Bioinformatic Approaches for Genome Finishing

Ph. D. Thesis

submitted to the
Faculty of Technology,
Bielefeld University, Germany
for the degree of Dr. rer. nat.

by

Peter Husemann

July, 2011

Referees:

Prof. Dr. Jens Stoye
PD Dr. Andreas Tauch

Gedruckt auf alterungsbeständigem Papier nach DIN-ISO 9706.
Printed on non-aging paper according to DIN-ISO 9706.

To my brother.

Contents

1 Introduction **1**

2 Sequencing Technologies – Biological Background **5**
2.1 DNA – The Backbone of Life on Earth 5
2.2 Technologies to Assess the Sequence of DNA 8
2.2.1 Traditional: Sanger Sequencing 8
2.2.2 The 'Next' Generation: Massively Parallel Sequencing 9
2.2.3 Third Generation: Single Molecule Sequencing 13
2.3 Genome Sequencing 15
2.3.1 Shotgun Sequencing 15
2.3.2 Assembly Phase 17
2.3.3 Genome Finishing 18

3 Efficient Matching of Contigs **23**
3.1 Finding Local Similarities 24
3.1.1 Smith-Waterman 25
3.1.2 Seed and Extend Heuristics 27
3.1.3 Search Space Filtering 28
3.2 Matching by Filtering with q-Grams 29
3.2.1 General Idea 29
3.2.2 Building an Index of the Reference Genome 30
3.2.3 Filtering for Similarities 31
3.3 *r2cat* – The Related Reference Contig Arrangement Tool 33
3.3.1 Matching 34
3.3.2 Visualization 37
3.3.3 Simple Contig Mapping 39

4 Advanced Contig Layouting using Multiple Reference Genomes 43
4.1 The Contig Adjacency Graph . . . 44
4.1.1 Notation . . . 44
4.1.2 Weighting the Adjacency Edges . . . 47
4.1.3 Creating a Basic Contig Adjacency Graph . . . 48
4.2 Finding a Layout of the Contigs . . . 50
4.2.1 Traveling Salesman Tour Through the Graph . . . 50
4.2.2 Fast Adjacency Discovery Algorithm . . . 52
4.3 Enhancements of the Graph Creation . . . 53
4.3.1 Including Phylogenetic Distances . . . 53
4.3.2 Integrating Additional Information . . . 54
4.4 Variations of the Contig Layouting . . . 54
4.4.1 Handling Rearrangements . . . 55
4.4.2 Repeat-aware Layouting . . . 55

5 Realization of the Software 63
5.1 Implementational Milestones . . . 63
5.1.1 *r2cat* . . . 63
5.1.2 *treecat* . . . 64
5.1.3 *repcat* . . . 65
5.1.4 *htscat* . . . 66
5.2 External Software and Libraries . . . 68
5.2.1 FreeHEP Graphics Export . . . 68
5.2.2 Graphviz . . . 69
5.2.3 NetBeans Platform . . . 69
5.2.4 Prefuse Graph Visualization . . . 69

6 Layouting Corynebacteria Contigs 71
6.1 Background and Preparatory Steps . . . 71
6.1.1 Description of the Datasets . . . 71
6.1.2 Determining a Reference Layout . . . 74
6.1.3 Parameter Estimation for the Contig Adjacency Graph . . . 74
6.1.4 Other Software for Contig Layouting . . . 77
6.2 Evaluation on Real Sequencing Data . . . 79
6.2.1 Single Reference Based Ordering . . . 80
6.2.2 Multiple Reference Based Layouting . . . 83
6.2.3 Layouting Repetitive Contigs . . . 90

7 Summary and Outlook 95

Acknowledgments 99

Bibliography 101

Chapter 1

Introduction

Molecular biology and bioinformatics are entangled by a fortunate synergetic effect: Advances in biology provide more and other types of data which again drive the development of computer aided analyses. Advanced analyses, in turn, help to interpret and understand the data and thus lead to more sophisticated theories, and eventually result in new experiments providing further data. The research field of *genome analysis*, for example, progressed tremendously through this effect.

Genomic sequences are considered having a huge potential to gain deeper insight into the life of organisms, and to shed light on the inheritance of their properties. Accordingly, techniques for the *sequencing of genomes* – that is the determination of their nucleotide sequence – have evolved during the last decades. Especially the recent high throughput techniques increased the availability and enhanced the simplicity of sequencing considerably.

