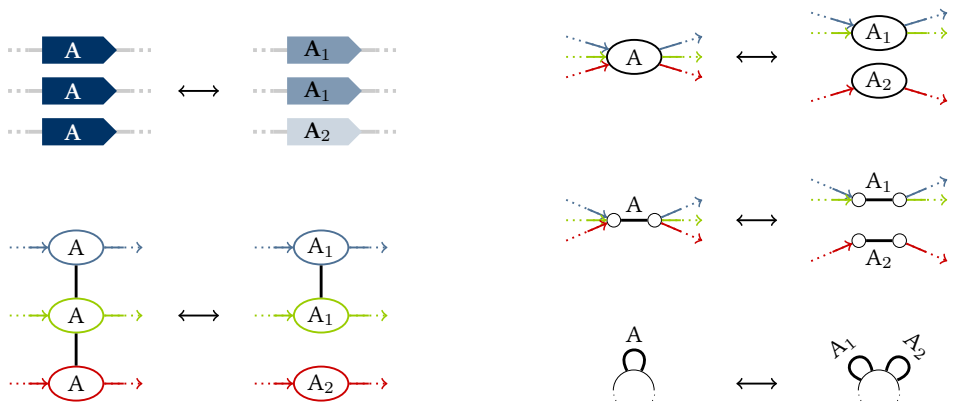

Figure 4.20: An example for the reciprocal modifications of merging and splitting blocks. A block A with three occurrences is split into two blocks A_1 and A_2 with two and one occurrences, respectively (top left). The two blocks A_1 and A_2 can again be merged to form the block A. In addition, the figure displays the effect in an alignment graph (bottom left), in an A-Bruijn graph (top right), in an Enredo graph (middle right), and the simplest effect in a cactus graph (bottom right).

of blocks and adjacencies for modification and the order of applying the modifications is not a trivial task and crucial for obtaining accurate genome alignments.

4.4.1 Splitting blocks

Both visiting paths and short cycles can be eliminated by splitting blocks into several blocks of smaller size. With the goal of creating longer colinear paths, `ABA` and `Enredo` eliminate cycles and visiting paths from the graph structures by partitioning the set of segments of one block into two blocks. `DRIMM-Synteny` splits only single segments from a block for eliminating visiting paths and cycles. Most rigorous is the "melting" operation in `Cactus` that simultaneously splits all blocks along a chain into single segments.

Definition of block splitting. A single splitting modification divides a larger block into two smaller blocks. Let $B \in \mathcal{B}$ be a block of size $n \geq 2$. The modification replaces the block $B = \{s_1, \ldots, s_n\}$ in the set of blocks $\mathcal{B}$ by two new blocks $B_1 = \{s_1, \ldots, s_k\}$ and $B_2 = \{s_{k+1}, \ldots, s_n\}$ of sizes k and $n - k$ where $1 \leq k < n$. This corresponds to the removal of alignments between the segments $s_1, \ldots, s_k$ and the segments $s_{k+1}, \ldots, s_n$.

Effect on the graph structures. Figure 4.20 shows an example for the effect of splitting a block in all four graph structures. Splitting blocks has the smallest effect on alignment graph structures. The modification simply removes a subset of the block edges from one E_B-connected component such that it decomposes into two components.

The effect on A-Bruijn graph structures is very similar to the effect on Enredo graph structures.

In A-Bruijn graph structures, the modification replaces the vertex representing the split block by two new vertices. The incoming and outgoing edges are assigned to the two new vertices according to the adjacencies of the segments in the new blocks. In Enredo graph structures, the modification replaces a pair of head and tail vertices connected by a block edge by two head and tail vertex pairs each connected by a block edge. Each adjacency edge incident to the original tail vertex is reconnected to one of the new tail vertices and each adjacency edge incident to the original head vertex is reconnected to one of the new head vertices.

In cactus graphs, the splitting modification can lead to complex rearrangements, where a vertex is replaced by two new vertices or two vertices are merged to a single vertex. For example, splitting a block can create a 3-edge connected component, whose vertices are subsequently merged. Fig. 4.20 shows only the simple case where the modification replaces one block edge by two new block edges.

4.4.2 Merging parallel blocks

Specific cycles can be eliminated by merging parallel blocks to form only a single block. This is the reverse modification to splitting blocks. The A-Bruijn graph approaches `ABA` and `DRIMM-Synteny` apply this modification to cycles where edges in both directions are present. But also the `Enredo` method and `Cactus` merge parallel blocks and term the modification "annealing".

Definition of parallel block merging. The merging modification for parallel blocks joins the sets of segments of two blocks. Let $B_1, B_2 \in \mathcal{B}$ be two blocks of size k and $n-k$ where $1 \leq k < n$. The modification replaces the two blocks $B_1 = \{s_1, \ldots, s_k\}$ and $B_2 = \{s_{k+1}, \ldots, s_n\}$ by a new block $B = \{s_1, \ldots, s_n\}$ of size n. This corresponds to adding alignments between the segments $s_1, \ldots, s_k$ and the segments $s_{k+1}, \ldots, s_n$.

Effect on the graph structures. Figure 4.20 illustrates that merging parallel blocks has the reverse effect of splitting blocks. In alignment graph structures, the modification adds at least one block edge between two different E_B-connected components. In A-Bruijn graph structures, the modification replaces two vertices by a single vertex. In Enredo graph structures, the modification replaces two pairs of head and tail vertices connected by block edges by a single pair of head and tail vertices connected by a block edge. In both A-Bruijn and Enredo graph structures the incoming and outgoing edges of the replaced vertices are reconnected to the new vertices. In cactus graphs, merging parallel blocks can lead to complex rearrangements analogously to splitting blocks.

4.4.3 Merging adjacent blocks

The elimination of cycles and visiting paths creates longer colinear paths. Often, these paths of adjacent blocks can be merged to a single block in order to reduce the complexity of the set of

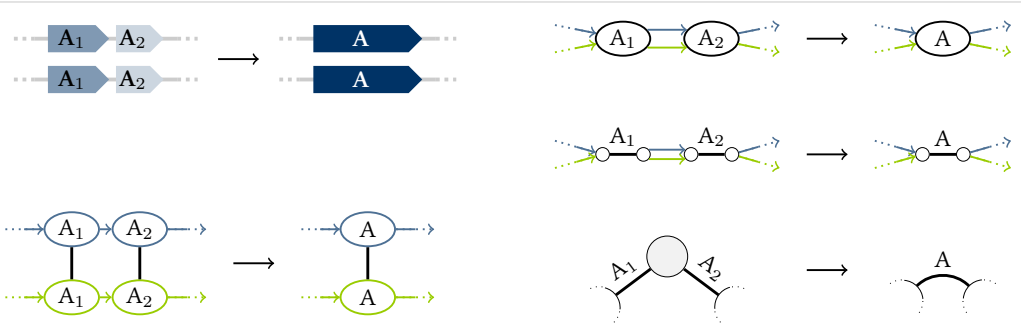

Figure 4.21: An example for the merging modification of adjacent blocks. Two blocks A_1 and A_2 that are adjacent without breakpoint are merged into a single block A with the same number of occurrences (top left). In addition, the figure displays the effect in an alignment graph (bottom left), in an A-Bruijn graph (top right), in an Enredo graph (middle right), and in a cactus graph (bottom right).

blocks. As a result, all maximal colinear paths become single blocks. The `Enredo` method applies this modification as part of the "joining" operation.

Definition of adjacent block merging. The merging modification for adjacent blocks joins the segments from the sets of two blocks. Let $B_1, B_2 \in \mathcal{B}$ be two blocks that are adjacent without breakpoint. The modification replaces the two blocks $B_1 = \{r_1, \ldots, r_n\}$ and $B_2 = \{t_1, \ldots, t_n\}$ by a new block $B = \{s_1, \ldots, s_n\}$ of the same size where s_i is the concatenation of the adjacent segments r_i and t_i for all $i = 1, \ldots, n$. The modification has no direct effect on the genome alignments but it prevents the creation of breakpoints between the two blocks.

Effect on the graph structures. Figure 4.21 shows an example for the effect of merging two adjacent blocks in all four graph structures. In alignment, A-Bruijn, and Enredo graph structures, the modification removes n directed edges, where n is the size of the merged blocks. In the alignment and A-Bruijn graph structures the endpoints of the removed edges are replaced by a single vertex. In the Enredo graph structure, the removed edges connect two end vertices each incident to a block edge. These two end vertices and block edges are replaced by a single block edge. In the cactus graph structure, the modification replaces a vertex incident to two edges by a single edge that represents the new block.

4.4.4 Genome segmentation

The segmentation modification affects the set of genomes rather than the set of blocks by fixing breakpoints in the final genome alignment. Using genome segmentation, `ABA` and the `Enredo` method remove small cycles. In addition, the `Enredo` method achieves segmentation by excluding long adjacencies from the beginning.

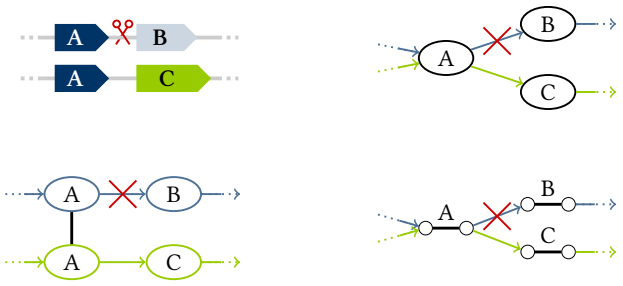

Figure 4.22: An example for the genome segmentation modification. The modification cuts the genomes at the adjacency position of two blocks A and B that are adjacent with breakpoint (top left). The figure also displays the effect in an alignment graph (bottom left), in an A-Bruijn graph (top right), and in an Enredo graph (bottom right).

The final genome segmentation process corresponds to applying the modification to all breakpoints at once. Instead of modifying the graph structures, we may derive the final segmentation directly from the graphs: After merging all blocks that are adjacent without breakpoint, the labels of all vertices or edges that represent blocks form the final segmentation.

Definition of the segmentation modification. The segmentation modification cuts the genomes $\mathcal{G}$ into segments. Given two adjacent segments $s_1 \in B_1$ and $s_2 \in B_2$ from two blocks $B_1, B_2 \in \mathcal{B}$ that are adjacent with breakpoint. Let $a = i_2 = i_1 + l_1$ be the adjacency position of the two segments $s_1 = (i_1, l_1, o_1)$ and $s_2 = (i_2, l_2, o_2)$. The modification cuts the corresponding genome $g \in \mathcal{G}$ at the position a into a segment of length i_2 and a segment of length $|g| - i_2$. The modification has no direct effect on the set of blocks but it fixes a breakpoint in the genome alignment.

Effect on the graph structures. Figure 4.22 shows an example for the effect of cutting adjacencies in the alignment, A-Bruijn, and Enredo graph structures. In all three graph structures, the modification removes a single directed edge. Segmentation may decompose the graph structures into several connected components. In cactus graphs, the effect may be hidden in a vertex if we do not connect the loose ends to the origin vertex. Connection to the origin vertex can again lead to complex rearrangements of the cactus graph.

4.5 Conclusion to the chapter

This part concludes the chapter with a summary, discussion, and outlook. The discussion addresses points that are not covered by the comparison and describes the benefits from representing genome alignments in graph structures. The outlook mentions more steps in genome alignment that this chapter did not cover and suggests a combination of the graph structures.

Summary. In the first part of this chapter, the clear distinction between graph structures and graph models allows to evaluate to what extent alignment graph structures, A-Bruijn graph structures, Enredo graph structures, and cactus graph structures model non-colinearity of a genome alignment. Alignment and A-Bruijn graph structures lack information about the orientation of segments, thus are less suited to detect inversion in the alignments. In A-Bruijn and Enredo graph structures, labels are necessary to reconstruct the sequence of blocks since they represent duplications and translocations ambiguously. Cactus graph structures are even ambiguous for duplications, translocations, and inversions but they subdivide the genomes into units that none of the other structures exhibit.

Transformations among the graph structures confirm the differences in information content. Ambiguity occurs in the transformations from A-Bruin to Enredo graph structures, from A-Bruijn to alignment graph structures, and from cactus to Enredo graph structures. Such ambiguities can be resolved with sparse labels that provide the missing information.

Substructures defined on the set of blocks often do not appear in the graph structures as subgraphs of certain topology. Labels are necessary to identify many substructures in most graph structures. Only Enredo graphs model colinear paths in their structure. Visiting paths cannot be identified in any of the four structures without labels. Cycles in the set of blocks mostly appear as cycles in all graph structures, but several cycles in the graph structures may represent a single cycle in the set of blocks.

Substructures indicate inconsistencies in the alignments that can be resolved by applying modifications, operations that change the set of blocks or segment the genomes. Modifications that split blocks and merge parallel and adjacent blocks have analogous effects on alignment, A-Bruijn and Enredo graph structures. In addition, genome segmentation has a clear correspondence in these structures. In contrast, simple modifications to the set of blocks can lead to complex rearrangements of the cactus graph structure.

Discussion. The aim of this chapter was to formally evaluate and compare the graph structures and graph-based approaches in a consistent way. For that matter, special features of individual graphs received less attention although they are worth mentioning and often very favorable. In addition, the list of compared graph structures and sets of transformations, substructures, and modifications are certainly not exhaustive. For the four selected graph structures, the chapter examines only one of several possible formal definition.

For example, the first part of this chapter defines A-Bruijn graphs with one vertex per block. Pevzner et al. however briefly mention in the first A-Bruijn graph publication a second vertex representing the reverse complement of each block. Furthermore, they "analyze both vertices of such pairs at the same time" [131]. One may consider the pairs of vertices to be connected by a second type of edges. The resulting graph structure has two vertices and one edge per block like the Enredo graph structure, but with a substantial difference: The two adjacencies of each block are represented by directed edges that are incident to the same vertex (see Fig. 4.23).

In addition, the A-Bruijn graph approaches address cycles not in general but classify them into several types of cycles. The first classification was introduced for repeat classification and then

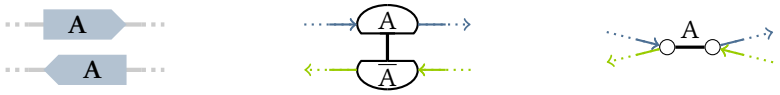

Figure 4.23: Two occurrences of a block A in opposite orientation as genome segments (left), alternative A-Bruijn graph structure with two vertices per block (middle), and Enredo graph structure (right).

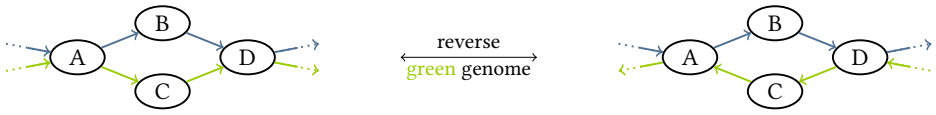

Figure 4.24: A cycle with edges in both directions (left) called "bulge" in [131] and a directed cycle (right) called "whirl" in [131] both formed by two segments from different genomes. When reverse complementing the green genome, the bulge becomes the whirl and vice versa.

used in `ABA` [133]. It classifies directed and undirected cycles ("whirls" and "bulges"). The direction, however, depends on the orientation of the genomes (see Fig. 4.24). Subsequently, Pham et al. [132] refined the classification in `DRIMM-Synteny` into "one-way", "two-way", and "composite" cycles. This classification is more detailed than the description of cycles in Section 4.3.3.

Independent from a specific graph structure, the chapter makes assumptions on the set of blocks that are not necessarily consistent with the output of local alignment tools. Most commonly, only pairwise and not multiple local alignments are given. Combining pairwise alignments to multiple alignments is not a trivial task since pairwise alignments can have conflicting gap patterns. We can avoid this problem and ignore gap patterns at this stage if a colinear realignment is carried out for each block after finishing the segmentation.

The analysis of the cactus graph structure demonstrates noticeable differences to the other structures with respect to the effects of modifications as well as the appearance of non-colinear changes and substructures. The cactus graph structure comprises less information about the adjacencies of blocks in the genomes. On the one hand, the cactus graph structure completely hides this information, but on the other hand, cactus chains are not visible in any of the other graphs. The cactus graph can be viewed as a structure with additional properties on top of the Enredo graph. Indeed, the `Cactus` method operates on two structures, the cactus graph structure and an "adjacency graph" that is essentially an Enredo graph.

The benefits of cactus graphs become intelligible when analyzing the biological meaning of the algorithmic steps in the transformation of Enredo graphs to cactus graphs. For example, the computation of 3-edge connected components seems arbitrary at first, but it reveals parts of a genome alignment that split into several alternatives (similar to "bulges" in A-Bruijn graphs). In cactus chains, only blocks are present that always occur in a conserved order. Thus, 3-edge connected components are being removed from chains by merging them into a single vertex, which creates separate chains for the alternatives.

The example of 3-edge connected components demonstrates the usefulness of representing genome alignments in graph structures: The genome alignment approach benefits from existing graph theoretical algorithms. Similarly, Pevzner et al. [131, 133] formulate the problem of A-Bruijn graph-based genome alignment as the maximum subgraph with large girth (MSLG) problem, a generalization of the maximum spanning tree problem [33].

Apart from the algorithmic benefits, graph structures established themselves also conceptually for genome alignment. The `Enredo` method mainly relies on the graph as a convenient data structure without resorting to standard graph algorithms. Furthermore, the `Cactus` method selects subsets of local alignments that results in the most favorable graph structure using the cactus alignment filter (CAF) algorithm. Although Paten et al. [121] formulate their objective as the maximum weight cactus subgraph with large chains (MSLC) problem, they search for a subset of the initial set of local alignment rather than for a subgraph of the initial cactus graph structure.

Outlook. As mentioned in the discussion, the evaluation of graph representations for genome alignments can be extended by more graphs, transformations, substructures, and modifications. Apart from this, a number of steps of graph-based genome alignment has not been addressed in this chapter at all. While 4.3 assesses whether identification of substructures in the graph structures is possible or not, it does not mention algorithms and corresponding complexities to identify them. Furthermore, algorithms that determine the order of modifications have the potential to greatly influence genome alignment accuracy. The mentioned approaches mostly apply iterative algorithms with various strategies and end criteria. However, a detailed analysis of the algorithms is beyond the scope of this chapter and left for future work.

Another important task will be to give advice or even automate the selection of customized parameter values. Several parameters are mentioned above, for example the length threshold for short cycles. Such parameters influence the trade-off between length and size of blocks in the final genome alignment. A good choice of parameter values typically depends on the input data, for example the similarity of the genomes. In very similar genomes we can expect much longer and larger blocks than in more diverged genomes although homology can exist at various scales [91]. A recent approach takes the trade-off between size and length of blocks one step further: The tool `Sibelia` [109] builds a hierarchy of blocks at different resolutions and uses another graph structure called “iterative de Bruijn graphs”.

As a result from the comparison in this chapter, a combination of the alignment and Enredo graph structures suggests itself: A graph structure with two vertices per block occurrence and three types of edges. This graph structure might be interesting for future genome alignment representations since it models duplications, translocations, and inversions and allows to unambiguously identify colinear and visiting paths without labels. However, it is much larger than, for example, the compact A-Bruijn graph structure. Furthermore, we may distinguish two sets of edges by their directedness (block and adjacency edges in alignment and Enredo graphs) while one might understand three types of edges already as labeled edges. Eventually, the most appropriate graph structure depends on the application and specific objective.

Chapter 5

Rearrangement breakpoints in multiple genome alignments

The study described in this chapter was carried out in collaboration with Aaron Darling and Knut Reinert, presented at the Workshop on Algorithms in Bioinformatics (WABI) 2012, and published in the conference proceedings [87]:

B. Kehr, K. Reinert, and A. E. Darling. Hidden breakpoints in genome alignments. In B. Raphael and J. Tang, editors, *Algorithms in Bioinformatics*, volume 7534 of *Lecture Notes in Computer Science*, pages 391–403. Springer Berlin Heidelberg, 2012

This chapter proposes a concept for rearrangement breakpoints in the comparison of multiple genomes. Breakpoints are the evidence for non-colinear rearrangement events that happened since the divergence of genomes from a common ancestral genome. The number of breakpoints among pairs of genomes often serves as an estimate of the evolutionary distance between species [114]. Similarly, scoring functions for selecting local alignments in methods for computing genome alignments use pairwise breakpoint counts [37]. However, some breakpoints of rearrangement events are not visible in the comparison of two genomes when sequence segments were lost during evolution or when the resolution of the alignment is low with missing blocks. By taking into account multiple genomes, identification of such hidden breakpoints becomes possible.