Although the popular press occasionally confuses the *determination* of a complete genome sequence with a *deciphering* of the corresponding genome, there is a big difference. Genomic data are complex and their analysis is often challenging. Further, the plain sequence of nucleotides is not all-encompassing, as the finding of epigenetic information, such as methylation patterns, demonstrates.

Yet, knowing the genomic sequence allows more detailed analyses than observing only the visible characteristics of a species. In the investigation of a newly sequenced genome, the locations of putative genes can be searched, or already known genes can be annotated. Approaches exist which try to computationally fold the proteins that these genes encode in order to predict their functions.

Besides genes, other functional sequences exist in a genome which are sometimes recognizable by special nucleotide patterns, so called motifs. Sophisticated motif discovery programs can detect, for example, transcription factor binding sites which play a role in the regulation of gene expression.

The subject of *comparative genomics* profits from the increasing number of already sequenced genomes. As early as the 18th century, Carl Linnæus tried to order and

categorize life. However, the availability of genomic sequences in current times undoubtedly boosts the accuracy at which a "tree of life" according to Darwins theory of evolution can be (re)constructed. Opposed to a distinction of species based on visible characteristics, nucleotide sequences provide a much finer granularity. This even resulted in taxonomic reclassifications of some species after sequencing their genome revealed that they actually belong to another genus.

Comparing the gene content of different species can also provide hints on their evolutionary relationship. To this end, similarities of genomic regions are determined by aligning the corresponding sequences. Genes with a similar sequence that occur in distinct genomes are commonly presumed to be homologous which means that they were obtained from a common ancestor.

If homologous genes of a set of species are given, then, on a more abstract level, clusters of genes can be searched in which the genes often appear in near proximity across species. These gene clusters might indicate a certain evolutionary pressure to stay together, and possibly even imply a functional relation.

Due to evolutionary processes, sometimes genomes become rearranged. That is, blocks of DNA are displaced, or reversed, foreign pieces are included, or fragments of the genome get lost or are duplicated. These rearrangements can be studied at the level of homologous sequences to answer the question which 'operations' may have caused a rearrangement structure that is observed between two species.

Rearrangements can also be investigated at nucleotide level. In bacteria, for instance, an inversion can possibly disable a gene, rendering an originally pathogenic germ completely harmless. Vice versa, a toxin producing gene can be introduced into usually innocuous species.

Comparing the genomes of individuals belonging to the same species can help to gain deeper understanding as well. For example, in humans minor changes in the genome, the so called single nucleotide polymorphisms (SNPs), are suspected to be involved in cancer development.

In biotechnology, bacteria are grown to produce desired compounds or enzymes, and usually the production strains have undergone a process of artificial selection to increase their efficiency. The comparison of such a production strain with the wild type it originated from may help to comprehend which genomic changes led to an enhanced efficiency.

First attempts were successful to create a synthetic 'minimal' genome that contains only the genes necessary to proliferate. This opens the door to fabricated species containing several genes of different species, almost like building blocks, to generate certain wanted phenotypes or properties.

All these examples show an impressive development. Nevertheless, the analysis of genomes has not tapped its full potential. Coming back to the initially mentioned synergetic effect, maybe some of computer aided analyses will help to find new aspects worth investigating which then open new fields of research. Genomic

sequences are surely a key component in this process, and having more of them available also drives the development of methods to analyze them.

Sequencing itself is becoming a routine task. Yet, all current sequencing approaches still have limitations, and the most striking one is the small read length: To date, only a few hundred bases can be read consecutively from genomes that are typically several orders of magnitude larger.

Improvements in chemistry, methodology, and machinery in the latest developments lead to longer reads, an easier handling of the experiments, and a massive parallelization of the sequencing and therefore also to higher throughput. Though, it has not changed that in an assembly phase many overlapping reads of a sequencing run have to be stitched together.

The goal of this assembly phase is to obtain a single and complete nucleotide sequence of the genome – or of each chromosome in the case of multi-chromosomal species. Unfortunately, very often this goal is impossible to reach, or at least its outcome is unreliable. Instead, several contiguous stretches of nucleotides, the so called *contigs*, are reconstructed. Missing or unreliable read information leads to gaps that separate the contigs.

In the process of finishing, these gaps are resequenced in the lab in order to join the contigs to the desired complete sequence. It is helpful to know which of the contigs are adjacent to specifically pick the DNA in between for further sequencing. Instead of trying all possible adjacencies, sometimes the initial sequencing already provides hints as to which of the contigs are adjacent.

Another established approach is to use sequences of other genomes as a guiding reference. These *reference genomes* originate in general from closely related species that are assumed to have a high degree of synteny. By matching the contigs onto the reference and analyzing the matches, adjacencies of the contigs can be estimated.

The work presented in this thesis is based on the concept of using reference genomes to estimate adjacencies of contigs. We present algorithms and approaches applicable for one or several reference genomes and compare our performance in an evaluation with real sequencing data. Some parts of this thesis have already been published in advance [41, 42, 43].

Overview of this Thesis The following Chapter 2 provides the biological background of this work by describing the role of genomic sequences and the processes to acquire them. This includes information about recent high throughput procedures, as well as a description of the typical work-flow to sequence whole genomes.

Chapter 3 gives an introduction how to compute similarities between sequences and proposes a method which we use for matching contigs onto reference genomes. Additionally, the application *r2cat* is presented that can be used to create synteny plots and to arrange a set of contigs according to matches onto a reference genome.

The subsequent Chapter 4 contains the main methodical contribution of this thesis. It defines the concept of a *contig adjacency graph* which we use to collect information about neighboring contigs from related reference genomes. We give the algorithms to compute this graph and a heuristic to extract the most promising adjacencies of the contigs. Additionally, we discuss variations to include additional information and provide enhancements to deal with the special case of repetitive contigs that map non-uniquely to the reference genomes.

In Chapter 5, we describe how the software implementing our ideas has evolved and which features are currently available. Further, we briefly introduce external software that we employ.

After this, we show an evaluation of our software in Chapter 6. To this end, we compare our implementations to other related approaches using contigs from real sequencing data.

Finally, Chapter 7 concludes and gives an overview of possible improvements and unintended applications of our methods.

Chapter 2

Sequencing Technologies – Biological Background

In this chapter, we want to provide the biological background for the contents of the remaining thesis. We start with an introduction to DNA and its role for the life sciences, and go on with procedures to access the information contained in DNA molecules, namely their nucleotide sequence. Finally, in Section 2.3, we explain in this context how genome sequencing projects work, and why they usually involve bioinformatics.

Some basic biological information presented in this chapter have been extracted from a textbook on molecular genetics [52], further information about the newer sequencing technologies were obtained from a recently published Ph. D. thesis [78].

2.1 DNA – The Backbone of Life on Earth

The deoxyribose nucleic acid (DNA) is a key component for life on earth. All living organisms contain thread-like DNA molecules which encode most, if not all, inheritable information. While first experiments and findings with respect to DNA can be dated back to the 18th century, the role as a material of inheritance was first published in 1944 by Oswald Avery and colleagues [8]. This publication sparked the interest of Erwin Chargaff and he began to investigate this promising molecule. He discovered that the DNA of all organisms feature the same amounts of the bases adenine (A) and thymine (T) as well as the same amounts of cytosine (C) and guanine (G) and postulated that these bases occur pairwise [21]. The chemical structure of the bases is shown in the upper part of Figure 2.1 on the following page. In 1953, James Watson and Francis Crick [102] published a suggestion for the structure of DNA molecules that is accepted to be valid until today. Inspired by X-Ray crystallography data of Maurice Wilkins [32] and Rosalind Franklin [107] – published in the same issue of *Nature* – Watson and Crick proposed a double helix as shown in