This chapter demonstrates the existence of hidden breakpoints in alignments of multiple genomes. The first part introduces relevant background from the field of genome rearrangement. The second part presents the concept of hidden breakpoints and a counting method for sets of three genomes. The third part evaluates the concept by comparing the counts of pairwise and

hidden breakpoints in a large number of simulated and calculated alignments. Finally, the last part concludes the chapter with a summary, a discussion of both the potential and limitations of hidden breakpoints, and an outlook on future research.

5.1 Background on genome rearrangement

The question for the breakpoint-based evolutionary distance of two genomes is only one of many problems studied in the field of genome rearrangement. Further problems in this field ask for the sequence arrangement of ancestral genomes or even the precise evolutionary changes that led to extant genomes. This part of the chapter only introduces the modeling of genomes used in all these problems (Section 5.1.1) and types of rearrangement distance measures (Section 5.1.2). Furthermore, Section 5.1.3 addresses the re-use of breakpoints and its influence on the distance measures. Finally, Section 5.1.4 describes in more detail how rearrangement distance measures integrate into methods for computing genome alignments. For a comprehensive overview of genome rearrangement problems see for example the book by Fertin *et al.* [57].

5.1.1 Modeling genomes for rearrangement analysis

Studies of genome rearrangement examine genomes on the level of blocks. The input to these studies is a set of blocks and a set of genomes modeled as sequences of the blocks. Often, blocks are assumed to be genes, and therefore functional units of the genomes. We leave the biological function of blocks open and only assume occurrences of the same blocks to be generally homologous.

The following three parameters further specify the model of genomes as sequences of blocks. The choices for these parameters can be arbitrarily combined to define a model of genomes.

- *Number of chromosomes:* A model distinguishes unichromosomal from multichromosomal genomes. Unichromosomal genomes have only one chromosome, while multichromosomal genomes have an arbitrary number of chromosomes.
- *Shape of the chromosomes:* A model either requires all chromosomes to be linear, or requires all chromosomes to be circular, or allows both linear and circular chromosomes in the genomes.
- *Copy number of blocks per genome:* A model differentiates between genomes with or without gain/loss of blocks and with or without duplication of blocks. In genomes without gain/loss and without duplications, each block occurs exactly once. In the presence of gain/loss but without duplications, each block occurs at most once per genome. If both gain/loss and duplications are present, the blocks occur in arbitrary copy numbers per genome.

Depending on the genomes under comparison, a model is appropriate that is more or less restrictive with respect to these parameters. A widely studied model of genomes in rearrangement studies are *signed permutations* of blocks (where signs indicate the orientation of blocks). Signed

permutations correspond to a model of unichromosomal genomes with linear chromosomes and exactly one occurrence of each block per genome. A slightly less restrictive but well-recognized model is the *Hannenhalli-Pevzner (HP) model* [72, 73]. The HP model allows for multichromosomal genomes but also requires linear chromosomes and exactly one block occurrence per genome. If we allow the chromosomes to be linear or circular in addition, we obtain the model of genomes underlying the basic *double-cut and join (DCJ) distance* [17, 168]. Finally, the least restrictive model allows for multichromosomal genomes with circular and linear chromosomes and arbitrary copy numbers of block occurrences per genome.

The choice of model parameters does not only make strong biological assumptions, but also has an influence on the theoretical complexity of a rearrangement problem. Problems that are NP-hard in one model, are sometimes solvable in polynomial time in another model [153]. Models with gain/loss and duplications usually make the problems more challenging. For this reason, more restrictive models without gain/loss or without duplications are often employed, even though the genomes have varying copy numbers of blocks.

If a model restricts the copy number of blocks per genome, pre-processing of the genomes is necessary to exclude gain/loss or duplications. A frequently-used solution for avoiding gain/loss is projecting the genomes to the set of common blocks and thereby excluding blocks from the genomes that are unique to a subset of the genomes. Excluding all but one copy of a duplicated block is more challenging since the position of the chosen copy affects the distance to other genomes. As a possible solution, Sankoff suggested the calculation of an exemplar genome [144], which chooses the copy that best reflects the block's position in a common ancestor of the genomes. However, the exclusion of block occurrences from the genomes is known to degrade the comparison and leads to breakpoint re-use (see Section 5.1.3).

5.1.2 Rearrangement distance measures

There are two significantly different types of rearrangement distance measures for pairs of genomes. The breakpoint distance is a solely descriptive distance measure. Other distance measures are based on assumptions on the mode of evolution and count the minimum number of operations necessary to transform one genome into the other. In the following we briefly introduce and discuss both the breakpoint distance and operation-based distance measures.

Breakpoint distance measure. The breakpoint distance describes only visible differences of genomes and makes no additional assumptions on the mode of genome evolution. Thus, it is an only descriptive distance measure.

Formally, the *breakpoint distance* $d(g_1, g_2)$ counts the number of breakpoints in one genome g_1 with respect to another genome g_2. Thus, it counts the number of adjacencies present in one genome that are not conserved in the other. The idea is that each breakpoint corresponds to a genome breakage event that happened during evolution.

Already in 1984, Nadeau and Taylor used the number of "disruptions", by which they refer to breakpoints, to calculate a rate of chromosomal evolution [114]. Because the breakpoint distance

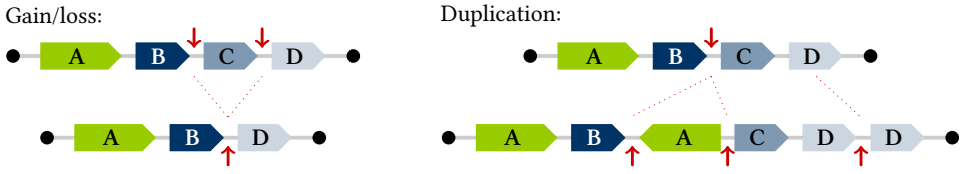

Figure 5.1: The asymmetric effect of gain/loss and duplications on the breakpoint distance. Red arrows indicate breakpoints (adjacencies that are not present in the other genome). On the left, block C is unique to the upper genome and results in an extra breakpoint in this genome. On the right, the dispersed duplication of block A and the tandem duplication of block D each result in an extra breakpoint in the lower genome.

is simple but meaningful, it is used until today in phylogenetic studies, for example in [18].

The breakpoint distance is only symmetric if all blocks occur exactly once per genome. In the presence of gain/loss or duplications, extra breakpoints appear in that genome that has the additional block occurrences. Figure 5.1 displays examples for this asymmetry. To avoid asymmetry, the breakpoint distance is usually computed only for genomes without duplications and for the projections of these genomes to the set of blocks common to both of them.

Operation-based measures. The other type of distance measures counts operations in evolutionary scenarios of two genomes. A *scenario* is a sequence of operations that transforms one genome into another. Generally, the length (number of operations) of the shortest scenario is used as the operation-based distance of two genomes.

Underlying this type of distance measure are assumptions about the operations by which the genomes have evolved. For example, the *reversal distance* [74, 84, 163] assumes that genomes evolve through inversions of one or several consecutive blocks. Further, the *translocation distance* [71, 85] counts operations by which sets of consecutive blocks at the ends of genomes are exchanged. Being less restrictive, the *double-cut and join (DCJ) distance* [17, 168] allows for all operations that involve two breakpoints, which includes inversions, translocations, chromosome circularization and more.

Operation-based distance measures rely on the correctness of the assumptions about the operations. They limit the set of operations by which genomes can evolve. However, new mechanisms of genome evolution are still being discovered. For example, none of the operation-based measures accounts for the recently described process of chromothripsis [152] that leads to massive rearrangement of a chromosome in one event. Thus, operation-based measures could be too restrictive for general estimations of evolutionary distances [145].

5.1.3 Re-use of breakpoints

When several evolutionary events interrupt colinearity of two genomes at the same breakpoint, this is called *breakpoint re-use*. Breakpoint re-use does not imply that a genome broke twice at

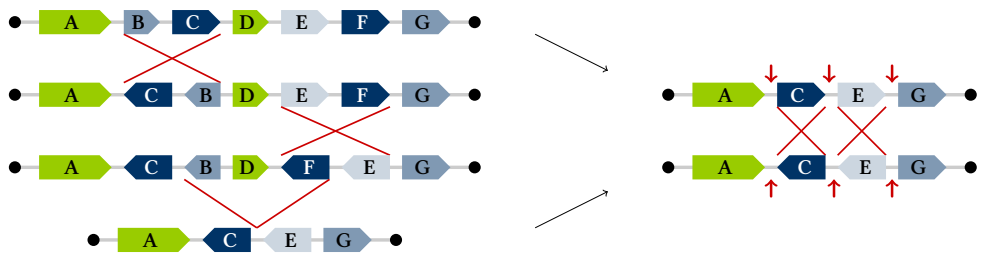

Figure 5.2: *breakpoint re-use* Breakpoint re-use caused by loss of three blocks occurrences. The left hand side shows the evolution of a genome through two inversion events and one loss event from an initial genome with seven blocks to a final genome with four blocks. The right hand side shows the projection of the initial and final genome to common blocks. Although two inversion events occurred resulting in four breakpoints, the projection reveals only three breakpoints (red arrows). The breakpoint between blocks C and E seems to be due to both inversion events.

the exact same genomic position. Despite distinct breakage positions, several breakage events can appear as a single breakpoint.

This effect primarily arises from modeling genomes as sequences of blocks. On the one hand, the resolution of the blocks affects how close breakpoints approximate the breakage positions. On the other hand the projection to only blocks common to both genomes leads to coinciding breakpoints. The following three paragraphs describe both these causes of breakpoint re-use in more detail.

The resolution of blocks depends on the precision with which similarities were detected. If only highly significant local alignments are defined as blocks, conservation and rearrangement events of shorter segments remain invisible. By definition, a breakpoint is a non-conserved adjacency of two blocks, and an adjacency is a possibly non-empty segment. These segments are longer at low resolution of blocks than at high resolution of blocks. Hence, at low resolution small undetected blocks are hidden in adjacencies and breakpoints. Rearrangement events that involve the undetected blocks create the effect of breakpoint re-used.

Low block resolution as a cause of breakpoint re-use was extensively discussed in the literature. For example, Pevzner and Tesler studied the presence of small undetected blocks between the human and mouse genomes [130]. An alignment of mammalian genomes at higher resolution by Ma *et al.* [103] confirmed less breakpoint re-use when small blocks are included in the analysis. Further studies systematically examined the direct influence of alignment resolution on breakpoint re-use [11].

The second known cause of breakpoint re-use is loss of sequence segments. Loss in one genome leads to unique segments in the other genome. As described above, unique segments are usually excluded from the comparison. However, along these segments there can be breakage positions of rearrangement events that happened prior to the loss. These breakage positions create the effect of breakpoint re-use in the projected genomes even at the highest alignment resolution. Figure 5.2 displays an example for breakpoint re-use caused by loss of three blocks.

Breakpoint re-use has an influence on the accuracy of distance estimation with the types of

measures described in Section 5.1.2. The breakpoint distance accounts for each breakpoint only once, and hence underestimates the number of breakage events in the presence of breakpoint re-use. In general, the breakpoint distance is a lower bound to the actual number of breakage events that happened during evolution. Operation-based distance measures can compensate for some breakpoint re-use [130]. However at high rates of breakpoint re-use, distances estimated with operation-based measures come close to distances of random block permutations [145]. Thus, sequences of blocks are not sufficient to reconstruct the evolutionary history of two genomes at high rates of breakpoint re-use [145]. In short, both types of distance measures suffer from breakpoint re-use.

Computation of a rate of breakpoint re-use from both types of rearrangement distance measures recently initiated a controversial discussion in the literature [4, 101, 103, 129, 130, 145, 146]. Subject of the discussion was whether this rate of re-use is appropriate to make implications on the randomness of evolutionary changes. Since the following parts of the chapter are independent from this specific rate, we refer to the literature for details on the controversy.

5.1.4 Rearrangement distances in methods for computing genome alignments

Rearrangement distances assist methods for computing genome alignments in selecting the subset of local alignments. The idea is analogous to applying similarity scores for optimizing colinear alignment (see Section 3.1.1). Similarity scores reward matches with positive scores and apply negative penalties for mismatches and gaps. Thus, they apply penalties for substitutions, insertions, and deletions.

The genome aligner `progressiveMauve` [37] logically extends similarity scoring schemes to non-colinear alignments by adding a negative penalty for non-colinear changes. Rearrangement distances provide the necessary information for this extension: the number of breakpoints or evolutionary changes. This number multiplied by a penalty can guide methods for computing genome alignments in finding the subset of local alignments with the best trade-off between alignment score and genome segmentation.

Specifically, let $\mathcal{A}$ be the set of all local alignments and $S(a)$ be the score of the local alignment $a \in \mathcal{A}$. Colinear alignment methods select the subset $A \subseteq \mathcal{A}$ that maximizes the sum of alignment scores without violating colinearity. Non-colinear alignment methods can replace the requirement for colinearity by a penalty p for non-colinearity:

$$\underset{A \subseteq \mathcal{A}}{\operatorname{argmax}} \; -p \cdot d(A) + \sum_{a \in A} S(a)$$

where $d(A)$ is the rearrangement distance among all genomes given the segmentation induced by the subset A.

Since rearrangement distances are typically defined for pairs of genomes, `progressiveMauve` uses a sum-of-pairs approach to determine $d(A)$ from pairwise breakpoint distances. As mentioned above, the pairwise breakpoint distance is, however, only a lower bound to the actual

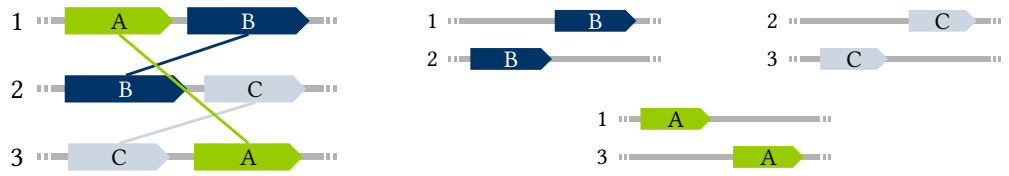

Figure 5.3: An example for a hidden breakpoint in three genomes (left). Only two of the three blocks occur in each of the genomes such that the pairwise projections to common blocks (right) reveal no pairwise breakpoints.

number of breakage events. In addition, multiple genomes contain more information than pairs of genomes for resolving breakpoint re-use [103], and hence allow the actual distance to be approximated more closely. The following presents an extension of the breakpoint distance for sets of more than two genomes that is suited to be applied in scoring functions for genome alignments.

5.2 Hidden rearrangement breakpoints

This part of the chapter introduces the concept of hidden breakpoints (Section 5.2.1) and presents a method for counting them in alignments of three genomes (Section 5.2.2). Hidden breakpoints reveal breakpoint re-use in pairwise comparison of genomes using the information provided by additional genomes. The counting method for hidden breakpoints employs the distance of these multiple genomes to a median genome. The last section of this part (Section 5.2.3) provides details on the calculation of median distances in the breakpoint model.

5.2.1 The concept of hidden breakpoints

Multiple alignments of evolutionarily related sequences achieve higher accuracy in predicting homology than pairwise alignments [140]. When considering a third sequence in the alignment of two sequences, we gain information for resolving ambiguities in the pairwise alignments. The concept of hidden breakpoints exploits this gain of information through a third sequence for extending the breakpoint distance measure.

As described in Section 5.1.3, the breakpoint distance is only a lower bound to the actual number of breakage events that happened during the course of evolution. Through loss of sequence segments and low alignment resolution, several breakage events appear as a single breakpoint in the pairwise comparison of genomes. Looking at more than two genomes at once reveals some of these additional breakage events.

Consider the example of three genomes consisting of three blocks in Fig. 5.3 (see also [36, p. 78f.]). In pairwise comparison of the three genomes, only gain/loss of blocks is visible, but the order of the blocks remains unchanged (in this example only one block is common to each pair of

genomes). Thus, the sum-of-pairs breakpoint distance is 0. However, inference of an order of the three blocks as they could have appeared in an ancestral genome, reveals a circular dependency: block A precedes block B, which precedes block C, which precedes again block A. This cycle becomes visible only when considering all three genomes. It implies a rearrangement event in the evolutionary history of the three genomes.

Figure 5.4 displays possible evolutionary scenarios along a phylogenetic tree of the three genomes. Notably, the order of blocks in the three extant genomes of this example can only be explained with a rearrangement event but not with gain/loss alone. However, the rearrangement event creates no breakpoints visible in the pairwise projections.

The lack of breakpoints in the pairwise projections is caused by breakpoint re-use. The rearrangement event uses the same breakpoints as the gain/loss events, and thus it is hidden in pairwise comparisons.

Although the order of blocks in the ancestral genome is ambiguous and unknown as is the evolutionary scenario, the three-way comparison provides evidence for an additional evolutionary change. This evolutionary changes implies additional breakage events that are hidden through breakpoint re-use. We call these breakage events *hidden breakpoints*.

The number of hidden breakpoints added to the number of pairwise breakpoints approximates the actual number of breakage events more closely. Nevertheless, this distance estimate is still a lower bound: A third genome reveals hidden breakpoints only if it differs from both genomes at the re-used breakpoint.

Comparison to cycles in genome alignment graphs. The representation of the example from Fig. 5.3 in an alignment graph structure, A-Bruijn graph structure, or Enredo graph structure contains a cycle. This cycle uses adjacencies from all three genomes. The example suggests a correspondence between hidden breakpoints and cycles formed by three or more genomes in graph structures. However, Fig. 5.5 provides an example of three genomes with a hidden breakpoint but without a three-way cycle in the graph representations. Furthermore, Fig. 5.6 shows an example without hidden breakpoint but with three-way cycle in the graph representations. Thus, there is no equivalence between three-way cycles and hidden breakpoints.

5.2.2 A median approach for counting hidden breakpoints

This section presents a counting approach for hidden breakpoints among three genomes. The approach leaves the location of hidden breakpoints within the genomes open and only provides a count $\mathcal{H}$. Furthermore, the approach provides no information about the distribution of the hidden breakpoints over the branches of the tree. The count $\mathcal{H}$ is the total number of breakpoints for all branches of the phylogenetic tree of three genomes.

The counting approach is based on the breakpoint distance of the input genomes to a median genome. In median genomes the blocks are arranged such as to minimize the total distance to all genomes. Median genomes are used, for example, to predict the order of blocks in ancestral genomes [21].

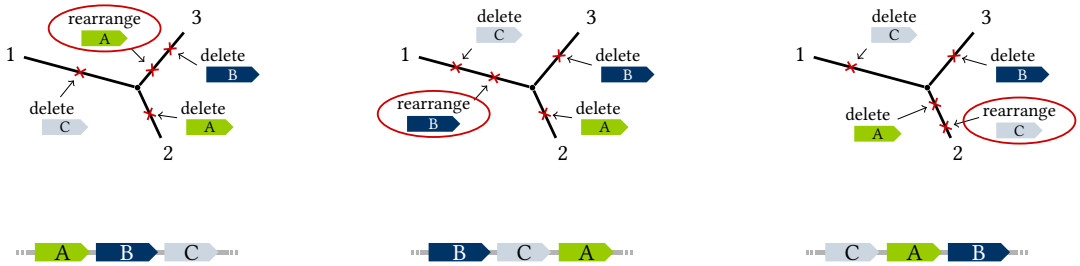

Figure 5.4: Three possible evolutionary scenarios (top) for the example from Fig. 5.3 and ancestral genomes (bottom) that label the inner vertices of the trees. In each scenario there is a rearrangement event in addition to gain/loss events of all blocks. Note that the displayed ancestral genomes may be a common ancestor of only two of the three genomes assuming that the root of the trees is along one of the three branches.

Figure 5.5: A hidden breakpoint in three genomes (left) and a possible evolutionary scenario (right). The rearrangement event is visible in the pairwise comparison of genomes A and B. It happened either on the branch from genome 1 to the inner vertex or on the branch from genome 2 to the inner vertex. Thus, either in the comparison of genomes 1 and 3 or in the comparison of genomes 2 and 3, an additional rearrangement breakpoint is hidden by the loss of block B.