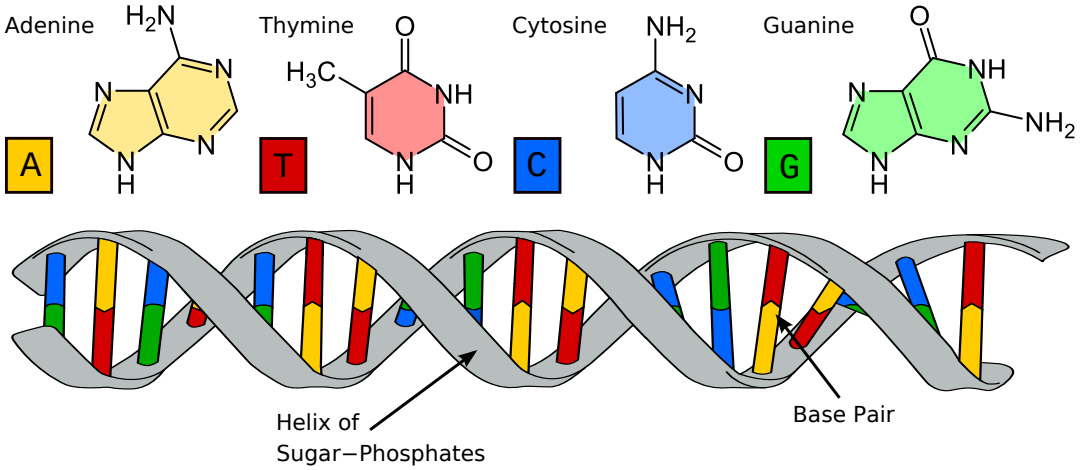

Figure 2.1: Chemical structure of the four nucleotides and molecular structure of DNA. Image is licensed under CC BY-SA 3.0 and was adopted from [106].

the lower part of Figure 2.1. The sugar-phosphate backbones of the helix are held together by hydrogen bonds of the bases on the inside of the helices, comparable to the steps of a ladder. In each base pair, they proposed, a *purine* base (`A` or `G`) would bind to a *pyrimidine* base (`C` or `T`). Together with Chargaff's observations, it was clear that `A` has to pair with `T`, and `C` pairs with `G`. In some understatement the authors wrote: "It has not escaped our notice that the specific pairing we have postulated immediately suggests a possible copying mechanism for genetic material". This pairing idea laid the foundations for several important discoveries like the process of DNA replication, the deciphering of the genetic code, and eventually the first sequencing of a human genome in 2001 [55, 99].

Each cell of any known organism on earth contains genomic DNA that is often given in several distinct molecules called *chromosomes*. The nucleotide sequences of the chromosomes can be separated into non-coding and coding stretches. The latter are termed *genes* and their nucleotide sequences describe blueprints for proteins or other functional molecules which can have a variety of functions, like the enzymatic digestion of other proteins or the regulation of the expression of other genes. Collectively, these functions of expressed genes determine the properties, the capabilities, and the appearance of that organism. In a nutshell, the sequence of nucleotides *encodes* the properties of its organism. It is thus desirable to use genomic sequences as a data source to predict genes and their functions, to discover regulatory pathways, or to investigate evolutionary relationships of the species on a molecular level.

The process of acquiring the sequence of bases is called *sequencing* and will be addressed in the next sections. Please note that the term 'sequencing' is often used ambiguously. Depending on the context, it either refers to the technical process of reading single bases from fragments of DNA molecules as addressed in the next

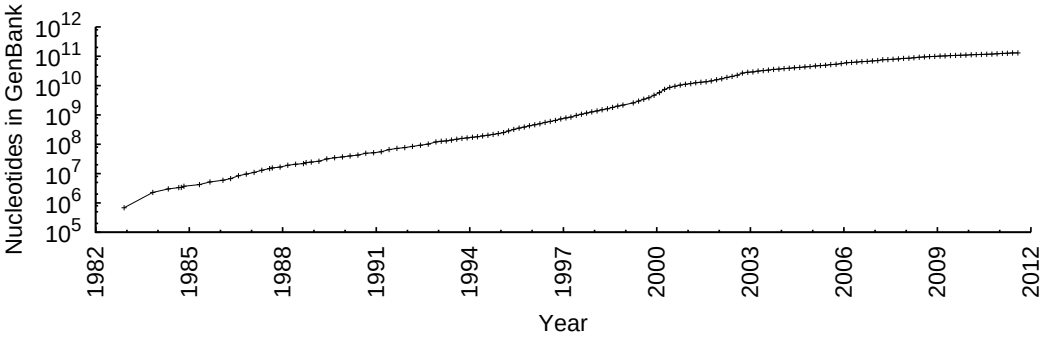

Figure 2.2: Size of nucleotide sequences stored in GenBank. Data from Section 2.2.8 of ftp://ftp.ncbi.nih.gov/genbank/gbrel.txt

section, or it stands for the combined work-flow of obtaining the complete genomic sequence of an organism as described in Section 2.3.

Although genomic DNA molecules usually consist of two strands of nucleotides, it suffices to sequence and store one of them because the second strand is the reverse complement of the first one. The number of available genomic sequences stored in this way is constantly growing. This is in particular consolidated by Figure 2.2 that shows the growth of GenBank [12, 104], a central database for genetic sequences hosted by the National Center for Biotechnology Information (NCBI).