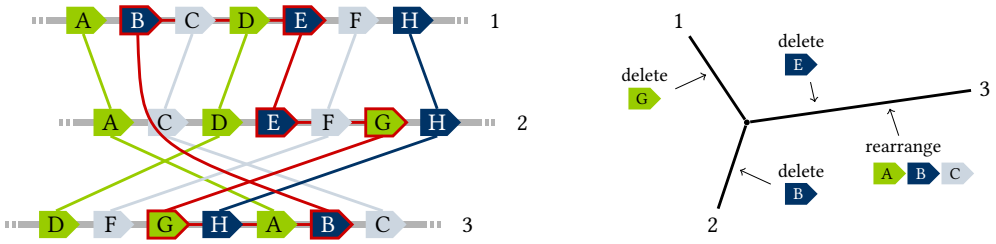

Figure 5.6: Three genomes without hidden breakpoint (left) that result in a three-way cycle (red) in graph representations. Each of the four evolutionary events in the scenario (right) uses different breakpoints. The cycle appears due to both loss and rearrangement of block B. However, the rearrangement is embedded in block A and C. Hence, it creates no hidden breakpoint but breakpoints visible in pairwise comparisons.

Usually, median genomes are computed from only blocks present in all genomes and, thus, provide no information about gain/loss. The counting approach presented here is founded on the premise that the median genome contains any block present in at least two of the three genomes. Blocks unique to a single genome are assumed not to be present in the median. With blocks present in at least two of the three genomes, the median genome preserves information about some gain/loss events and reveals hidden breakpoints.

More formally, given a set of blocks B and a genome g, let $g(B)$ be the projection of g to blocks present in B. Further, let $d_B(g_1, g_2) := d(g_1(B), g_2(B))$ be the breakpoint distance of two genomes g_1 and g_2 projected to the same set of blocks B. Then, a generalized definition of a median genome is given as follows:

Definition 2 (Median genome). *A* median genome M *for* n *genomes* $g_1, \ldots, g_n$ *is a permutation of all blocks in a given set* B *such that* M *minimizes the sum*

$$d_M := \sum_{k=1}^{n} d_B(g_k, M) \ .$$

The counting approach uses the median of three genomes g_1, g_2, and g_3 with the set of blocks present in at least two of them. More precisely, the counting approach uses the distance to the median d_M, but not the median genome itself. Many median genomes with the same minimal d_M may exist. The hidden breakpoint count $\mathcal{H}$ is independent from the permutation of blocks in the median genome and also from the number of median genomes that exist.

In the remainder of this section, let B be the set of blocks present in at least two of the three genomes g_1, g_2, and g_3. Furthermore, let the distance to the median d_M of the three genomes be given. Section 5.2.3 describes an efficient algorithm for calculating d_M.

The three distances $d_B(g_k, M)$, where $k = 1, 2, 3$, that sum up to d_M take into account all blocks in B. This includes blocks that are not present in g_k but in the other two genomes. As a result, the distance d_M accounts for breakpoints of some gain/loss events and not only for breakpoints of rearrangement events among the three genomes g_1, g_2, and g_3. However, gain/loss creates asymmetry in the breakpoint distance (see Fig. 5.1), and hence these breakpoints are usually avoided through projection.

In order to remove breakpoints of gain/loss events, the counting approach subtracts the fraction f that is attributed to pairwise breakpoints from d_M. The below definition of f includes breakpoints of gain/loss and breakpoints of rearrangement events. What remains is the number of hidden breakpoints $\mathcal{H}$.

Let $M_{k,l}$ with $k, l \in \{1, 2, 3\}$ be the permutation of all blocks in B that minimizes the distance to g_k and g_l. In other words, if $k = 1$ and $l = 2$, $M_{1,2}$ is the median of the two genomes g_1 and g_2, which is aware of blocks in g_3. Then, the fraction f of the distance d_M that is attributed to pairwise breakpoints is

$$f := \frac{1}{2} \sum_{k<l \in \{1,2,3\}} d_B(g_k, M_{k,l}) + d_B(g_l, M_{k,l}) \ .$$

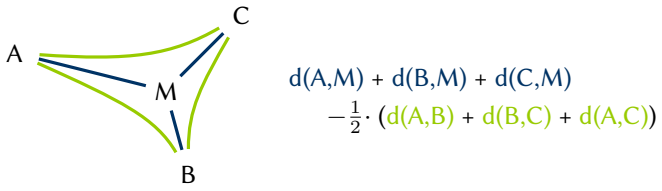

Figure 5.7: Summary of the hidden breakpoint counting approach. Given three genomes, the counting approach computes the distance to a median genome (dark blue) and the distances between the pairs of genomes (green) for the set of blocks common to at least two of the three genomes. The sum of the pairwise distances accounts for each branch of the tree twice. When removing half the pairwise distances from the median distance, the number of breakpoints hidden in pairwise projections remains.

The sum runs over the three possible pairs of genomes in the three-way comparison. Even though the pairwise medians are possibly different and possibly different from M, the sum counts the distance from each genome to a median exactly twice (see Fig. 5.7). Thus, multiplication by $\frac{1}{2}$ is required to set f in relation to d_M.

Having calculated all pairwise distances and the distance to the median, the hidden breakpoint count is

$$\mathcal{H} := d_M - f \ .$$

Figure 5.7 illustrates the simplicity of this approach.

5.2.3 Computation of the median distance in the breakpoint model

For practical usage of the hidden breakpoint count, an efficient algorithm for computing the distances to the medians is necessary. For most distance measures and genome models, including the breakpoint distance measure with only linear chromosomes, the median problem is NP-hard. However, for the breakpoint distance and a more general model with genomes consisting of multiple, possibly circular chromosomes, Tannier *et al.* recently obtained a polynomial result [153, Theorem 1]. Their proof provides a polynomial time algorithm for the breakpoint median problem for multichromosomal genomes.

The computation of median distances for calculating counts of hidden breakpoints follows the proof by Tannier *et al.* [153]. In short, the algorithm constructs a weighted graph from the set of genomes and computes a maximum weight perfect matching (see below). The matching corresponds to a median genome, and the weight of the graph reduced by the weight of the matching corresponds to the median distance.

The following describes in detail the graph construction, explains how a the median distance derives from a maximum weight perfect matching, and suggests a graph reduction. In the graph described by Tannier *et al.* [153] the number of edges is quadratic in the number of vertices. This slows down the computation of a maximum weight matching and is prohibitive for large-scale

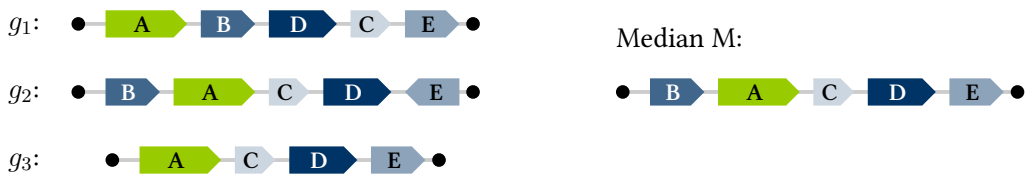

Figure 5.8: An instance of the median problem with three genomes and a set of five blocks. The left hand side shows three genomes projected to the five blocks occurring in at least two of the three genomes. Each of these blocks occurs once in median genomes, as for example in the median shown on the right.

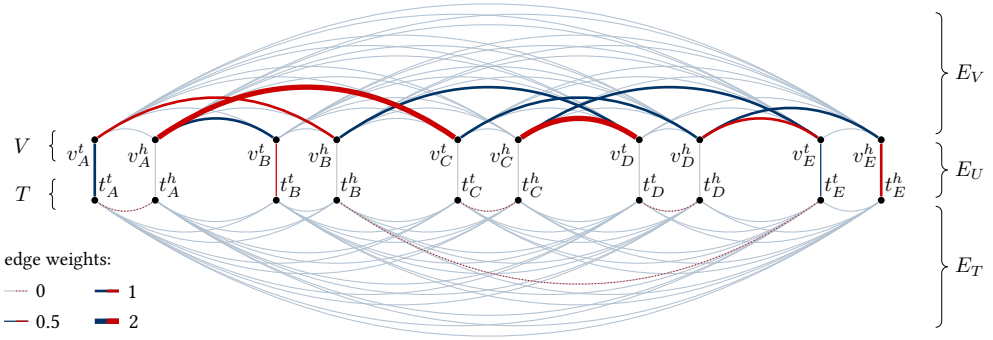

Figure 5.9: The graph for the median instance from Fig. 5.8. The graph has two sets of vertices V and T and three sets of edges E_V, E_U, and E_T. Line widths denote edge weights. Red edges form a maximum weight perfect matching, which corresponds to the median genome shown in Fig. 5.8.

hidden breakpoint counting. The graph reduction cuts down the number of edges by a linear factor.

Another limitation of the algorithm by Tannier *et al.* [153] is the model of genomes: multichromosomal genomes with both linear and circular chromosomes but without gain/loss and duplications. Including gain/loss as does Definition 2, has an influence on the input set of blocks for graph construction, but is otherwise consistent with the algorithm. In contrast, duplications change the underlying matching problem. This section ends with considerations for genomes with duplications.

Graph construction from blocks and genomes. Using the generalized definition of a median, the input to the median problem is a set of blocks B and a set of genomes $g_1, \ldots, g_n$ (see Fig. 5.8 for an example). The blocks define the vertices and the genomes define the edges and edge weights of a weighted undirected graph $G = (V \cup T, E_V \cup E_T \cup E_U)$. The graph G has two sets of vertices V and T, and three sets of edges E_V, E_T, and E_U. Edges in E_V connect vertices from V, edges in E_T connect vertices from T, and edges in E_U connect vertices from V with vertices from T (see Fig. 5.9 and [153]).

The graph has four vertices for each block $b_i \in B$: A head vertex $v_i^h \in V$, a tail vertex $v_i^t \in V$, a head telomere vertex $t_i^h \in T$, and a tail telomere vertex $t_i^t \in T$. The head and tail vertices represent the two ends of a block b_i, and the telomere vertices represent the possibilities that chromosomes start or end with b_i. Because median genomes can consist of arbitrarily many chromosomes, the four vertices per block, in particular separate telomere vertices for all blocks, are necessary.

Edges connect vertices of V and T such that the subgraphs $G_V = (V, E_V)$ and $G_T = (T, E_T)$ are complete. That is, all pairs of vertices $u, v \in V$ are connected by edges $e_V \in E_V$ and all pairs of vertices $s, t \in T$ are connected by edges $e_T \in E_T$. In addition, edges $e_U \in E_U$ connect each head vertex $v_i^h \in V$ with its head telomere vertex $t_i^h \in T$ and each tail vertex $v_i^t \in V$ with its tail telomere vertex $t_i^t \in T$. The edges E_V represent possible genome adjacencies of blocks, and the edges E_U possible telomere adjacencies. The edges E_T are necessary for technical reasons.

Edge weights indicate the number of genomes in which the corresponding adjacency or telomere adjacency is present. All edges E_T are assigned a weight of 0. For assigning weights to the edges E_V and E_U, we start by initializing all edge weights with 0. Next, we iterate over all genomes and increase for each adjacency of two blocks the weight of the corresponding edge $e_V \in E_V$ by 1, and for each telomere adjacency the weight of the corresponding edge $e_T \in E_T$ by $\frac{1}{2}$ [153]. Depending on the orientation of the adjacent blocks, we increase the weight of the edge between the tail and head vertex, tail and tail vertex, head and tail vertex, or head and head vertex; or between the head and head telomere vertex or tail and tail telomere vertex. The total weight of the resulting graph is the total number of block occurrences in the input genomes.

Maximum weight perfect matching. A maximum weight perfect matching on this weighted graph corresponds to a median genome (see Fig. 5.8 and Fig. 5.9). Together with the total weight of the graph, the weight of the matching allows to derive the distance to the median d_M.

The computation of a maximum weight perfect matching is a standard graph theoretical problem: Given a graph $G = (V, E)$, a subset of the edges $E_M \subseteq E$ is called a *matching* if every vertex $v \in V$ has a degree of 0 or 1 in the subgraph $G_M = (V, E_M)$. A *perfect matching* is a matching in which every vertex of the subgraph $G_M = (V, E_M)$ has a degree of 1, thus every vertex is incident to exactly one edge of the matching E_M. Perfect matchings do not exist for every graph, and other graphs have multiple perfect matchings. A *maximum weight perfect matching* of a weighted graph $G = (V, E)$ is a perfect matching $E_M \subseteq E$ that maximizes the sum of the weights of all edges $e \in E_M$.

A perfect matching $E_M \subset E_V \cup E_U \cup E_T$ on the above described graph $G = (V \cup T, E_V \cup E_T \cup E_U)$ corresponds to a genome M formed by all blocks in B. Each vertex $v \in V$ of the graph is incident to exactly one edge of the matching E_M, and hence each block end represented by the vertices is adjacent to exactly one other block end or telomere. Thus, the edges $E_M \cap (E_V \cup E_U)$ represent the adjacencies of a genome M. The edges in E_T guarantee that a perfect matching exists: All vertices in T that are not matched to vertices in V can be matched by an edge from E_T to an arbitrary unmatched vertex in T.

The genome M is a median genome if E_M is a maximum weight perfect matching on G. A max-

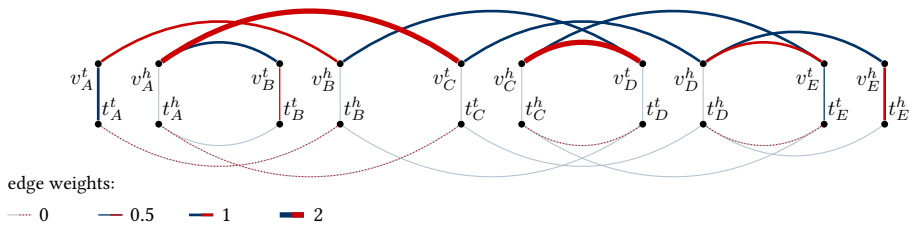

Figure 5.10: The reduced graph for the median instance from Fig. 5.8 and Fig. 5.9. Note that the maximum weight perfect matching (red) uses other edges between telomere vertices than that in Fig. 5.9.

imum weight perfect matching maximizes the weight of the edges in the matching and thereby minimizes the weight of the remaining edges $(E_V \cup E_U \cup E_T) \setminus E_M$. An edge in the weighted graph G has a weight greater than 0 if it corresponds to an adjacency in at least one genome from which G was constructed. Edges that have a weight greater than 0 and are not part of the perfect matching, represent breakpoints between this input genome and M. Thus, if we add up the weights of the edges in $(E_V \cup E_U) \setminus E_M$, we obtain the sum of the breakpoint distances of all input genomes to M. This sum is the distance to the median d_M.

In summary, the total weight of the graph G reduced by the total weight of a maximum weight perfect matching E_M is the median breakpoint distance d_M.

Reducing the number of graph edges. Algorithms for computing a maximum weight perfect matching depend on the number of vertices and number of edges in the graph. Edmonds showed that maximum weight perfect matchings can be computed in polynomial time by inventing the blossom algorithm [52, 53]. The blossom algorithm was followed by many implementations and running time improvements [93]. For example, a recent implementation in the LEMON graph library [46] needs $\mathcal{O}(nm \log n)$ time where n is the number of vertices and m is the number of edges.

The computation of maximum weight perfect matchings is the computationally most expensive task in calculating hidden breakpoint counts. Therefore, running time improvements of the matching computation greatly influence the overall efficiency. The graph described above is highly connected, with the number of edges being quadratic in the number of blocks. Since the number of edges affects the running time, the removal of edges reduces the running time.

The example in Fig. 5.9 illustrates that a large number of the edges in G have a weight of 0. Edges with a weight of 0 do not contribute to the weight of a perfect matching, but nevertheless can be necessary for a perfect matching to exist. However, a large fraction of the edges in G is not necessary for a perfect matching to exist and also does not affect the weight of the maximum weight perfect matching. The following theorem and its proof allow the number of edges to be reduced by a linear factor when computing the distance to a median genome d_M. Figure 5.10 displays the reduced graph for the example from Fig. 5.8.

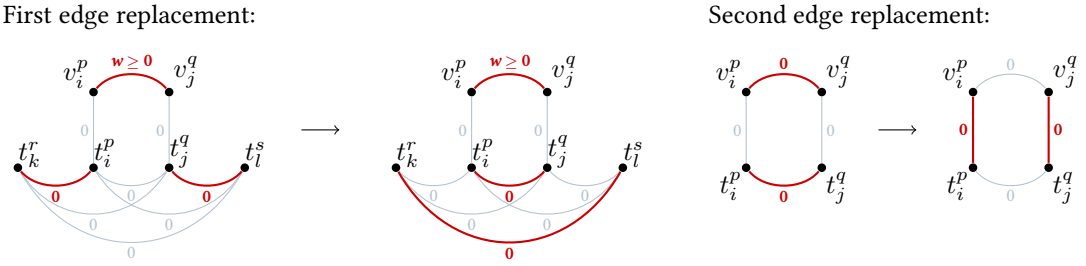

Figure 5.11: Edge replacements in the maximum weight perfect matching E_M (red) for graph reduction according to Theorem 2. The left shows the replacement of two edges from E_T by two other edges from E_T as to create pairs of edges in E_V and E_T. The right shows the replacement of such pairs by edges from E_U. As a result, all edges in $E_M \cap E_V$ have a weight greater than 0.

Theorem 2. *Let w be the weight of a maximum weight perfect matching in the graph $G = (V \cup T, E_V \cup E_T \cup E_U)$ constructed as described above from a set of blocks B and genomes $g_1, \ldots, g_n$. After removing from G*

1. *the edges in E_V that have a weight of 0 and connect vertices v_i^p and v_j^q and*
2. *the corresponding edges in E_T that connect vertices t_i^p and t_j^q*

where $p, q \in \{h, t\}$ and $b_i, b_j \in B$, a perfect matching of weight w still exists in G.

Proof. Given an arbitrary maximum weight perfect matching E_M, this proof will replace subsets of edges in E_M without changing the degree of any vertex in the subgraph $G_M = (V \cup T, E_M)$ and without changing the total weight of E_M. After the replacements, the matching E_M does not use any of the edges listed in Theorem 2, and thus they can be removed. A first edge replacement affects only edges from E_T (see Fig. 5.11, left). A second edge replacement substitutes edges from E_V and E_T by edges from E_U (see Fig. 5.11, right).

The first edge replacement involves the ends $p, q, r, s \in \{h, t\}$ of four blocks $b_i, b_j, b_k, b_l \in B$. Let an edge between the vertices $v_i^p, v_j^q \in V$ be part of the matching E_M. Further, let E_M contain an edge connecting the vertex $t_i^p \in T$ to a vertex $t_k^r \in T$ where $t_k^r \neq t_j^q$. Then, the vertex $t_j^q \in T$ is connected to another vertex $t_l^s \in T$ in E_M. Since all edges between vertices from T have a weight of 0, we can replace the edges between t_i^p and t_k^r and between t_j^q and t_l^s in E_M by the edges between t_i^p and t_j^q and between t_k^r and t_l^s without affecting the weight of E_M. After repeating this replacement for all edges in $E_M \cap E_V$, the matching contains pairs of edges from E_V and E_T.

The second edge replacement addresses edges in $E_M \cap E_V$ that have a weight of 0. After the first edge replacement, the matching E_M contains an edge between the vertices $t_i^p, t_j^q \in T$ for each edge in $E_M \cap E_V$ between the vertices $v_i^p, v_j^q \in V$. If the weight of both edges is 0, then these two edges can be replaced by the two edges between v_i^p and t_i^p and between v_j^q and t_j^q from E_U without affecting the weight of E_M. The two edges in E_U also have a weight of 0, otherwise the matching E_M was not of maximum weight.

After both replacements, the maximum weight perfect matching E_M does not contain any edges from E_V that have a weight of 0 nor the corresponding edges in E_T. Thus, we can safely remove these edges from the graph G without affecting the weight of the resulting maximum weight perfect matching. □

In the reduced graph, the number of edges is linear in the number of blocks multiplied by the number of genomes: The reduced graph contains at most two edges per adjacency of the genomes and one edge per telomere adjacency. Thus, the simplification reduces the running time, which depends linearly on the number of edges, by a linear factor.