The interest of scientists in these DNA sequences is diverse: In the prokaryotic domain that contains unicellular organisms without a true cell nucleus, like for example bacteria, some species are sequenced to increase their biotechnological efficiency. In *Corynebacterium glutamicum* the genome sequence helped to understand the metabolomic pathway such that the glutamate production could be optimized [50]. Furthermore, pathogenic bacteria like *Bacillus anthracis* [77] or *Mycobacterium tuberculosis* [23] are investigated to gain insight about their deathly effects.

Eukaryotic genomes feature, in contrast to prokaryotes, a much larger and more complex genome. Since the effort to sequence such genomes is still rather high, only few are sequenced completely, which then for example serve as model organisms. Analyzing for instance the genomic sequence of the model organism *Arabidopsis thaliana* [64] has the potential to reveal deeper insights into metabolic pathways which might be common in other plants. In the evolving field of personalized medicine, it is imaginable to use the genome of patients to design drugs specifically tailored to their needs.

A by-product of the availability of genomic sequences is a new dimension of accuracy in evolutionary trees. While initially the trees to order and categorize life were constructed based on phenotypical observations, today the available genome sequences provide a much finer granularity for phylogenetic reconstruction.

2.2 Technologies to Assess the Sequence of DNA

As mentioned before, knowing the genomic sequences of species has an invaluable impact on the understanding of biology. Unfortunately, it is not possible with the current technologies to acquire the whole genome at once. All available methods are restricted to sequence only very small parts of a DNA molecule into so-called *reads*. These commonly have a length of up to 1,000 nucleotides depending on both the technology used and also on properties that are specific to the nucleotide sequence.

This section describes, in historical order, the most widely used approaches for sequencing DNA, and the following Section 2.3 explains the general way to cope with the read size limitations of these approaches. For a more detailed overview, the interested reader may look at [44] or [47]. Since there is a rapid evolution regarding techniques and methods, this overview can only be a snapshot of the current state.

2.2.1 Traditional: Sanger Sequencing

One of the first approaches to sequence DNA was the so-called chain-termination method, also known as *Sanger sequencing* [83]. Although the technique was developed over 30 years ago, it is – with some modifications – still used today. We will first elucidate the traditional approach and then briefly address the modifications.

In the traditional procedure, the DNA fragment to be read, also termed *template*, is first enriched in a process called *amplification* such that a high number of copies exists. On each single stranded template copy, a *primer sequence* is hybridized such that the complementary strand can be synthesized by an enzyme, the *DNA polymerase*. Starting from the primer sequence, in a *polymerase chain reaction (PCR)* the sequence of complementary nucleotides is elongated if sufficient deoxynucleotides (dNTP: d$\mathtt{A}$TP, d$\mathtt{G}$TP, d$\mathtt{C}$TP, and d$\mathtt{T}$TP) are available. If, however, a *di*deoxynucleotide (**d**dNTP) is incorporated, the chain reaction stops and it is not possible to include further dNTPs. This can be exploited in a statistical way to read the nucleotide sequence of the template.

To this end, four different reaction sets are prepared in parallel. Each one contains the four dNTPs where usually one is radioactively marked. Additionally, one type of nucleotide is provided as ddNTP in a small fraction to one of the four reaction sets. When adding the template DNA and the polymerase, the chain reaction begins.

Since randomly and with a small chance a ddNTP is incorporated, a collection of DNA fragments with different sizes of the complementary strand is produced. If the dideoxynucleotide of a reaction set is for example thymine, it is clear that the original strand has a complementary adenine at each stop position.

The strands are then denaturated and subsequently separated by size with gel electrophoresis, for each reaction set on a single lane. Fragments of the same size form a band where the shortest fragments have traveled the longest way on the gel. The bands of the radioactively marked complementary strands can then be

visualized by autoradiography. Each visible band corresponds to a single base at a certain position. By comparing the relative positions of the bands of all four lanes, the nucleotide sequence of the template can be recovered.

Today, the radioactive markers have been replaced by fluorescent dyes attached to the ddNTPs, featuring different colors for each base. This allows to combine the different lanes to a single capillary where the nucleotides are read with a laser scanner. With all improvements, this technique is able to produce reads of up to 1,000 bases and it is thus still in use in many labs to produce high quality reads.