Note that the reduction is only possible when computing some median genome or when computing the distance to a median genome. For computing all possible median genomes, the full graph with all edges is necessary. The second edge replacement in the proof substitutes adjacencies of possible median genomes by telomere adjacencies. Thus, the median genomes that we obtain from a reduced graph are preferentially divided into multiple chromosomes instead of using adjacencies not present in any of the input genomes. However, the distance d_M is the same for all median genomes such that the reduction considerably speeds up the calculation of hidden breakpoint counts.

Considerations for duplications. In a median genome as defined by Definition 2 and computed with the above described algorithm, each block from the input set of blocks B occurs exactly once. This limits the median approach for counting hidden breakpoints to genomes with at most one copy per block. To consider duplications in the calculation of hidden breakpoint counts, the median definition and computation needs to be generalized. The following suggests such a generalization.

If more than one input genome has multiple copies of a block, it is likely that an ancestral genome as represented by the median genome had multiple copies. Analogously to including all blocks that occur in at least two of the three genomes, we suggest to include the number of copies that are present in at least two genomes in a median genome. Thus, we assume additional copies of a single genome not to be present in a median genome.

With differing copy numbers of blocks in the compared genomes and the median, we suggest to compute the breakpoint distance based on positional assignments. An input genome has either fewer copies, an equal number of copies, or more copies than the median genome. We suggest to treat additional copies in the input genomes or the median genome as gain/loss and to assume for the other copies a positional assignment that minimizes the breakpoint distance. As we are only interested in the distance, we can leave the actual assignment task open.

When constructing the graph for the median computation, there are two options for representing duplicated blocks: in multiple sets of four vertices (one set for each copy) or in a single set of four vertices (head, tail, head telomere, and tail telomere). If the graph has one set of vertices for each copy in the median, the assignment of weights to the edges leads to ambiguities. If the graph has only one set of vertices per block, then the median genome corresponds to a subgraph in which some vertices are incident to multiple edges. This subgraph is not a perfect matching anymore.

A generalization of the perfect matching problem in graphs is the *f-factor problem.* An f-factor of a graph $G = (V, E)$ is a subset of edges $E_M \subseteq E$ such that each vertex $v \in V$ is incident to exactly $f(v)$ edges in E_M, where $f : V \to \mathbb{N}$ is a function that assigns each vertex a positive integer number. If we represent each block of the genomes as a single set of four vertices in the graph, we can use the function f to specifies the number of copies of the corresponding block in the median genome and replace the computation of a maximum weight perfect matching by the computation of a maximum weight f-factor.

Before solving the f-factor problem, a further extension of the graph is necessary for the edge weighting according to adjacencies. If an adjacency of the same blocks is present multiple times in a genome, we have to add multiple edges between the same two vertices. The weight of the first edge represents the number of genomes that contain the adjacency at least once, the weight of the second edge the number of genomes that contain it at least twice, and so on. The median genome is a maximum weight f-factor in this weighted multigraph.

To solve the f-factor problem, Tutte suggested a graph transformation that turns the f-factor problem into a perfect matching problem [156, 165]. If this transformation generalizes to the maximum weight problem, we can compute the distance to a median genome and, consequentially, a hidden breakpoint count for genomes with duplications.

5.3 Breakpoint analysis of genome alignments

The previous part of the chapter introduced hidden breakpoints as a theoretical concept. To demonstrate that hidden breakpoints are abundant and relevant in the comparison of genomes, this part of the chapter examines the numbers of pairwise and hidden breakpoints in simulated and calculated genome alignments.

The first section of this part describes the setup of the analysis (Section 5.3.1). The next section examines pairwise and hidden breakpoint counts in true alignments of simulated genomes (Section 5.3.2), and the last section evaluates breakpoint counts in calculated alignments of the same simulated genomes and compares them to the true alignments (Section 5.3.3).

5.3.1 Setup of analysis

This section describes the setup of the analysis on simulated data both for the true alignments and the calculated alignments. The setup includes a description of the data sets, details of the evaluation metrics, and a brief summary of the tested alignment programs.

Simulated data sets. Sections 5.3.2 and 5.3.3 examine breakpoint counts in two data sets, *InvNt* and *InvGL*, with 400 and 200 alignment problems, respectively. Each alignment problem consists of nine evolved sequences of length 1 Mbp and their true alignment. The data sets allow for testing the influence of different evolutionary mutation rates on the breakpoint counts.

Table 5.1: Mutation rates for the simulations of the two data sets *InvNt* and *InvGL*.

	inversions	indels	small gain/loss	large gain/loss	nt subst.
InvNt	$0 \,..\, 2 \times 10^{-4}$	1×10^{-3}	5×10^{-4}	2×10^{-5}	$0 \,..\, 1$
InvGL	$0 \,..\, 2 \times 10^{-4}$	1×10^{-3}	5×10^{-4}	$0 \,..\, 1 \times 10^{-4}$	0.01

The large gain/loss rate and the inversion rate are sampled in steps of 1×10^{-5}, the nucleotide substitution rate (nt subst.) in steps of 0.05.

The alignment problems were generated with the program `sgEvolver` [35]. It simulates evolution with a standard Markov process by changing the sequences with five different mutation events at specified rates along a given phylogeny. The mutation events include inversions, nucleotide substitutions, and insertions and deletions of three different size distributions: indels, small gain/loss events, and large gain/loss events.

In the simulation of the data sets *InvNt* and *InvGL*, inversions have geometrically distributed lengths with an expected value of $50,000$, nucleotide substitutions follow an HKY process, indels have Poisson distributed lengths around $\lambda = 3$, small gain/loss events have geometrically distributed lengths with an expected value of 200, and large gain/loss events have uniformly distributed lengths between $10,000$ and $50,000$. The phylogeny that relates the nine sequences in all alignment problems is displayed in Fig. C.1 in the appendix.

Each of the 600 alignment problem uses a different set of mutation rates along the branches of the phylogeny scaled by the respective branch length. The data set *InvNt* samples different inversion rates and different nucleotide substitution rates, and the data set *InvGL* different inversion rates and different large gain/loss rates while keeping the other rates at a fixed value (see Table 5.1). More details of the simulation process with `sgEvolver` can be found in the paper [87].

In the true alignments, we expect only the inversion rate and large gain/loss rate to influence the breakpoint counts but not the nucleotide substitution rate. Hence, *InvNt* is well suited to analyze the effect of the inversion rate, and *InvGL* adds the effect of the large gain/loss rate. In calculated alignments, a high nucleotide substitution rate can lead to missing blocks. Here, both data sets allow for examination of the effect of two evolutionary rates.

Evaluation metrics. Section 5.3.2 reports pairwise and hidden breakpoint counts in the true alignments, and Section 5.3.3 reports pairwise and hidden breakpoint counts in the calculated alignments. In addition, Section 5.3.3 evaluates the difference of breakpoint counts in the calculated alignments with respect to the true alignments and correlates this error with previously described metrics for alignment accuracy.

The alignments of the data sets *InvNt* and *InvGL* consist of nine genomes, but the numbers of breakpoints reported in Sections 5.3.2 and 5.3.3 are pairwise or three-way counts. The reported *pairwise breakpoint counts* are the sum of counts from the projections to all pairs of genomes, and the *hidden breakpoint counts* are the sum of counts from the projections to all triplets of genomes calculated with the median approach. For nine genomes, there are 36 pairs and 84 triplets. The *pairwise breakpoint error* is the difference of pairwise breakpoint counts in calculated

and true alignments, and the *hidden breakpoint error* is the difference of hidden breakpoint counts in calculated and true alignments.

As a breakpoint-independent metric, Section 5.3.3 measures precision and recall of nucleotide alignment accuracy and of indel alignment accuracy [37] in the calculated alignments with respect to the true alignments. The *nucleotide alignment accuracy* compares each pair of aligned nucleotides in the calculated alignments with the aligned pairs in the true alignments. The *indel alignment accuracy* compares the alignment to gaps as described in detail in [37, Section "Accuracy evaluation metrics"]. Both comparisons give numbers of true positives (TP), false positives (FP), and false negatives (FN), which can be combined to the indel or nucleotide precision TP/(TP+FP) and the indel or nucleotide recall TP/(TP+FN). The F_1 score combines precision and recall to one accuracy measure:

$$F_1 = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} .$$

Tested alignment programs. For comparing breakpoint counts in calculated alignments, we tested the genome alignment programs `TBA`, `progressiveMauve`, and `Mugsy`. All three programs compute positional homology alignments, and hence avoid alignment of duplications. Thus, it is possible to analyze breakpoint counts in the alignments calculated by these programs using the median approach without an extension to duplications.

The threaded blockset aligner (`TBA`) [20] initially generates pairwise local alignments with the program `BLASTZ` [148] and then uses a procedure called `MULTIZ` to merge pairwise alignments into multiple alignments (blocks) following a guide tree. In addition, `TBA` splits blocks generated by `MULTIZ` to ensure a partial order of "threaded" blocks. For the breakpoint analysis in this work, alignments were calculated with the guide tree displayed in Fig. C.1 and `TBA` version 2009-Jan-21.

The algorithm implemented in `progressiveMauve` [37] begins by identifying multiple local alignments (approximate multi-matches) among the input genomes, and progressively groups colinear matches into blocks, "locally colinear blocks" (LCBs). It distinguishes positionally homologous regions from random sequence matches with a sum-of-pairs breakpoint scoring scheme as described in Section 5.1.4. At the end of the algorithm, `progressiveMauve` realigns each block with `Muscle` [51] and rejects alignments that are unrelated according to a homology hidden Markov model. The alignments for the breakpoint analysis below were obtained using `progressiveMauve` version 2011-02-02 with default options.

The most recent of the three tested programs is `Mugsy` [9]. `Mugsy` generates local pairwise alignments with `NUCmer` [41] from the `MUMmer 3.0` package [94], which includes a filter for duplications. After removing overlaps of the `NUCmer` matches with a match refinement procedure, `Mugsy` constructs an alignment graph, which is then collapsed into a graph similar to the A-Bruijn graph (see Section 4.1.2). `Mugsy` extracts blocks from this graph using heuristics that eliminate breakpoints including a min-cut max-flow algorithm, and realigns the blocks using `SeqAn::T-Coffee`. The analysis below uses alignments from `Mugsy` version v1r2.2.

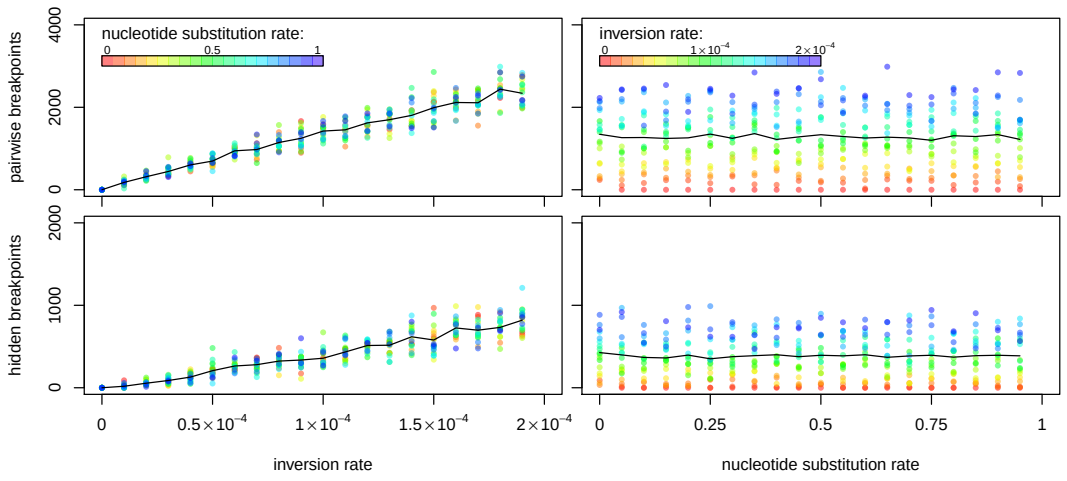

Figure 5.12: Breakpoint counts in true alignments of the data set *InvNt*. Each point corresponds to one alignment in the data set. The two sides display the same data by different mutation rates: by inversion rates (left) or by nucleotide substitution rates (right). Colors indicate the respective other rate: the nucleotide substitution rate (left) or the inversion rate (right). The black line indicates the average breakpoint counts.

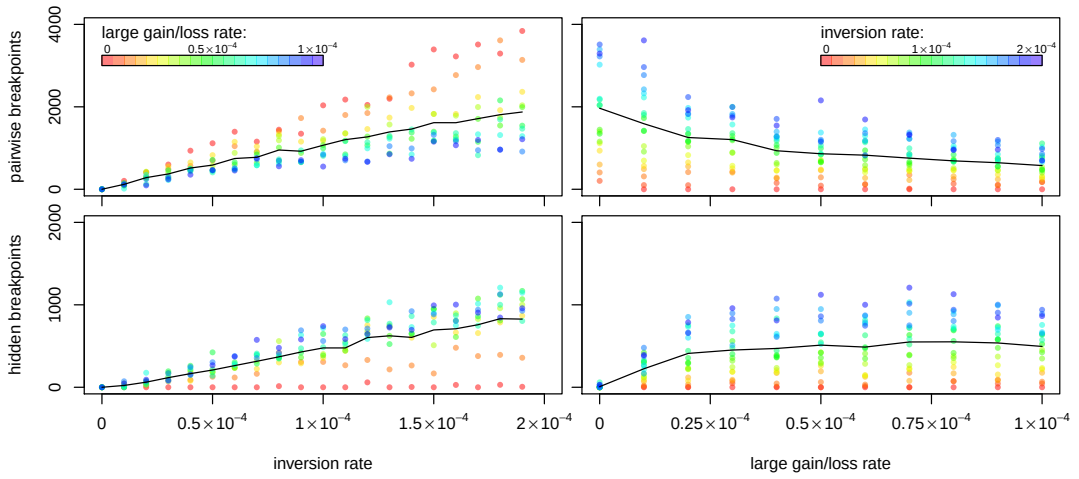

Figure 5.13: Breakpoint counts in true alignments of the data set *InvGL*. Each point corresponds to one alignment in the data set. The two sides display the same data by different mutation rates: by inversion rates (left) or by large gain/loss rates (right). Colors indicate the respective other rate: the large gain/loss rate (left) or the inversion rate (right). The black line indicates the average breakpoint counts.

5.3.2 Breakpoint counts in true alignments

Fig. 5.12 displays the pairwise and hidden breakpoint counts in the true alignments of *InvNt* and Fig. 5.13 displays the counts in the true alignments of *InvGL*. Both figures display the pairwise and hidden breakpoint counts by inversion rate (left), and in addition by nucleotide substitution rate or large gain/loss rate, respectively (right). The pairwise breakpoint counts range from 0 to 3840 and the hidden breakpoint counts from 0 to 1263 across all 600 alignment problems.

Fig. 5.12 demonstrates that both the pairwise breakpoint counts and the hidden breakpoint counts grow nearly linearly with the inversion rate in *InvNt*. There are no breakpoints without inversion events, and the number of breakpoints is largest with the largest inversion rate. Thus, inversion events are necessary for both pairwise and hidden breakpoints to be present in alignments generated with `sgEvolver`. As expected, the nucleotide substitution rate has no influence on the breakpoint counts in the true alignments.

Fig. 5.13 confirms this dependence on the inversion rate with the second data set *InvGL*. Again, the pairwise and hidden breakpoint counts grow nearly linearly with increasing inversion rate. However, in the presence of large gain/loss events (blue on the left in Fig. 5.13), the pairwise breakpoint counts grow slower and the hidden breakpoint counts grow faster than without large gain/loss events (red on the left in Fig. 5.13). Thus, the ratio of hidden to pairwise breakpoints changes with the large gain/loss rate.

The data set *InvGL* illustrates that the large gain/loss rate has a reverse effect on pairwise breakpoint counts than on hidden breakpoint counts (Fig. 5.13, right). The pairwise breakpoint counts drop for growing gain/loss rates. However, the hidden breakpoint counts grow with the gain/loss rate up to about 0.5×10^{-4}. Inversions alone create hardly any hidden breakpoints but they create pairwise breakpoints. Gain/loss events lead to fewer pairwise breakpoints but instead more hidden breakpoints.

In both data sets, the variance in breakpoint counts is due to insertion and deletion events of all three size distributions (indels, small gain/loss, and large gain/loss) at random positions. With a small inversion rate, the probability that a deletion event hides a breakage event from pairwise comparison is smaller. Therefore, the variance is smaller at small inversion rates than at large inversion rates.

In summary, the true alignments confirm that hidden breakpoints are abundant in multiple genome alignments. Furthermore, the effect of the gain/loss rate indicates that hidden breakpoint counts recover some breakage events that are hidden from the pairwise breakpoint distance through loss of sequence.

5.3.3 Breakpoint counts in calculated alignments

Fig. 5.14 and Fig. 5.16 display the breakpoint counts in alignments calculated with `progressiveMauve`, `Mugsy`, and `TBA` for the two data sets by mutation rates. Furthermore, Fig. 5.15 and Fig. 5.17 display the breakpoint errors in relation to nucleotide accuracy and indel accuracy of the same alignments. Figures that show the alignment accuracies by mutation rates can be found

in the appendix (Fig. C.2 and C.3 for *InvNt*, and Fig. C.4 and C.5 for *InvGL*).

Fig. 5.14 reveals that the tested programs mostly overestimate the number of breakpoints in the tested range of inversion rates and nucleotide substitution rates of *InvNt*. For all programs, both the pairwise and hidden breakpoint counts are lowest without inversions and grow roughly linear with an increasing inversion rate similar to the breakpoint counts in true alignments. The two programs `progressiveMauve` and `Mugsy` that implement a breakpoint-reducing algorithm are closer to the breakpoint counts in the true alignments than `TBA`, which does not explicitly account for breakpoints.

While `progressiveMauve` and `TBA` constantly overestimate both pairwise and hidden breakpoint counts, `Mugsy` sometimes underestimates pairwise breakpoint counts. At nucleotide substitution rates above 0.5, the pairwise breakpoint counts in `Mugsy` alignments drop significantly below the pairwise breakpoint counts in true alignments. However, the hidden breakpoint counts do not drop together with the pairwise breakpoint counts. In Fig. 5.14 on the right, the hidden breakpoint counts do not show a significant change at a nucleotide substitution rate of 0.5.

Fig. 5.15 demonstrates a very high overall nucleotide accuracy and high indel accuracy in `progressiveMauve` and `TBA`. However, those `Mugsy` alignments that underestimate the pairwise breakpoint count have a very low nucleotide and indel accuracy. As opposed to the hidden breakpoint counts in Fig. 5.14, the hidden breakpoint error reveals a visible increase for the alignments with lower pairwise breakpoints.

These results suggests that `Mugsy` misses a significant fraction of blocks at high nucleotide substitution rates (see also Fig. C.2 and C.3 in the appendix). Some breakage events remain hidden from pairwise comparison due to low alignment resolution, but the hidden breakpoint counts seem to recover some of them.

The data set *InvGL* and Fig. 5.16 further demonstrate that all three programs constantly overestimate both the pairwise and hidden breakpoint counts across the tested range of inversion rates if the nucleotide substitution rate is fixed at a value as low as 0.01. The breakpoint error is on average even higher in this data set. For example, the average pairwise breakpoint error of `progressiveMauve` at an inversion rate of 0 is approximately 796 (Fig. 5.16, top left), whereas it is 218 in *InvNt* (Fig. 5.14, top left).

Both the pairwise and the hidden breakpoint counts in the calculated alignments are closest to the counts in the true alignments if there are no large gain/loss events (Fig. 5.16, right). For growing large gain/loss rates, the pairwise breakpoint count in the true alignments goes down, but it stays constant or even increases in the calculated alignments. Furthermore, the hidden breakpoint counts in the calculated alignments do not only grow up to a large gain/loss rate of 0.5×10^{-4} as in the true alignments but over the full tested range of large gain/loss rates. Thus, both the pairwise breakpoint error and the hidden breakpoint error increase together with the large gain/loss rate.