Newer technologies, however, outperform the Sanger method in terms of totally sequenced bases per instrument run. The next two sections introduce some of these *high-throughput sequencing* (HTS) techniques and elaborate their advantages and disadvantages.

2.2.2 The 'Next' Generation: Massively Parallel Sequencing

In the last few years, several new approaches to read DNA have been established in the labs. The intentions for the development of these methods were to reduce the cost and to increase the throughput of a single sequencing run in comparison to the Sanger method. Thus they are often referred to as *next-generation* sequencing (NGS) technologies. This term is adequate with respect to Sanger sequencing but, as we will see in Section 2.2.3, there are already successor techniques of the 'next' generation. That is why we call the technologies described in this section "*second-generation* sequencing approaches".

All of the modern approaches have in common that many fragments of the DNA are processed in parallel, yielding a very high number of reads per sequencing run. According to Magi *et al.* [62], second generation approaches follow the general pattern of first duplicating each fragment many times in an amplification step, while fixing the copies to a surface, and then reading the sequences of the copies by iterative cycles of enzymatic reactions and (mostly) imaging-based data collection.

The main disadvantage is that the generated reads are considerably shorter than the ones from the traditional approach. But it seems only to be a question of time until improvements in the chemistry and methodology enable the first HTS approaches to keep up with or even outperform Sanger with respect to the read size.

Table 2.1 lists the main features of three well established second-generation approaches. Each of them and a newcomer technology, for which these data were not available yet, will be explained in the following paragraphs.

Roche – 454 Sequencer

Introduced in 2005, the GenomeSequencer was the first commercially available second-generation sequencing platform. Until now, although with refined chemi-

Table 2.1: Comparison of the main features of three widely available second generation sequencing technologies; based on Magi *et al.* [62], updated according to the company websites as of June 2011.

	Roche 454	Illumina GA	ABI SOLiD
Amplification method	Emulsion PCR	Bridge PCR	Emulsion PCR
Sequencing method	Pyrosequencing	Reversible dye terminators	Sequencing by ligation
Read lengths	400 – 800 bp	35 – 150 bp	35 – 75 bp
Sequencing run time	0.4 d	2 – 14 d	1 – 7 d
Throughput per day	1 Gbp	6 – 55 Gbp	10 – 30 Gbp
Error rate	0.1%	1.5%	4%

cals, a sequencing by synthesis approach [63] is used that was developed by 454 Life Science[1] which is now a subsidiary of Roche Applied Science.

The 454 sequencing works as follows: First the DNA is fragmented randomly in small parts of 500 – 1,000 bases, for example by nebulization. Specific adaptor sequences are then appended to each fragment such that these can bind on the surface of specially prepared beads. Starting with a single fragment on a bead, the fragment is amplified to millions of copies in a process called *emulsion PCR*. There, the beads with fragment are distributed with PCR reagents in water-in-oil microreactors such that all fragments are amplified separately. The beads are then placed, together with DNA polymerase and further enzymes, on an optical array of fibers. The array consists of several million wells where a single well has the appropriate dimensions to hold exactly one bead.

After this preparatory step, the reading of the bases employs the *pyrosequencing* technique that was developed already in the 1980es [45]. Pyrosequencing is based on the principle that each incorporation of a nucleotide via DNA polymerase releases pyrophosphate. The pyrophosphate can then be detected by utilizing the enzymes sulfurylase and luciferase that produce, in a cascade of reactions, finally a light emission [70].

To read the sequence of the DNA templates that are immobilized to the beads, the different nucleotides are flowed cyclically in a fixed order (`T`, `A`, `C`, `G`) over the optical array. If in a cycle an `A` is provided, all template DNAs with a complementary `T` as next unpaired nucleotide will produce a light flash. The emitted light from the millions of copies of the template on a bead is strong enough to be detected with a CCD camera. While the picture taken is then used for basecalling, the remaining nucleotides are washed from the plate such that they do not interfere with the next step in the cycle.

[1] http://www.454.com

Problematic for this approach are *homopolymers* on the templates, which are regions where the same nucleotide occurs repeatedly. For few nucleotides, the emitted light is approximately proportional to the number of integrated bases. While two or three bases are clearly distinguishable, basecalling is not reliable for homopolymers of length six or more bases due to a saturation in the detector. Thus, 454 reads tend to have insertion or deletion errors in homopolymer stretches [63]. However, substitution errors are rare [62].