Fig. 5.17 indicates that the breakpoint error correlates with nucleotide and indel accuracy of the alignments. Both the pairwise and the hidden breakpoint error is lowest in alignments with high nucleotide and indel accuracy. This suggests that large gain/loss events generally make the alignment of genomes more difficult for the tested programs (see also Fig. C.4 and C.5 in the

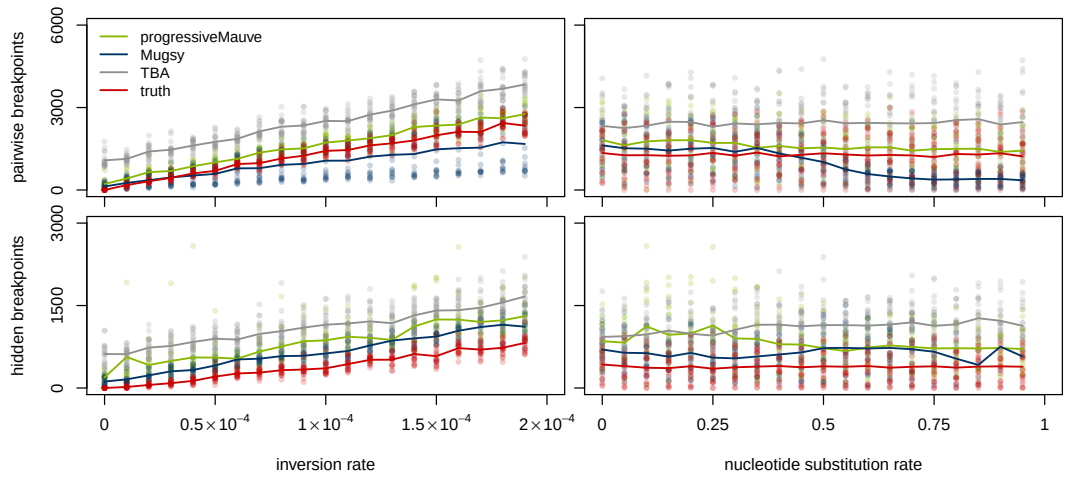

Figure 5.14: Breakpoint counts in calculated alignments of the data set *InvNt*. Each point corresponds to one calculated alignment (gray, dark blue, green) or one true alignment (red). The two sides display the same data by different mutation rates: by inversion rates (left) or by nucleotide substitution rates (right). The lines indicate the average breakpoint count per tested alignment program or true count.

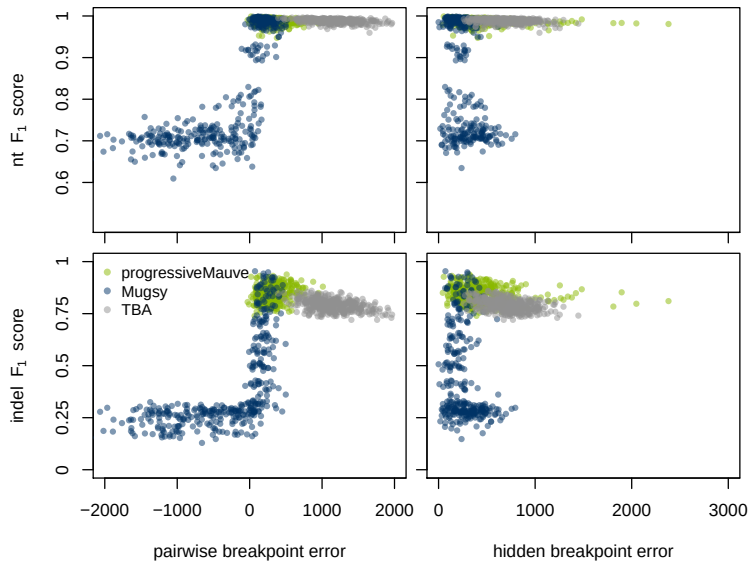

Figure 5.15: Breakpoint error in relation to nucleotide accuracy and indel accuracy in the data set *InvNt*. Each point corresponds to one calculated alignment. The top and bottom display the same breakpoint counts by different alignment accuracy measures, the nucleotide accuracy F_1 score (top) and the indel accuracy F_1 score (bottom).

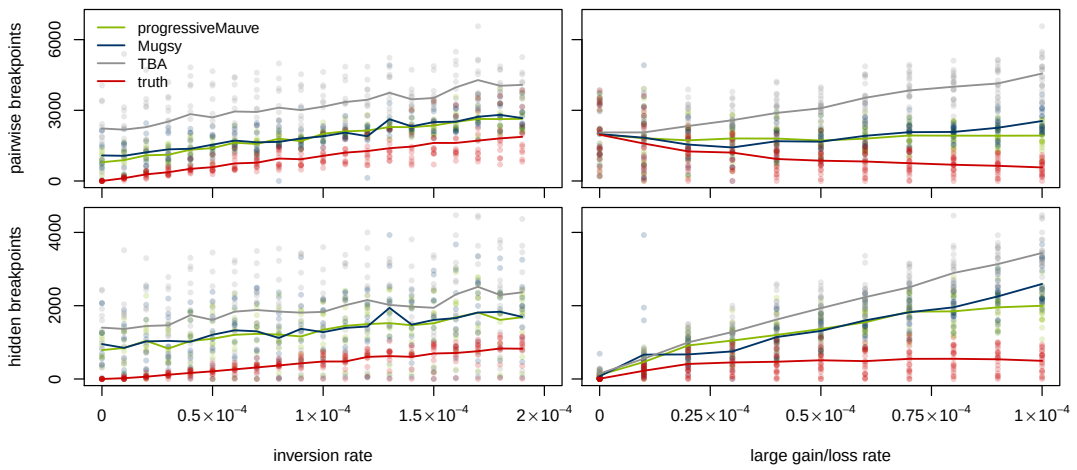

Figure 5.16: Breakpoint counts in calculated alignments of the data set *InvGL*. Each point corresponds to one calculated alignment (gray, dark blue, green) or one true alignment (red). The two sides display the same data by different mutation rates: by inversion rates (left) or by large gain/loss rates (right). The lines indicate the average breakpoint count per tested alignment program or true count.

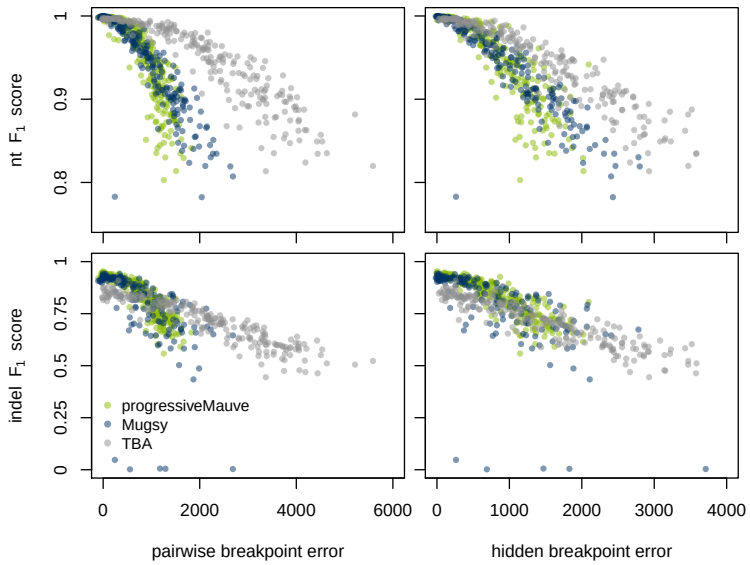

Figure 5.17: Breakpoint error in relation to nucleotide accuracy and indel accuracy in the data set *InvGL*. Each point corresponds to one calculated alignment. The top and bottom display the same breakpoint counts by different alignment accuracy measures, the nucleotide accuracy F_1 score (top) and the indel accuracy F_1 score (bottom).

appendix).

In summary, high nucleotide substitution rates and high large gain/loss rates can make the alignment of genomes more difficult but with different effects on the breakpoint counts. Many nucleotide substitutions can lead to an underestimation of pairwise breakpoints whereas the large gain/loss events lead to overestimation of both hidden and pairwise breakpoints. The inversion rate influences the number of breakpoints, but not the breakpoint error and alignment accuracy in these data sets.

Together with the observations for the breakpoint counts in true alignments, these results confirm that loss of sequence and low alignment resolution decrease pairwise breakpoint counts and create hidden breakpoints. Moreover, all tested programs systematically overestimate both pairwise and hidden breakpoint counts. The two programs `progressiveMauve` and `Mugsy`, which explicitly account for breakpoints, have a higher overall accuracy than `TBA`. Hence, a penalty for hidden breakpoints in a scoring function for local alignment selection appears promising.

5.4 Conclusion to the chapter

This part concludes the chapter with a summary, a discussion, and an outlook. The discussion addresses the potential and limitations of the concept and counting method of hidden breakpoints. The outlook suggests both future research on hidden breakpoints and future research using hidden breakpoints.

Summary. This chapter introduced the concept of hidden breakpoints in alignments of three or more genomes. Three or more genomes provide evidence for genome breakage events that are hidden from pairwise comparison. In pairwise genome alignments, these breakage events create the impression of breakpoint re-use due to sequence loss events or low alignment resolution. Hidden breakpoints partly recover the lost information using a third genome. Conceptually, hidden breakpoints give a lower bound to the number of re-used breakpoints – independent from the controversially discussed breakpoint re-use rate.

Besides describing the concept, the chapter contributes a counting method for hidden breakpoints in three genomes. The counting method projects the genomes to blocks common to at least two of the three genomes. A median of the projected genomes serves as a basis for computing a hidden breakpoint count. This count is the distance to the median reduced by the fraction attributed to pairwise breakpoints.

For the computation of a median genome, the counting method follows recent results of Tannier *et al.* [153]: It constructs a graph and computes a maximum weight perfect matching that represents a median genome. A reduction of the graph described in this chapter allows for efficient computation of the matching, and thus for efficient computation of the median distance and hidden breakpoint count.

The analysis of pairwise and hidden breakpoint counts in the third part of the chapter suggests that hidden breakpoints are abundant if not pervasive in genome alignments. Gain/loss events

in the simulated genome alignments as well as low resolution in calculated alignments reduce pairwise breakpoints and create hidden breakpoints. Furthermore, low alignment accuracy correlates with overestimation of pairwise and hidden breakpoint counts. The results confirm the concept and suggest that hidden breakpoints are suited to measure the degree of non-colinearity in a scoring function of genome alignment methods.

Discussion. Hidden breakpoint counts improve the lower bound to the number of breakage events given by the breakpoint distance. They use a third genome to provide evidence for breakpoint re-use. In contrast to the controversially discussed breakpoint re-use rate [130], the concept and counting method for hidden breakpoints makes no assumptions on the set of evolutionary operations. Hidden breakpoint counts are independent from the biological mechanisms, and thus not specific to a particular rearrangement model.

However, the amount of breakpoint re-use predicted with hidden breakpoint counts is limited to the information provided by the third genome. Breakpoint re-use along a single branch of the phylogeny of the compared genomes cannot be identified with hidden breakpoints. Only if a speciation event is between two usages of a breakpoint, the re-use can be identified as a hidden breakpoint. Thus, the number of breakpoints including both hidden and pairwise breakpoints is still a lower bound to the actual number of breakage events that happened during evolution.

Furthermore, the location of hidden breakpoints in the genomes remains unknown. Hidden breakpoint counts only provide a number, but provide no information about which breakpoints have been re-used. Moreover, the breakage events can not even be assigned to one branch of the phylogeny of the compared genomes. The median approach counts hidden breakpoints only among triplets of genomes.

The median genome calculated in the counting approach does not necessarily represent an ancestral genome. Often, many median genomes with the same distance to the compared genomes exist. The reduction of graph edges described in Section 5.2.3 excludes some possible median genomes from consideration. The counting approach only makes use of the distance to the median, which is equal for all median genomes. Hence, the approach can use any median genome without predicting the actual ancestral genome.

Despite these limitations, the concept of hidden breakpoint counts seems promising for improving the resolution and accuracy of genome alignments. Pairwise breakpoint penalties in the scoring function of genome alignment methods often remove blocks with low positive scores from the alignment. However, these blocks can represent homology. Figure 5.18 shows an example where a small block creates a pairwise breakpoint. Without the small block, there is no pairwise breakpoint but a hidden breakpoint (see also Fig. 5.5). Thus, low alignment resolution replaces the pairwise breakpoint by a hidden breakpoint in this example. With a penalty for hidden breakpoints, a genome alignment method might keep the small block. Hence, a penalty for hidden breakpoints could help distinguishing spurious local alignments from true homologies.

Outlook. The presented counting method and analysis of hidden breakpoint counts provide new opportunities for genome alignment methods and leave room for future research. This in-

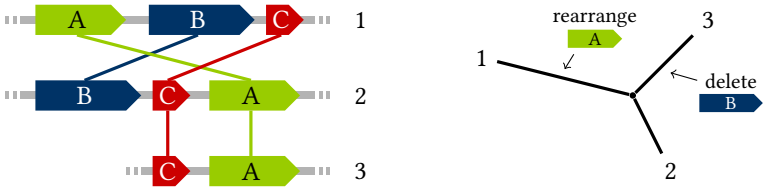

Figure 5.18: The example from Fig 5.5 at higher alignment resolution with an additional small block (red). In contrast to Fig. 5.5, the rearrangement event between genomes 1 and 3 is visible in pairwise comparison. Thus, identification of the red block turns a hidden breakpoint into a pairwise breakpoint.

cludes generalizations of the counting method as well as additional analysis of the abundance of hidden breakpoints among real genomes (see below). Possibly, hidden breakpoint counts can also contribute to research on models of genome evolution. Furthermore, the integration of hidden breakpoint counts into scoring functions of genome alignment methods is an open but promising task.

A first generalization of the counting method will be to allow for arbitrary copy numbers of blocks in the genomes. It requires a generalized graph construction as described at the end of Section 5.2.3 and a graph transformation following Tutte [156] for solving the f-factor problem. The counting method itself as described in Section 5.2.2 remains the same. Testing and evaluating a generalized counting method could include again both simulated alignments and calculated alignments. Tests on calculated alignments can then include genome aligners that allow for duplications, such as `Cactus` [121].

Another generalization of the counting method could address breakpoints hidden from the comparison of three or more genomes. For example, the four genomes AB, BC, CD, and DA hide breakpoints of a rearrangement event from all pairwise and all three-way comparisons. Although the phylogenetic tree of four or more genomes has not only one inner vertex, the distance to a median genome of more than three genomes provides information about hidden breakpoints. Reduced by the number of pairwise and three-way breakpoints, the median distance of four genomes might further improve the lower bound to the number of breakage events.

Further analyses of hidden breakpoint counts should examine alignments of real genomes. These analyses could test the influence of alignment resolution on pairwise and hidden breakpoint counts by calculating genome alignments with different parameter settings. Furthermore, alignments of different sets of species with known differences in the rate of gain/loss can confirm the results from simulated data. Comprehensive testing might potentially reveal significant ratios of hidden to pairwise breakpoints.

For integrating hidden breakpoint counts into the scoring function of genome alignment methods, future research needs to set pairwise and hidden breakpoint counts in relation to each other. The number of three-way comparisons grows faster with the number of genomes than the number of pairwise comparisons does. A scoring function has to account for this difference when applying penalties for pairwise and hidden breakpoints.

In addition, the integration into genome alignment methods requires development of an algorithm that optimizes a scoring function with pairwise and hidden breakpoint penalties. A possible first step will be to investigate how removal or addition of local alignments affect hidden breakpoint counts, and thus the score of genome alignments. A change of the score that depends only on the arrangement of a limited number of blocks would, for example, be favorable for algorithm development.

Chapter 6

Conclusion

This thesis contributed to three aspects of computing and modeling multiple whole-genome alignments. It suggested an efficient approach for the computation of local similarities in whole-genomes, examined graph data structures for representing non-colinear alignments, and extended the breakpoint distance for measuring the degree of non-colinearity in multiple alignments. The following summarizes these contributions, discusses their integration into a genome alignment method including additional steps for future research, and mentions the latest developments and challenges.

Summary of contributions

The first contribution of this thesis is a fully sensitive and efficient local alignment approach. Chapter 3 extended a previously introduced, lossless and efficient filtering algorithm for local alignments by a verification strategy that guarantees full sensitivity. The chapter substantiated this guarantee by a theoretical proof (see Theorem 1 on page 48). The filtering algorithm and verification strategy were implemented in the tool `STELLAR` and evaluated with a parameter study and with a comparison to other local alignment tools. The comparison confirms that the novel approach, despite being fully sensitive, is fast enough to compete with the most widely used seed-and-extend approaches, which often miss a significant amount of local alignments. Thus, `STELLAR` is well-suited for identifying local similarities of whole-genomes.

The next contribution is a comparison and theoretical assessment of graph data structures for genome alignments. Chapter 4 described alignment graphs, A-Bruijn graphs, Enredo graphs, and cactus graphs using consistent terminology for the first time, and revealed that all four graphs rely on vertex or edge labels for representing non-colinearity. Furthermore, the description of transformations proved that the graph structures without labels differ in their information content. Substructures that assist graph-based genome alignment approaches in processing the set

of local alignments appear only partly in the graph structures. Thus, the graph structures have limitations in identifying non-colinearity. They trade different amounts of information about non-colinear changes for compactness of the representation. Nevertheless, the graph data structures conceptually provide a valuable basis for methods that compute genome alignments.

The third contribution is the definition and a counting method for breakpoints that are hidden from pairwise comparisons but become visible in multiple genome alignments. Hidden breakpoints give evidence for breakpoint re-use without making assumptions on evolutionary mechanisms. We consider hidden breakpoint counts as an extension of the pairwise breakpoint distance to multiple genomes, an extension that takes into account gain/loss of blocks. Chapter 5 introduced hidden breakpoints as well as a counting method that handles genomes without duplications. In addition, it proposed a generalization of the counting method to genomes with duplications. An analysis of breakpoint counts in alignments of simulated genomes suggested that hidden breakpoints can improve genome alignment resolution and accuracy if included in the scoring functions of methods for computing genome alignments.

The genome alignment context

The three contributions of this thesis are per se independent from each other, but each addresses challenges that are characteristic of the multiple genome alignment problem: the enormous length of genomes and non-colinearity of homologous regions. More specifically, local alignment approaches have to be both efficient and sensitive for computing genome alignments (Chapter 3); non-colinearity among genome sequences requires new data structures for alignment representation (Chapter 4); and scoring functions for selecting subsets of local alignments need to measure the degree of non-colinearity among multiple genomes (Chapter 5).

In future work, the three contributions can be combined in a new genome alignment method. This method would compute local alignments using `STELLAR`, apply the insights from the graph comparison of Chapter 4 for choosing an appropriate data structure to represent the set of local alignments, and select those local alignments that optimize a scoring function with a penalty for both pairwise and hidden breakpoints. However, additional steps would be necessary to obtain a valid and accurate genome alignment.

These additional steps could include those mentioned in Chapter 1 such as the refinement of partial overlaps, consistency extension, recursion, and realignment, but also further steps that are less well established. The following paragraphs consider the selection of fewer genome pairs for local alignment computation, the integration of other match types with the set of local alignments, and an algorithm for finding the optimal subset of local alignments. These steps seem promising but need further investigation before being applied in methods for computing genome alignment.

Genome pair selection. The computation of local alignments among all genome pairs is very time consuming and prohibitive when comparing many genomes. The number of pairs is quadratic in the number of input genomes. However, the number of pairwise comparisons can

be reduced since the homology relation is transitive (see Fig. 1.8 on page 7). Transitivity allows derivation of similarities between all pairs given alignments of only a linear number of pairs. A similar idea underlies progressive alignment [55]. Fewer pairwise comparisons will allow more genomes in one alignment.

The challenge is to select those pairs for computing local alignments that keep the accuracy of the resulting genome alignment high. The program `FSA` [22] proves that colinear alignments sacrifice hardly any accuracy when reducing the number of pairwise comparisons, and the same seems likely for genome alignments. However, according to the authors of `FSA`, "developing a good theory of which pairs to use to construct the best alignment with the fewest comparisons [...] remains an open problem" [22].

Future research will need to investigate the factors that influence accuracy when selecting pairs. `FSA` selects all pairs of a spanning tree computed from common q-gram counts and determines a number of additional, random pairs with the Erdős-Rényi theory about the connectivity of random graphs [54]. However, using this theory for adding extra pairs seems arbitrary since a spanning tree already ensures connectivity. Maybe methods for identifying uncertain branches in a guide tree computed from common q-gram counts can lead to a better selection of pairs.

Integration of other match types. The integration of information other than sequence similarity in genome alignments could guide the selection of local alignments, and thus improve the prediction of homologies. For example, functional annotations of many sequenced genomes are available in databases and sometimes expression data is present. This information could be used to confirm the identified sequence similarities and clarify ambiguities during the processing of local alignments.

Approaches for this integration would have to bring together information from different levels. Local alignments have nucleotide level resolution, but annotations are given only on the level of sequence segments. A possible solution is to extrapolate alignments of the segments with matching annotations, and then check the agreement of extrapolated matches with local alignments using a certain tolerance. An annotation match that overlaps with a local alignment could raise the score of the local alignment. To handle partial overlaps, a procedure similar to segment match refinement [69, 135] for extrapolated matches could be developed in future research.

Finding the optimal subset of local alignments. Chapter 1 mentioned that many methods for computing genome alignments use greedy approaches to select a subset of local alignments. These greedy approaches risk running into local optima instead of finding the globally optimal genome alignment. A promising alternative to greedy algorithms might be a simulated annealing algorithm for finding the optimal subset of local alignments with a high probability.

Simulated annealing algorithms combine two ideas of recent alignment approaches: iteratively adding and removing local alignments and a probabilistic model. The approach taken by the `Cactus` program [121] iteratively adds and removes local alignments. However, the algorithm in `Cactus` asks the user to define the parameters for a deterministic sequence of iterations. Pachter and coworkers suggested a probabilistic algorithm called sequence annealing [22, 147] for com-

puting colinear alignments. However, this algorithm only adds local alignments with a random component and does not have a removal (melting) step. Thus by combining the two ideas, simulated annealing algorithms overcome limitations and might be able to approximate optimal genome alignments more closely.

Latest developments and challenges

The above examples of additional steps demonstrate that the genome alignment problem leaves room for future research. Apart from developing individual steps of computational methods, an important task is to further establish what a good genome alignment is. This includes formal problem definitions as well as evaluation metrics and benchmarking sets.

Methods for computing genome alignments share the strategy of combining local alignments into a set of blocks, but they disagree in the definition of a genome alignment. In particular, they disagree in how to handle duplications. A method decides whether to align all copies of a duplicated segment in one block or to choose only the positionally conserved copy (see also Fig. 1.7 on page 7). These are two different objectives. Both are correct predictions of homology, however at different resolutions.

The ultimate goal should be a prediction of homology at all resolutions. Genomes can be aligned at different resolutions [91], but a genome alignment represented as a set of blocks can predict homology only at one. Thus, prediction of homology at all resolutions will require new data structures.

Current developments already face the challenge of homology prediction at several resolutions. The authors of the recent genome alignment method `Sibelia` [109] suggest a hierarchical structure of alignments. Furthermore, Paten *et al.* recently introduced a new data structure, a history graph, that unifies alignments with evolutionary scenarios [125]. Future research may need to discuss and develop such structures.

Evaluation of methods for computing genome alignments is being addressed by a current effort, the Alignathon [50]. The initiators invited researchers to compute and submit genome alignments of provided data sets. At the time of this writing the final results have not yet been published, but the submissions confirm that the available methods differ in their objectives. For making a comparison possible, the initiators had to process the submitted alignments with several additional filters.

In addition, the submissions to the Alignathon indicate that not very many software implementations of genome alignment methods are available. Implementations of several methods discussed in this thesis are not available or maintained anymore (for example `ABA` and `DRIMM-Synteny`) and other implementations have many dependencies that make installation hard. Hopefully the availability of genome alignment software will change with further development of the field.

As sequencing technologies keep improving, the availability of genome alignment tools will become relevant for more and more studies. Currently, the comparison of genomes from multiple individuals of the same species is gaining importance. Furthermore, the comparison of cancer

genomes, where massive rearrangement is observed, might be a future application for genome alignments.

Despite its enormous potential, the multiple genome alignment problem remains a challenging task in terms of both computing and modeling. Future research needs to establish a common understanding of genome alignment and related problems, and certainly will further improve available methods. The three contributions of this thesis are a promising basis for a novel multiple whole-genome alignment program.

Bibliography

[1] 1000 Genomes Project Consortium. A map of human genome variation from population-scale sequencing. *Nature*, 467(7319):1061–1073, 2010.

[2] M. D. Adams, S. E. Celniker, R. A. Holt, C. A. Evans, J. D. Gocayne, P. G. Amanatides, S. E. Scherer, P. W. Li, R. A. Hoskins, R. F. Galle, R. A. George, S. E. Lewis, S. Richards, M. Ashburner, S. N. Henderson, G. G. Sutton, J. R. Wortman, M. D. Yandell, Q. Zhang, L. X. Chen, R. C. Brandon, Y. H. Rogers, R. G. Blazej, M. Champe, B. D. Pfeiffer, K. H. Wan, C. Doyle, E. G. Baxter, G. Helt, C. R. Nelson, G. L. Gabor, J. F. Abril, A. Agbayani, H. J. An, C. Andrews-Pfannkoch, D. Baldwin, R. M. Ballew, A. Basu, J. Baxendale, L. Bayraktaroglu, E. M. Beasley, K. Y. Beeson, P. V. Benos, B. P. Berman, D. Bhandari, S. Bolshakov, D. Borkova, M. R. Botchan, J. Bouck, P. Brokstein, et al. The genome sequence of drosophila melanogaster. *Science*, 287(5461):2185–2195, 2000.

[3] M. A. Alekseyev and P. A. Pevzner. Breakpoint graphs and ancestral genome reconstructions. *Genome Res*, 19(5):943–957, 2009.

[4] M. A. Alekseyev and P. A. Pevzner. Comparative genomics reveals birth and death of fragile regions in mammalian evolution. *Genome Biol*, 11(11):R117, 2010.

[5] G. Allan, S. Krakowka, J. Ellis, and C. Charreyre. Discovery and evolving history of two genetically related but phenotypically different viruses, porcine circoviruses 1 and 2. *Virus Res*, 164(1-2):4–9, 2012.

[6] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *J Mol Biol*, 215(3):403–410, 1990.

[7] S. F. Altschul, T. L. Madden, A. A. Schäffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Res*, 25(17):3389–3402, 1997.

[8] S. V. Angiuoli, J. C. Dunning Hotopp, S. L. Salzberg, and H. Tettelin. Improving pan-genome annotation using whole genome multiple alignment. *BMC Bioinformatics*, 12:272, 2011.

[9] S. V. Angiuoli and S. L. Salzberg. Mugsy: fast multiple alignment of closely related whole genomes. *Bioinformatics*, 27(3):334–342, 2011.

[10] A. N. Arslan and Ö. Eğecioğlu. Efficient algorithms for normalized edit distance. *Journal of Discrete Algorithms*, 1:3–20, 2000.

[11] O. Attie, A. E. Darling, and S. Yancopoulos. The rise and fall of breakpoint reuse depending on genome resolution. *BMC Bioinformatics*, 12 Suppl 9:S1, 2011.

[12] V. Bafna and P. A. Pevzner. Genome rearrangements and sorting by reversals. In *Foundations of Computer Science, 1993. Proceedings., 34th Annual Symposium on*, pages 148–157, 1993.

[13] T. L. Bailey, M. Boden, F. A. Buske, M. Frith, C. E. Grant, L. Clementi, J. Ren, W. W. Li, and W. S. Noble. MEME SUITE: tools for motif discovery and searching. *Nucleic Acids Res*, 37(Web Server issue):W202–W208, 2009.

[14] N. A. Belal and L. S. Heath. A theoretical model for whole genome alignment. *J Comput Biol*, 18(5):705–728, 2011.

[15] M. D. Bennett and I. J. Leitch. Nuclear DNA amounts in angiosperms: targets, trends and tomorrow. *Ann Bot*, 107(3):467–590, 2011.

[16] A. Bergeron, C. Chauve, and Y. Gingras. Formal models of gene clusters. In I. I. Măndoiu and A. Zelikovsky, editors, *Bioinformatics Algorithms*, chapter 8, pages 175–202. John Wiley & Sons, Inc., 2007.

[17] A. Bergeron, J. Mixtacki, and J. Stoye. A unifying view of genome rearrangements. In P. Bücher and B. M. Moret, editors, *Algorithms in Bioinformatics*, volume 4175 of *Lecture Notes in Computer Science*, pages 163–173. Springer Berlin Heidelberg, 2006.

[18] M. Bernt, C. Bleidorn, A. Braband, J. Dambach, A. Donath, G. Fritzsch, A. Golombek, H. Hadrys, F. Jühling, K. Meusemann, M. Middendorf, B. Misof, M. Perseke, L. Podsiadlowski, B. von Reumont, B. Schierwater, M. Schlegel, M. Schrödl, S. Simon, P. F. Stadler, I. Stöger, and T. H. Struck. A comprehensive analysis of bilaterian mitochondrial genomes and phylogeny. *Mol Phylogenet Evol*, 69(2):352–364, 2013.

[19] M. R. Berthold, N. Cebron, F. Dill, T. R. Gabriel, T. Kötter, T. Meinl, P. Ohl, C. Sieb, K. Thiel, and B. Wiswedel. KNIME: the Konstanz information miner. In *Studies in Classification, Data Analysis, and Knowledge Organization (GfKL 2007)*. Springer, 2007.

[20] M. Blanchette, W. J. Kent, C. Riemer, L. Elnitski, A. F. A. Smit, K. M. Roskin, R. Baertsch, K. Rosenbloom, H. Clawson, E. D. Green, D. Haussler, and W. Miller. Aligning multiple genomic sequences with the threaded blockset aligner. *Genome Res*, 14(4):708–715, 2004.

[21] G. Bourque and P. A. Pevzner. Genome-scale evolution: reconstructing gene orders in the ancestral species. *Genome Res*, 12(1):26–36, 2002.

[22] R. K. Bradley, A. Roberts, M. Smoot, S. Juvekar, J. Do, C. Dewey, I. Holmes, and L. Pachter. Fast statistical alignment. *PLoS Comput Biol*, 5(5):e1000392, 2009.

[23] N. Bray and L. Pachter. MAVID: constrained ancestral alignment of multiple sequences. *Genome Res*, 14(4):693–699, 2004.

[24] T. A. Brown. *Genomes*. Wiley-Liss, Oxford, 2nd edition, 2002.

[25] M. Brudno, M. Chapman, B. Göttgens, S. Batzoglou, and B. Morgenstern. Fast and sensitive multiple alignment of large genomic sequences. *BMC Bioinformatics*, 4:66, 2003.

[26] M. Brudno, S. Malde, A. Poliakov, C. B. Do, O. Couronne, I. Dubchak, and S. Batzoglou. Glocal alignment: finding rearrangements during alignment. *Bioinformatics*, 19 Suppl 1:i54–i62, 2003.

[27] S. Burkhardt, A. Crauser, P. Ferragina, H.-P. Lenhof, E. Rivals, and M. Vingron. q-gram based database searching using a suffix array (QUASAR). In *Proceedings of the third annual international conference on Computational molecular biology*, RECOMB '99, pages 77–83, New York, NY, USA, 1999. ACM.

[28] C. Camacho, G. Coulouris, V. Avagyan, N. Ma, J. Papadopoulos, K. Bealer, and T. L. Madden. BLAST+: architecture and applications. *BMC Bioinformatics*, 10:421, 2009.

[29] R. Carlson. The pace and proliferation of biological technologies. *Biosecur Bioterror*, 1(3):203–214, 2003.

[30] R. Carlson. Planning for toy story and synthetic biology: It's all about competition. `http://www.synthesis.cc/2013/04/`, Oct 2013.

[31] F. Chiaromonte, V. B. Yap, and W. Miller. Scoring pairwise genomic sequence alignments. *Pac Symp Biocomput*, pages 115–126, 2002.

[32] P. E. C. Compeau, P. A. Pevzner, and G. Tesler. How to apply de Bruijn graphs to genome assembly. *Nat Biotechnol*, 29(11):987–991, 2011.

[33] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 2009.

[34] M. Csűrös. Performing local similarity searches with variable length seeds. In S. Sahinalp, S. Muthukrishnan, and U. Dogrusoz, editors, *Combinatorial Pattern Matching*, volume 3109 of *Lecture Notes in Computer Science*, pages 373–387. Springer Berlin Heidelberg, 2004.

[35] A. C. E. Darling, B. Mau, F. R. Blattner, and N. T. Perna. Mauve: multiple alignment of conserved genomic sequence with rearrangements. *Genome Res*, 14(7):1394–1403, 2004.

[36] A. E. Darling. *Computational analysis of genome evolution*. PhD thesis, University of Wisconsin – Madison, 2006.

[37] A. E. Darling, B. Mau, and N. T. Perna. progressiveMauve: multiple genome alignment with gene gain, loss and rearrangement. *PLoS One*, 5(6):e11147, 2010.

[38] M. O. Dayhoff, R. M. Schwartz, and B. C. Orcutt. A model of evolutionary change in proteins. *Atlas of protein sequence and structure*, 5(suppl 3):345–352, 1978.

[39] N. G. de Bruijn. A combinatorial problem. *Proc Nederl Akad Wetensch*, 49:758–764, 1946.

[40] A. L. Delcher, S. Kasif, R. D. Fleischmann, J. Peterson, O. White, and S. L. Salzberg. Alignment of whole genomes. *Nucleic Acids Res*, 27(11):2369–2376, 1999.

[41] A. L. Delcher, A. Phillippy, J. Carlton, and S. L. Salzberg. Fast algorithms for large-scale genome alignment and comparison. *Nucleic Acids Res*, 30(11):2478–2483, 2002.

[42] F. Delsuc, H. Brinkmann, and H. Philippe. Phylogenomics and the reconstruction of the tree of life. *Nat Rev Genet*, 6(5):361–375, 2005.

[43] C. N. Dewey. Aligning multiple whole genomes with Mercator and MAVID. *Methods Mol Biol*, 395:221–236, 2007.

[44] C. N. Dewey. Positional orthology: putting genomic evolutionary relationships into context. *Brief Bioinform*, 12(5):401–412, 2011.

[45] C. N. Dewey and L. Pachter. Evolution at the nucleotide level: the problem of multiple whole-genome alignment. *Hum Mol Genet*, 15 Spec No 1:R51–R56, 2006.

[46] B. Dezs, A. Jüttner, and P. Kovács. LEMON - an open source C++ graph template library. *Electron Notes Theor Comput Sci*, 264(5):23–45, 2011.

[47] C. B. Do, M. S. P. Mahabhashyam, M. Brudno, and S. Batzoglou. ProbCons: probabilistic consistency-based multiple sequence alignment. *Genome Res*, 15(2):330–340, 2005.

[48] A. Döring, D. Weese, T. Rausch, and K. Reinert. SeqAn an efficient, generic C++ library for sequence analysis. *BMC Bioinformatics*, 9:11, 2008.

[49] I. Dubchak, A. Poliakov, A. Kislyuk, and M. Brudno. Multiple whole-genome alignments without a reference organism. *Genome Res*, 19(4):682–689, 2009.

[50] D. Earl. Alignathon: preliminary results of a whole genome alignment assessment. `http://compbio.soe.ucsc.edu/alignathon/`, 2013.

[51] R. C. Edgar. MUSCLE: multiple sequence alignment with high accuracy and high throughput. *Nucleic Acids Res*, 32(5):1792–1797, 2004.

[52] J. Edmonds. Maximum matching and a polyhedron with 0,1-vertices. *Journal of Research of the National Bureau of Standards*, 69B:125–130, 1965.

[53] J. Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17:449–467, 1965.

[54] P. Erdős and A. Rényi. On the evolution of random graphs. In *Publications of the Mathematical Institute of the Hungarian Academy of Sciences*, pages 17–61, 1960.

[55] D. F. Feng and R. F. Doolittle. Progressive sequence alignment as a prerequisite to correct phylogenetic trees. *J Mol Evol*, 25(4):351–360, 1987.

[56] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, FOCS '00, pages 390–398, 2000.

[57] G. Fertin, A. Labarre, I. Rusu, E. Tannier, and S. Vialette. *Combinatorics of Genome Rearrangement*. The MIT Press, 2009.

[58] J. W. Fickett. Fast optimal alignment. *Nucleic Acids Res*, 12(1 Pt 1):175–179, 1984.

[59] R. D. Fleischmann, M. D. Adams, O. White, R. A. Clayton, E. F. Kirkness, A. R. Kerlavage, C. J. Bult, J. F. Tomb, B. A. Dougherty, and J. M. Merrick. Whole-genome random sequencing and assembly of Haemophilus influenzae Rd. *Science*, 269(5223):496–512, 1995.

[60] R. W. Floyd. Nondeterministic algorithms. *J ACM*, 14(4):636–644, 1967.

[61] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

[62] J. Fostier, S. Proost, B. Dhoedt, Y. Saeys, P. Demeester, Y. V. de Peer, and K. Vandepoele. A greedy, graph-based algorithm for the alignment of multiple homologous gene lists. *Bioinformatics*, 27(6):749–756, 2011.

[63] M. C. Frith, M. Hamada, and P. Horton. Parameters for accurate genome alignment. *BMC Bioinformatics*, 11:80, 2010.

[64] A. Gogol-Döring and K. Reinert. *Biological Sequence Analysis Using the SeqAn C++ Library*. Chapman & Hall/CRC Mathematical & Computational Biology. CRC Press, Boca Raton, USA, Nov 2009.

[65] R. E. Green, J. Krause, A. W. Briggs, T. Maricic, U. Stenzel, M. Kircher, N. Patterson, H. Li, W. Zhai, M. H.-Y. Fritz, N. F. Hansen, E. Y. Durand, A.-S. Malaspinas, J. D. Jensen, T. Marques-Bonet, C. Alkan, K. Prüfer, M. Meyer, H. A. Burbano, J. M. Good, R. Schultz, A. Aximu-Petri, A. Butthof, B. Höber, B. Höffner, M. Siegemund, A. Weihmann, C. Nusbaum, E. S. Lander, C. Russ, N. Novod, J. Affourtit, M. Egholm, C. Verna, P. Rudan, D. Brajkovic, Z. Kucan, I. Gusic, V. B. Doronichev, L. V. Golovanova, C. Lalueza-Fox, M. de la Rasilla, J. Fortea, A. Rosas, R. W. Schmitz, P. L. F. Johnson, E. E. Eichler, D. Falush, E. Birney, J. C. Mullikin, et al. A draft sequence of the neandertal genome. *Science*, 328(5979):710–722, May 2010.

[66] T. Gregory. Animal genome size database. `http://www.genomesize.com`, 2005.

[67] T. R. Gregory, J. A. Nicol, H. Tamm, B. Kullman, K. Kullman, I. J. Leitch, B. G. Murray, D. F. Kapraun, J. Greilhuber, and M. D. Bennett. Eukaryotic genome size databases. *Nucleic Acids Res*, 35(Database issue):D332–D338, 2007.

[68] P. Hallin and S. Borini. Genome atlas database. `http://www.cbs.dtu.dk/services/GenomeAtlas/`, Oct 2012.

[69] A. Halpern, D. Huson, and K. Reinert. Segment match refinement and applications. In R. Guigó and D. Gusfield, editors, *Algorithms in Bioinformatics*, volume 2452 of *Lecture Notes in Computer Science*, pages 126–139. Springer Berlin Heidelberg, 2002.

[70] R. W. Hamming. Error detecting and error correcting codes. *Bell System Technical Journal*, 29(2):147–160, 1950.

[71] S. Hannenhalli. Polynomial-time algorithm for computing translocation distance between genomes. *Discrete Applied Mathematics*, 71(1–3):137–151, 1996.

[72] S. Hannenhalli and P. Pevzner. Towards a computational theory of genome rearrangements. In J. Leeuwen, editor, *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*, pages 184–202. Springer Berlin Heidelberg, 1995.

[73] S. Hannenhalli and P. Pevzner. Transforming men into mice (polynomial algorithm for genomic distance problem). In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, pages 581–592, 1995.

[74] S. Hannenhalli and P. A. Pevzner. Transforming cabbage into turnip: polynomial algorithm for sorting signed permutations by reversals. *J ACM*, 46(1):1–27, 1999.

[75] F. Harary and G. E. Uhlenbeck. On the number of husimi trees: I. *Proc Natl Acad Sci U S A*, 39(4):315–322, 1953.