Illumina/Solexa – Genome Analyzer

One year after 454, the Solexa sequencing platform came on the market. In 2007, Solexa was acquired by Illumina,[2] which started an extensive advertising campaign resulting in the Genome Analyzer currently being the most frequently used second-generation sequencing device [62].

The sequencing technology, based on the work of Turcatti *et al.* [96], is largely similar to 454's but has a few essential differences. The DNA is also fragmented but the amplification occurs directly on the surface of a solid planar glass slide, the *flow cell*, with a technique called *bridge PCR* [1, 31]. During this process, the templates are covalently attached to the surface and form bridge-like structures. After several repetitions of bridge PCR, the flow cell consists of spots that contain millions of copies of each fragment.

The reading of the bases is done, for each spot of the flow cell, again in a highly parallel fashion. The difference is that a reversible terminator chemistry is employed for sequencing. For each type of base, the terminator is fluorescently labeled with a different color, which allows to provide all bases simultaneously in one reading cycle. After incorporating a single labeled base, the polymerase function stops due to the terminator which effectively avoids the problems with homopolymers that are present in the 454 technique. In the next step, all fluorescent dyes are activated by a laser and an image is recorded which is then used to call the bases in the different spots. While the nucleotide stays in place, the label and the terminator are subsequently removed such that the process can be repeated for the next base. Unfortunately, there is an increasing rate of erroneous base calls towards the end of Illumina reads since an incomplete removal of the label or terminator can happen, which hinders a reliable detection of the following bases.

Although the read sizes generated with this approach are substantially shorter compared to 454, see Table 2.1, Illumina's Genome Analyzer finds a variety of applications and thus also customers since the sequencing costs are lower and the total throughput per run is much higher. Preferred applications include resequencing, gene expression analysis and single nucleotide polymorphism (SNP) detection.

[2] http://www.illumina.com

We have described the two above mentioned sequencing techniques in little more detail since they are actively used in the Center for Biotechnology (CeBiTec) at Bielefeld University. Our labs sequence mostly the genomes of prokaryotes but currently the purchased Illumina and 454 sequencing devices are also used to sequence the eukaryotic genome of a Chinese hamster.

The sequencing methods described in the following are not (yet) established in Bielefeld. Nevertheless, we give a short review to address advances and unique points of the other available or currently developed methods.

Applied Biosystems – SOLiD

The SOLiD system [85] is based on sequencing by ligation; its acronym stands for **S**upported **O**ligonucleotide **Li**gation and **D**etection. The SOLiD sequencing device is commercially available since 2007 and like the previous techniques widely spread. Currently, Applied Biosystems[3] that is a part of Life Technologies performs the marketing and sells the ABI SOLiD sequencer.

Like 454, an emulsion PCR is used for the amplification of the fragments but instead of the enzyme polymerase, a DNA ligase is employed in order to read the bases. The ligase is able to incorporate oligonucleotides – for example octamers that consist of eight nucleotides – into the complementary single stranded template DNA. While the first two bases of each octamer are known and colorcoded with four fluorescent dyes of different color, the remaining six random bases are needed to stabilize the binding. Although the four colors ambiguously encode the 16 possible di-nucleotides, they are chosen in a way that knowing the first base and the color of a di-nucleotide uniquely identifies the second base.

To assess the sequence, the labeled octamers are ligated to the template DNAs and their color code is called. This has to be repeated in several rounds with varying starting positions such that every base is covered twice. Together with the color coding, this serves as an error correction to improve the accuracy of the base calls.

Compared to the other second-generation techniques in Table 2.1, ABI produces a very high throughput per day but also features the shortest reads. The rather high error rate in the table is given with respect to the raw color calls and can be reduced via the implicit error correction.

Ion Torrent – PGM Sequencer

The company Ion Torrent,[4] also acquired by Life Technologies, launched towards the end of 2010 the Personal Genome Machine (PGM). While the fragment preparation and the cyclic nucleotide washing steps of the PGM Sequencer are comparable

[3] http://www.appliedbiosystems.com
[4] http://www.iontorrent.com

to the 454 technology, the incorporated nucleotides are not recognized optically. Instead, hydrogen ions are measured that are released as a by-product every time the polymerase incorporates a nucleotide. In sequential washing steps of the different nucleotide types, the charge of the released ions is measured for many DNA fragments in parallel on an array chip that contains a layer of semiconductor sensors. Since in one washing step several nucleotides can be included into the fragment, this technique suffers also from the homopolymer problem.