[76] R. Harris. *Improved pairwise alignment of genomic DNA*. PhD thesis, The Pennsylvania State University, 2007.

[77] S. Henikoff and J. G. Henikoff. Amino acid substitution matrices from protein blocks. *Proc Natl Acad Sci U S A*, 89(22):10915–10919, 1992.

[78] M. Höhl, S. Kurtz, and E. Ohlebusch. Efficient multiple genome alignment. *Bioinformatics*, 18 Suppl 1:S312–S320, 2002.

[79] Y. Hou and S. Lin. Distinct gene number-genome size relationships for eukaryotes and non-eukaryotes: gene content estimation for dinoflagellate genomes. *PLoS One*, 4(9):e6978, 2009.

[80] P. Jokinen and E. Ukkonen. Two algorithms for approxmate string matching in static texts. In A. Tarlecki, editor, *Mathematical Foundations of Computer Science 1991*, volume 520 of *Lecture Notes in Computer Science*, pages 240–248. Springer Berlin Heidelberg, 1991.

[81] S. Karlin and S. F. Altschul. Methods for assessing the statistical significance of molecular sequence features by using general scoring schemes. *Proc Natl Acad Sci U S A*, 87(6):2264–2268, 1990.

[82] K. Katoh, K. Misawa, K. Kuma, and T. Miyata. MAFFT: a novel method for rapid multiple sequence alignment based on fast fourier transform. *Nucleic Acids Res*, 30(14):3059–3066, 2002.

[83] J. Kececioglu. The maximum weight trace problem in multiple sequence alignment. In *Proceedings of the 4th Symposium on Combinatorial Pattern Matching (CPM)*, volume 684 of *Lecture Notes in Computer Science*, pages 106–119. Springer-Verlag, 1993.

[84] J. Kececioglu and D. Sankoff. Exact and approximation algorithms for sorting by reversals, with application to genome rearrangement. *Algorithmica*, 13(1-2):180–210, 1995.

[85] J. D. Kececioglu and R. Ravi. Of mice and men: algorithms for evolutionary distances between genomes with translocation. In *Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms*, SODA '95, pages 604–613, Philadelphia, PA, USA, 1995. Society for Industrial and Applied Mathematics.

[86] J. D. Kececioglu and D. Sankoff. Efficient bounds for oriented chromosome inversion distance. In *Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching*, CPM '94, pages 307–325. Springer-Verlag, 1994.

[87] B. Kehr, K. Reinert, and A. E. Darling. Hidden breakpoints in genome alignments. In B. Raphael and J. Tang, editors, *Algorithms in Bioinformatics*, volume 7534 of *Lecture Notes in Computer Science*, pages 391–403. Springer Berlin Heidelberg, 2012.

[88] B. Kehr, K. Trappe, M. Holtgrewe, and K. Reinert. Genome alignment with graph data structures: a comparison. *Submitted*, 2013.

[89] B. Kehr, D. Weese, and K. Reinert. STELLAR: fast and exact local alignments. *BMC Bioinformatics*, 12 Suppl 9:S15, 2011.

[90] W. J. Kent. BLAT–the BLAST-like alignment tool. *Genome Res*, 12(4):656–664, 2002.

[91] W. J. Kent, R. Baertsch, A. Hinrichs, W. Miller, and D. Haussler. Evolution's cauldron: duplication, deletion, and rearrangement in the mouse and human genomes. *Proc Natl Acad Sci U S A*, 100(20):11484–11489, 2003.

[92] S. M. Kielbasa, R. Wan, K. Sato, P. Horton, and M. C. Frith. Adaptive seeds tame genomic sequence comparison. *Genome Res*, 21(3):487–493, 2011.

[93] V. Kolmogorov. Blossom V: a new implementation of a minimum cost perfect matching algorithm. *Mathematical Programming Computation*, 1(1):43–67, 2009.

[94] S. Kurtz, A. Phillippy, A. L. Delcher, M. Smoot, M. Shumway, C. Antonescu, and S. L. Salzberg. Versatile and open software for comparing large genomes. *Genome Biol*, 5(2):R12, 2004.

[95] T. W. Lam, W. K. Sung, S. L. Tam, C. K. Wong, and S. M. Yiu. Compressed indexing and local alignment of DNA. *Bioinformatics*, 24(6):791–797, 2008.

[96] E. S. Lander, L. M. Linton, B. Birren, C. Nusbaum, M. C. Zody, J. Baldwin, K. Devon, K. Dewar, M. Doyle, W. FitzHugh, R. Funke, D. Gage, K. Harris, A. Heaford, J. Howland, L. Kann, J. Lehoczky, R. LeVine, P. McEwan, K. McKernan, J. Meldrim, J. P. Mesirov, C. Miranda, W. Morris, J. Naylor, C. Raymond, M. Rosetti, R. Santos, A. Sheridan, C. Sougnez, N. Stange-Thomann, N. Stojanovic, A. Subramanian, D. Wyman, J. Rogers, J. Sulston, R. Ainscough, S. Beck, D. Bentley, J. Burton, C. Clee, N. Carter, A. Coulson, R. Deadman, P. Deloukas, A. Dunham, I. Dunham, R. Durbin, L. French, D. Grafham, et al. Initial sequencing and analysis of the human genome. *Nature*, 409(6822):860–921, 2001.

[97] C. Lee, C. Grasso, and M. F. Sharlow. Multiple sequence alignment using partial order graphs. *Bioinformatics*, 18(3):452–464, 2002.

[98] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.

[99] M. Li, B. Ma, D. Kisman, and J. Tromp. PatternHunter II: highly sensitive and fast homology search. *J Bioinform Comput Biol*, 2(3):417–439, 2004.

[100] A. Löytynoja and N. Goldman. An algorithm for progressive multiple alignment of sequences with insertions. *Proc Natl Acad Sci U S A*, 102(30):10557–10562, 2005.

[101] H. Luo, W. Arndt, Y. Zhang, G. Shi, M. A. Alekseyev, J. Tang, A. L. Hughes, and R. Friedman. Phylogenetic analysis of genome rearrangements among five mammalian orders. *Mol Phylogenet Evol*, 65(3):871–882, 2012.

[102] B. Ma, J. Tromp, and M. Li. PatternHunter: faster and more sensitive homology search. *Bioinformatics*, 18(3):440–445, 2002.

[103] J. Ma, L. Zhang, B. B. Suh, B. J. Raney, R. C. Burhans, W. J. Kent, M. Blanchette, D. Haussler, and W. Miller. Reconstructing contiguous regions of an ancestral genome. *Genome Res*, 16(12):1557–1565, 2006.

[104] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[105] A. Marzal and E. Vidal. Computation of normalized edit distance and applications. *IEEE T Pattern Anal*, 15:926–932, 1993.

[106] C. Médigue and I. Moszer. Annotation, comparison and databases for hundreds of bacterial genomes. *Res Microbiol*, 158(10):724–736, 2007.

[107] C. Meinel and M. Mundhenk. *Mathematische Grundlagen der Informatik: Mathematisches Denken und Beweisen*. Vieweg+Teubner Verlag, 3rd edition, 2006.

[108] W. Miller, D. I. Drautz, A. Ratan, B. Pusey, J. Qi, A. M. Lesk, L. P. Tomsho, M. D. Packard, F. Zhao, A. Sher, A. Tikhonov, B. Raney, N. Patterson, K. Lindblad-Toh, E. S. Lander, J. R. Knight, G. P. Irzyk, K. M. Fredrikson, T. T. Harkins, S. Sheridan, T. Pringle, and S. C. Schuster. Sequencing the nuclear genome of the extinct woolly mammoth. *Nature*, 456(7220):387–390, 2008.

[109] I. Minkin, A. Patel, M. Kolmogorov, N. Vyahhi, and S. Pham. Sibelia: a scalable and comprehensive synteny block generation tool for closely related microbial genomes. In A. Darling and J. Stoye, editors, *Algorithms in Bioinformatics*, volume 8126 of *Lecture Notes in Computer Science*, pages 215–229. Springer Berlin Heidelberg, 2013.

[110] C. Mora, D. P. Tittensor, S. Adl, A. G. B. Simpson, and B. Worm. How many species are there on Earth and in the ocean? *PLoS Biol*, 9(8):e1001127, 2011.

[111] B. Morgenstern, A. Dress, and T. Werner. Multiple DNA and protein sequence alignment based on segment-to-segment comparison. *Proc Natl Acad Sci U S A*, 93(22):12098–12103, 1996.

[112] B. Morgenstern, K. Frech, A. Dress, and T. Werner. DIALIGN: finding local similarities by multiple sequence alignment. *Bioinformatics*, 14(3):290–294, 1998.

[113] E. Myers. A sublinear algorithm for approximate keyword searching. *Algorithmica*, 12:345–374, 1994.

[114] J. H. Nadeau and B. A. Taylor. Lengths of chromosomal segments conserved since divergence of man and mouse. *Proc Natl Acad Sci U S A*, 81(3):814–818, 1984.

[115] National Center for Biotechnology Information (NCBI). Genome sequencing projects statistics. `http://www.ncbi.nlm.nih.gov/genomes/static/gpstat.html`, Oct 2013.

[116] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *J Mol Biol*, 48(3):443–453, 1970.

[117] L. Noé and G. Kucherov. Improved hit criteria for DNA local alignment. *BMC Bioinformatics*, 5:149, 2004.

[118] C. Notredame, D. G. Higgins, and J. Heringa. T-Coffee: a novel method for fast and accurate multiple sequence alignment. *J Mol Biol*, 302(1):205–217, 2000.

[119] C. Notredame, L. Holm, and D. G. Higgins. Coffee: an objective function for multiple sequence alignments. *Bioinformatics*, 14(5):407–422, 1998.

[120] I. Pacific Biosciences of California. Pacific biosciences introduces new chemistry with longer read lengths to detect novel features in DNA sequence and advance genome studies of large organisms. `http://investor.pacificbiosciences.com/releasedetail.cfm?ReleaseID=794692`, Oct 2013.

[121] B. Paten, M. Diekhans, D. Earl, J. S. John, J. Ma, B. Suh, and D. Haussler. Cactus graphs for genome comparisons. *J Comput Biol*, 18(3):469–481, 2011.

[122] B. Paten, D. Earl, N. Nguyen, M. Diekhans, D. Zerbino, and D. Haussler. Cactus: algorithms for genome multiple sequence alignment. *Genome Res*, 21(9):1512–1528, 2011.

[123] B. Paten, J. Herrero, K. Beal, and E. Birney. Sequence progressive alignment, a framework for practical large-scale probabilistic consistency alignment. *Bioinformatics*, 25(3):295–301, 2009.

[124] B. Paten, J. Herrero, K. Beal, S. Fitzgerald, and E. Birney. Enredo and Pecan: genome-wide mammalian consistency-based multiple alignment with paralogs. *Genome Res*, 18(11):1814–1828, 2008.

[125] B. Paten, D. R. Zerbino, G. Hickey, and D. Haussler. A unifying parsimony model of genome evolution. *ArXiv e-prints*, 2013.

[126] W. R. Pearson. Searching protein sequence libraries: comparison of the sensitivity and selectivity of the Smith-Waterman and FASTA algorithms. *Genomics*, 11(3):635–650, 1991.

[127] W. R. Pearson. The FASTA package of sequence comparison programs. `http://fasta.bioch.virginia.edu/fasta_www2/fasta_down.shtml`, Mar 2013.

[128] W. R. Pearson and D. J. Lipman. Improved tools for biological sequence comparison. *Proc Natl Acad Sci U S A*, 85(8):2444–2448, 1988.

[129] Q. Peng, P. A. Pevzner, and G. Tesler. The fragile breakage versus random breakage models of chromosome evolution. *PLoS Comput Biol*, 2(2):e14, 2006.

[130] P. Pevzner and G. Tesler. Human and mouse genomic sequences reveal extensive breakpoint reuse in mammalian evolution. *Proc Natl Acad Sci U S A*, 100(13):7672–7677, 2003.

[131] P. A. Pevzner, H. Tang, and G. Tesler. De novo repeat classification and fragment assembly. *Genome Res*, 14(9):1786–1796, 2004.

[132] S. K. Pham and P. A. Pevzner. DRIMM-Synteny: decomposing genomes into evolutionary conserved segments. *Bioinformatics*, 26(20):2509–2516, 2010.

[133] B. Raphael, D. Zhi, H. Tang, and P. Pevzner. A novel method for multiple alignment of sequences with repeated and shuffled elements. *Genome Res*, 14(11):2336–2346, 2004.

[134] K. R. Rasmussen, J. Stoye, and E. W. Myers. Efficient q-gram filters for finding all ε-matches over a given length. *J Comput Biol*, 13(2):296–308, 2006.

[135] T. Rausch, A.-K. Emde, D. Weese, A. Döring, C. Notredame, and K. Reinert. Segment-based multiple sequence alignment. *Bioinformatics*, 24(16):i187–i192, 2008.

[136] K. Reinert, H.-P. Lenhof, P. Mutzel, K. Mehlhorn, and J. D. Kececioglu. A branch-and-cut algorithm for multiple sequence alignment. In *Proceedings of the first annual international conference on Computational molecular biology*, RECOMB '97, pages 241–250, New York, NY, USA, 1997. ACM.

[137] S. Richards, Y. Liu, B. R. Bettencourt, P. Hradecky, S. Letovsky, R. Nielsen, K. Thornton, M. J. Hubisz, R. Chen, R. P. Meisel, O. Couronne, S. Hua, M. A. Smith, P. Zhang, J. Liu, H. J. Bussemaker, M. F. van Batenburg, S. L. Howells, S. E. Scherer, E. Sodergren, B. B. Matthews, M. A. Crosby, A. J. Schroeder, D. Ortiz-Barrientos, C. M. Rives, M. L. Metzker, D. M. Muzny, G. Scott, D. Steffen, D. A. Wheeler, K. C. Worley, P. Havlak, K. J. Durbin, A. Egan, R. Gill, J. Hume, M. B. Morgan, G. Miner, C. Hamilton, Y. Huang, L. Waldron, D. Verduzco, K. P. Clerc-Blankenburg, I. Dubchak, M. A. F. Noor, W. Anderson, K. P. White, A. G. Clark, S. W. Schaeffer, W. Gelbart, et al. Comparative genome sequencing of Drosophila pseudoobscura: chromosomal, gene, and cis-element evolution. *Genome Res*, 15(1):1–18, 2005.

[138] E. J. Richardson and M. Watson. The automatic annotation of bacterial genomes. *Brief Bioinform*, 14(1):1–12, 2013.

[139] C. Rinke, P. Schwientek, A. Sczyrba, N. N. Ivanova, I. J. Anderson, J.-F. Cheng, A. Darling, S. Malfatti, B. K. Swan, E. A. Gies, J. A. Dodsworth, B. P. Hedlund, G. Tsiamis, S. M. Sievert, W.-T. Liu, J. A. Eisen, S. J. Hallam, N. C. Kyrpides, R. Stepanauskas, E. M. Rubin, P. Hugenholtz, and T. Woyke. Insights into the phylogeny and coding potential of microbial dark matter. *Nature*, 499(7459):431–437, 2013.

[140] M. S. Rosenberg. Multiple sequence alignment accuracy and evolutionary distance estimation. *BMC Bioinformatics*, 6:278, 2005.

[141] R. K. Saiki, T. L. Bugawan, G. T. Horn, K. B. Mullis, and H. A. Erlich. Analysis of enzymatically amplified beta-globin and HLA-DQ alpha DNA with allele-specific oligonucleotide probes. *Nature*, 324(6093):163–166, 1986.

[142] F. Sanger, G. M. Air, B. G. Barrell, N. L. Brown, A. R. Coulson, C. A. Fiddes, C. A. Hutchison, P. M. Slocombe, and M. Smith. Nucleotide sequence of bacteriophage Φ X174 DNA. *Nature*, 265(5596):687–695, 1977.

[143] F. Sanger, S. Nicklen, and A. R. Coulson. DNA sequencing with chain-terminating inhibitors. *Proc Natl Acad Sci U S A*, 74(12):5463–5467, 1977.

[144] D. Sankoff. Genome rearrangement with gene families. *Bioinformatics*, 15(11):909–917, 1999.

[145] D. Sankoff. The signal in the genomes. *PLoS Comput Biol*, 2(4):e35, 2006.

[146] D. Sankoff and P. Trinh. Chromosomal breakpoint reuse in genome sequence rearrangement. *J Comput Biol*, 12(6):812–821, 2005.

[147] A. S. Schwartz and L. Pachter. Multiple alignment by sequence annealing. *Bioinformatics*, 23(2):e24–e29, 2007.

[148] S. Schwartz, W. J. Kent, A. Smit, Z. Zhang, R. Baertsch, R. C. Hardison, D. Haussler, and W. Miller. Human-mouse alignments with BLASTZ. *Genome Res*, 13(1):103–107, 2003.

[149] S. Schwartz, Z. Zhang, K. A. Frazer, A. Smit, C. Riemer, J. Bouck, R. Gibbs, R. Hardison, and W. Miller. PipMaker–a web server for aligning two genomic DNA sequences. *Genome Res*, 10(4):577–586, 2000.

[150] E. Siragusa, D. Weese, and K. Reinert. Fast and accurate read mapping with approximate seeds and multiple backtracking. *Nucleic Acids Res*, 41(7):e78, 2013.

[151] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *J Mol Biol*, 147(1):195–197, 1981.

[152] P. J. Stephens, C. D. Greenman, B. Fu, F. Yang, G. R. Bignell, L. J. Mudie, E. D. Pleasance, K. W. Lau, D. Beare, L. A. Stebbings, S. McLaren, M.-L. Lin, D. J. McBride, I. Varela, S. Nik-Zainal, C. Leroy, M. Jia, A. Menzies, A. P. Butler, J. W. Teague, M. A. Quail, J. Burton, H. Swerdlow, N. P. Carter, L. A. Morsberger, C. Iacobuzio-Donahue, G. A. Follows, A. R. Green, A. M. Flanagan, M. R. Stratton, P. A. Futreal, and P. J. Campbell. Massive genomic rearrangement acquired in a single catastrophic event during cancer development. *Cell*, 144(1):27–40, 2011.

[153] E. Tannier, C. Zheng, and D. Sankoff. Multichromosomal median and halving problems under different genomic distances. *BMC Bioinformatics*, 10:120, 2009.

[154] The Chimpanzee Sequencing and Analysis Consortium. Initial sequence of the chimpanzee genome and comparison with the human genome. *Nature*, 437(7055):69–87, 2005.

[155] J. D. Thompson, D. G. Higgins, and T. J. Gibson. CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic Acids Res*, 22(22):4673–4680, 1994.

[156] W. T. Tutte. A short proof of the factor theorem for finite graphs. *Canadian Journal of Mathematics*, 6:347–352, 1954.

[157] S. Tweedie, M. Ashburner, K. Falls, P. Leyland, P. McQuilton, S. Marygold, G. Millburn, D. Osumi-Sutherland, A. Schroeder, R. Seal, H. Zhang, and F. Consortium. FlyBase: enhancing drosophila gene ontology annotations. *Nucleic Acids Res*, 37(Database issue):D555–D559, 2009.

[158] J. C. Venter, M. D. Adams, E. W. Myers, P. W. Li, R. J. Mural, G. G. Sutton, H. O. Smith, M. Yandell, C. A. Evans, R. A. Holt, J. D. Gocayne, P. Amanatides, R. M. Ballew, D. H. Huson, J. R. Wortman, Q. Zhang, C. D. Kodira, X. H. Zheng, L. Chen, M. Skupski, G. Subramanian, P. D. Thomas, J. Zhang, G. L. G. Miklos, C. Nelson, S. Broder, A. G. Clark, J. Nadeau, V. A. McKusick, N. Zinder, A. J. Levine, R. J. Roberts, M. Simon, C. Slayman, M. Hunkapiller, R. Bolanos, A. Delcher, I. Dew, D. Fasulo, M. Flanigan, L. Florea, A. Halpern, S. Hannenhalli, S. Kravitz, S. Levy, C. Mobarry, K. Reinert, K. Remington, J. Abu-Threideh, E. Beasley, et al. The sequence of the human genome. *Science*, 291(5507):1304–1351, 2001.

[159] L. Wang and T. Jiang. On the complexity of multiple sequence alignment. *J Comput Biol*, 1(4):337–348, 1994.

[160] M. S. Waterman and M. Eggert. A new algorithm for best subsequence alignments with application to tRNA-rRNA comparisons. *J Mol Biol*, 197(4):723–728, 1987.

[161] J. D. Watson. The human genome project: past, present, and future. *Science*, 248(4951):44–49, 1990.

[162] J. D. Watson and F. H. Crick. The structure of DNA. *Cold Spring Harb Symp Quant Biol*, 18:123–131, 1953.

[163] G. Watterson, W. Ewens, T. Hall, and A. Morgan. The chromosome inversion problem. *Journal of Theoretical Biology*, 99(1):1–7, 1982.

[164] D. Weese, A.-K. Emde, T. Rausch, A. Döring, and K. Reinert. RazerS–fast read mapping with sensitivity control. *Genome Res*, 19(9):1646–1654, 2009.

[165] D. B. West. *Introduction to Graph Theory*. Pearson, 2nd edition, 2000.

[166] W. J. Wilbur and D. J. Lipman. Rapid similarity searches of nucleic acid and protein data banks. *Proc Natl Acad Sci U S A*, 80(3):726–730, 1983.

[167] Y. Yamazaki, H. Niki, and J.-i. Kato. Profiling of Escherichia coli chromosome database. *Methods Mol Biol*, 416:385–389, 2008.

[168] S. Yancopoulos, O. Attie, and R. Friedberg. Efficient sorting of genomic permutations by translocation, inversion and block interchange. *Bioinformatics*, 21(16):3340–3346, 2005.

[169] Z. Zhang, P. Berman, T. Wiehe, and W. Miller. Post-processing long pairwise alignments. *Bioinformatics*, 15(12):1012–1019, 1999.

[170] Z. Zhang, S. Schwartz, L. Wagner, and W. Miller. A greedy algorithm for aligning DNA sequences. *J Comput Biol*, 7(1-2):203–214, 2000.

Appendix A

STELLAR parameters and implementation

A.1 Appropriate parameter values for ε and q

For the filtering algorithm to work, we require ε-matches to contain at least one q-hit. Then,

$$T(n, q, \varepsilon) \geq 1 \tag{A.1}$$

where $T(n, q, \varepsilon) = n + 1 - q \cdot (\lfloor \varepsilon n \rfloor + 1)$ as defined in the q-gram lemma (Lemma 1). To equation A.1, we apply the following simple transformations:

$$\begin{array}{rcl|l} T(n, q, \varepsilon) & \geq & 1 & \text{(Lemma 1)} \\ n + 1 - q \cdot (\lfloor \varepsilon n \rfloor + 1) & \geq & 1 & -1 + q \cdot (\lfloor \varepsilon n \rfloor + 1) \\ n & \geq & q \cdot (\lfloor \varepsilon n \rfloor + 1) & / \, (\lfloor \varepsilon n \rfloor + 1) \\ q & \leq & \dfrac{n}{\lfloor \varepsilon n \rfloor + 1} \, . & \end{array}$$

Since $\lfloor \varepsilon n \rfloor + 1 > \varepsilon n$,

$$\frac{n}{\lfloor \varepsilon n \rfloor + 1} < \frac{n}{\varepsilon n} = \frac{1}{\varepsilon}$$

and therefore we have to choose q and ε such that

$$q < \frac{1}{\varepsilon} \, .$$

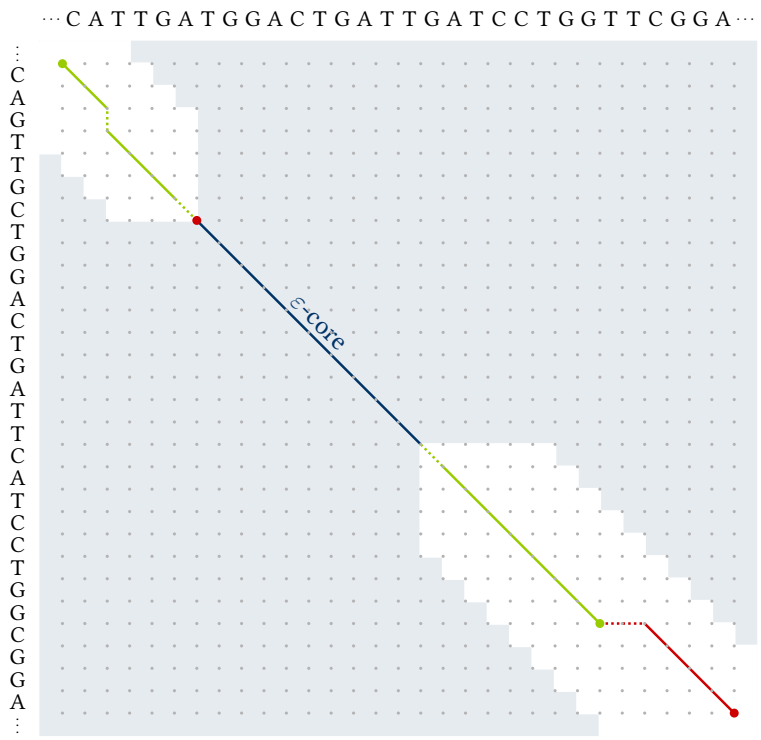

Figure A.1: A greedy approach might fail to identify the longest ε-match. Here, a possible greedy approach always cuts the end of the extended ε-core that is closer to an error until the error rate is $\leq \varepsilon$. In this example, it would return the ε-match between the two grey bars with a length of 24 ($\varepsilon = 1/8,\ n_0 = 20$). However, there is a longer ε-match (between the black bars) of length 25 with the same number of errors.

A.2 Overview of STELLAR source code

STELLAR makes use of mainly three modules from the SeqAn library: the `index` module for filtering, the `align` module for DP alignment algorithms, and the `seeds` module for ε-core extension. The main function `stellar()` in the file `stellar.h` uses the `Finder` interface from the `index` module specialized with `SwiftLocal` to identify SWIFT hits. For each SWIFT hit, `stellar()` calls the function `verifySwiftHit()`, which defines a `LocalAlignmentEnumerator<>` from the `align` module for banded Waterman-Eggert alignment. From the resulting ε-cores, which are of the type `Align<>`, ε-X-drops are filtered with the function `splitAtXDrops()`. This function implements the post-processing algorithm, which is not integrated in the the SeqAn library, yet. The verification steps 3 and 4, extension and identification of ε-matches, are carried out in `extendAndExtract()`. This function defines a `Seed<>` for each ε-core and extends it using `extendSeed()` from the `seeds` module. Afterwards, `extendAndExtract()` calls the function `bestExtension()` to detect ε-matches. The computation of additional values in DP alignment algorithms was not possible with SeqAn at the time of implementation, so a customized banded alignment was added for Step 4a. The function `longestEpsMatch()` carries out Step 4b. The traceback (Step 4c) is taken from the SeqAn library again. When an ε-match is found, STELLAR adds it to a `String` of `StellarMatch<>` objects with the function `insertMatch()`. `insertMatch()` calls `maskOverlaps()` to remove overlapping ε-matches if a preset number of ε-matches has been inserted. At the end, the ε-matches are passed to the function `outputMatches()`, where e-values are computed and the matches are written to a file.

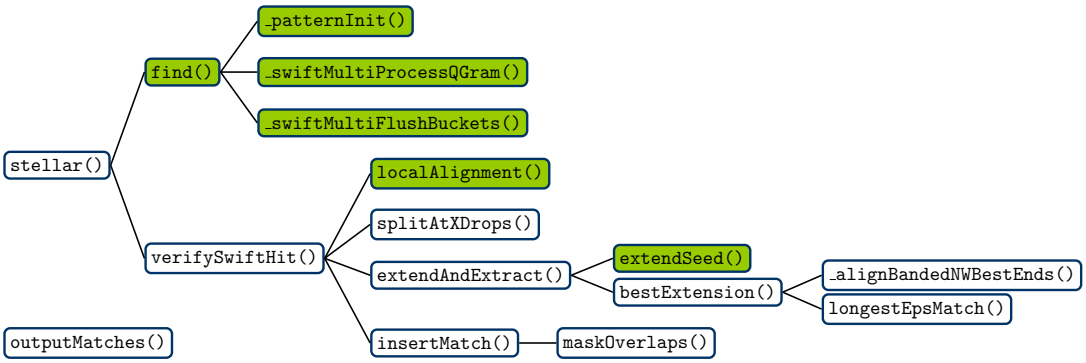

Figure A.2: Overview of functions in the STELLAR implementation. Functions shaded in green are already part of the SeqAn core library.

Appendix B

Generalization of Enredo graphs

The definition of Enredo graphs in Section 4.1.3 differs from its original definition [124] in some aspects. Paten et al. define adjacencies as non-empty segments, their initial Enredo graph is not a multigraph, and they use a length threshold for adjacencies [124]. The following paragraphs describe each of these differences to the definition in Section 4.1.3 in more detail.

Non-empty adjacencies. Section 4.1 assumes that the set of blocks $\mathcal{B}$ is a tiling of the genomes. Segments that are not aligned to any other segments appear in $\mathcal{B}$ as blocks of size 1. However, the Enredo method constructs the Enredo graph structure from a set of preferentially short alignments (to avoid overlaps). Thus, the blocks cover only parts of the genomes and adjacencies can have arbitrary lengths.

In Section 4.1, adjacencies that have a length > 0 are replaced by blocks that have a single occurence. Transferred to the Enredo graph structure, a single adjacency edge is replaced by a component consisting of a block edge between a head and a tail vertex connected to the rest of the graph structure by two directed edges that represent empty adjacencies (see Fig. B.1).

Multigraph. Paten et al. represent multiple adjacency edges between the same two vertices by a single adjacency edge in the initial phase of the Enredo method. This is a requirement to

Figure B.1: Replacement of a non-empty adjacency by a block B of size one in an Enredo graph. The labeled adjacency edge (left) becomes a block edge with two vertices and two adjacency edges (right).

distinguish them from additional adjacency edges created in later steps of the Enredo method. In addition, they assume the initial adjacencies to be homologous segments.

Section 4.1.3 expects all sets of segments that are assumed to be homologous being defined as blocks.[1] Given this, it is possible to generally allow multiple adjacency edges between the same two vertices. Segments assumed to be homologous are represented as a single block edge and segments assumed to be non-homologous are represented as separate adjacency edges.

Length threshold. The original Enredo graph structure does not represent adjacencies that are longer than a predefined length threshold. The Enredo method only adds edges to the graph structure that represent shorter adjacencies. This results already in a partial segmentation of the genomes. Section 4.1.3 defines the graph with all adjacencies and leaves segmentation to later stages.

[1] According to the definitions in Chapter 2, the only formal difference between a set of homologous adjacencies and a block are the orientation bits in block occurences. Once a set of adjacencies is assumed to be homologous, these bits can usually be added with minor sequence comparison effort.

Appendix C

Supplementary figures of breakpoint analysis

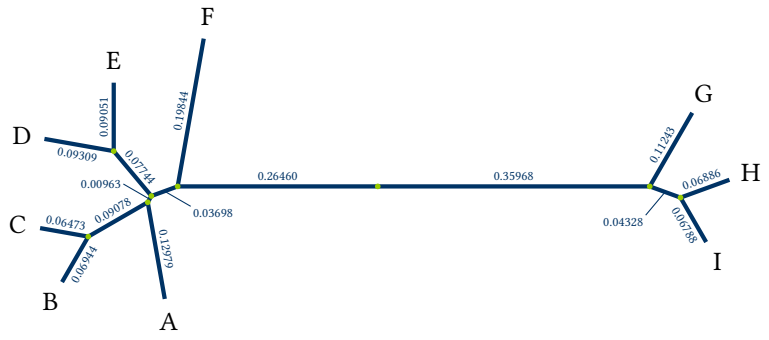

Figure C.1: The phylogeny of nine taxa used for the simulation of alignment problems in Section 5.3.1.

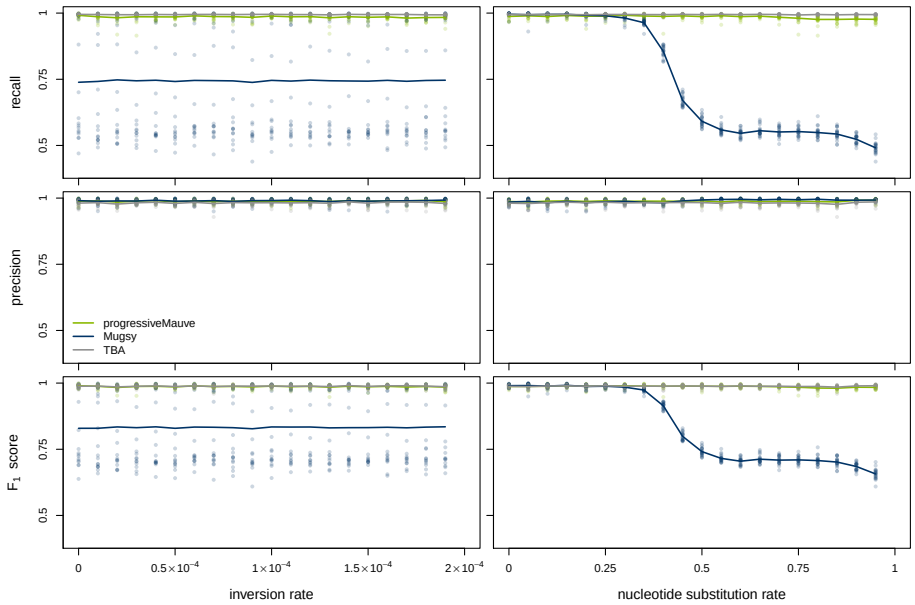

Figure C.2: Nucleotide accuracy in calculated alignments of the data set *InvNt*. Each point corresponds to one calculated alignment. The two sides display the same data by different mutation rates: by inversion rates (left) or by nucleotide substitution rates (right). The lines indicate the average recall (top), precision (middle), or F_1 score (bottom) per tested alignment program.

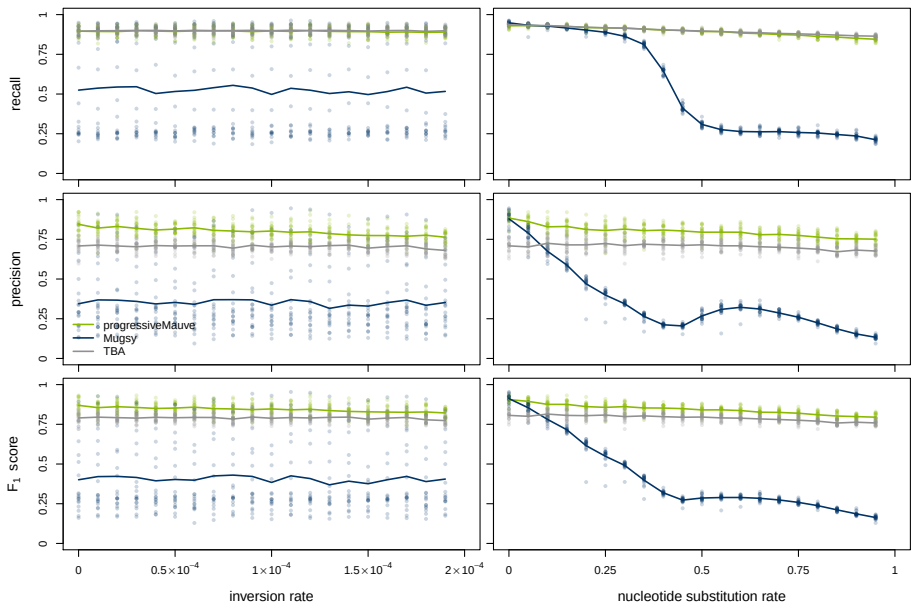

Figure C.3: Indel accuracy in calculated alignments of the data set *InvNt*. Each point corresponds to one calculated alignment. The two sides display the same data by different mutation rates: by inversion rates (left) or by nucleotide substitution rates (right). The lines indicate the average recall (top), precision (middle), or F_1 score (bottom) per tested alignment program.

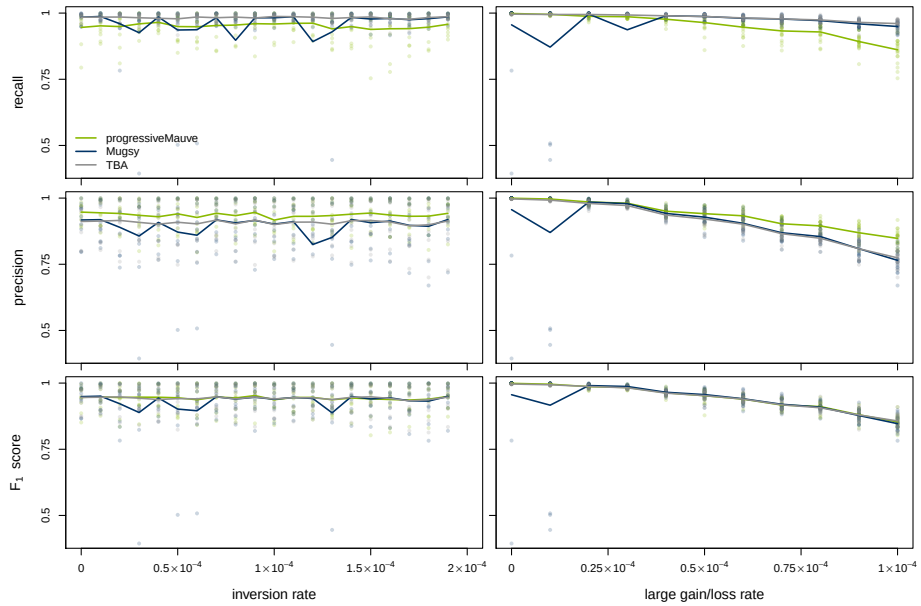

Figure C.4: Nucleotide accuracy in calculated alignments of the data set *InvGL*. Each point corresponds to one calculated alignment. The two sides display the same data by different mutation rates: by inversion rates (left) or by large gain/loss rates (right). The lines indicate the average recall (top), precision (middle), or F_1 score (bottom) per tested alignment program.

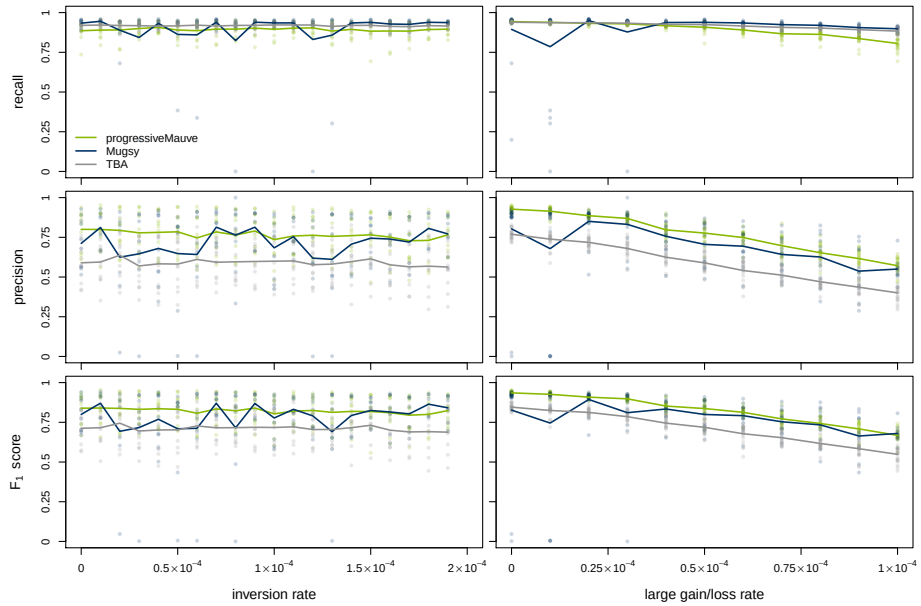

Figure C.5: Indel accuracy in calculated alignments of the data set *InvGL*. Each point corresponds to one calculated alignment. The two sides display the same data by different mutation rates: by inversion rates (left) or by large gain/loss rates (right). The lines indicate the average recall (top), precision (middle), or F_1 score (bottom) per tested alignment program.

Index

T

U

V

W

X

Curriculum Vitae

For reasons of data protection, the Curriculum Vitae is not published in the online version.

Eigenständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel und Quellen verwendet habe.

Berlin, den 19.11.2013 (Birte Kehr)