2.2.3 Third Generation: Single Molecule Sequencing

In the aforementioned procedures, every piece of DNA has to be amplified such that the resulting copies produce a signal strong enough to be detected. However, the amplification process has some severe drawbacks. Besides being time-demanding and also requiring costly chemicals, the amplification process shows an effect called *amplification bias*. Depending on the primer sequences and the PCR settings, some nucleotide sequences are more difficult to amplify than others. This results in a varying abundance of the copied fragments that is also manifested in a varying coverage of reads. With fewer reads it is harder to eliminate sequencing errors. A technique of reading the bases directly from a single molecule is thus desirable since it avoids amplification biases. First ideas of such a *Single Molecule Sequencing* (SMS) can be dated back to 1989 [49]. Nevertheless, there are several approaches still under development. In the remainder of this section, we present three techniques with a high potential to become the *third generation* of sequencing devices. Thompson and Milos give a more detailed review about SMS techniques [92].

Helicos – Heliscope

The first commercially available sequencing instrument that uses the SMS technique is the Heliscope from Helicos Biosciences.[5] The sequencing approach was initially described in 2003 [16] and has already been applied to sequence a viral genome [38] as well as an individual human genome [75].

The sequencing is similar to Illumina's approach: First, millions of random fragments of the genomic DNA are immobilized to a glass slide, the difference is however that no amplification is needed. Again, fluorescently labeled bases with reversible terminators are incorporated and an image is taken with a high-resolution optical microscope [78].

Reads of this method are with an average of 32 bases rather short and also affected by deletions [75], since it is likely that the emitted signal of a single base is missed by the detector.

[5] http://www.helicosbio.com

Pacific Biosciences – PacBio *RS*

The company Pacific Biosciences[6] has developed a **s**ingle **m**olecule sequencing technology that works in **r**eal-**t**ime (SMRT), meaning as fast as a DNA polymerase can include nucleotides into single stranded DNA [30]. The device is called PacBio *RS*; first commercial units were delivered in mid 2011.

The SMRT approach immobilizes, in contrast to the other HTS procedures, the DNA polymerase instead of the DNA template. Each polymerase is anchored to the bottom of a nanoscale reaction well called *zero mode waveguide* (ZMW) and loaded with a DNA template. The wells are flooded with fluorescently labeled nucleotides, each type in a different color, and illuminated by a laser from below. The dimensions of the ZMWs are chosen in such a way that the laser light cannot travel directly through the well and only illuminates its bottom. However, the wavelengths of the fluorescent dyes, when activated by the laser, are able to reach the top of the well and can be recorded by a camera. During its incorporation by DNA polymerase, any labeled nucleotide stays significantly longer at the bottom of a ZMW than by diffusion alone. This allows the SMRT technique to track the nucleotides that are incorporated at the same speed of the employed polymerase. Neither washing of different nucleotides, nor terminators are needed. The fluorescent labels are cleaved during the incorporation such that they do not interfere with new nucleotides.

This rather distinct technology to read nucleotides offers interesting variations to the general sequencing process: It is possible to circularize the template after it has been loaded to the polymerase. Doing this allows to sequence the *same physical piece* of DNA several times such that sequencing errors can easier be detected. Another novel idea is the so called *strobe reading*. Here, the laser is switched off at certain time points while the polymerase continues to include nucleotides. Although obviously nucleotides are missed, the advantage is that the total processed molecule length is much longer since the polymerase gets less damaged by the laser. Strobe reads are consequently a set of smaller reads originating from the same template, thus providing additional information about their order and orientation.

Oxford Nanopore Technologies

The Nanopore technology does neither use fluorescent labels nor an optical recognition system like most of the other approaches. Instead, it measures individual bases electronically while they travel through *nanopores* which are proteins that create nanoscale holes. When an electrical current is applied to the nanopore, nucleotides cause a disruption of this current as they pass through the pore. Different nucleotides cause distinguishable changes in current and can thus be classified.

[6] http://www.pacificbiosciences.com

Even the distinction between the four standard bases and methylated cytosine is possible. This is a remarkable feature because methylation patterns have been shown to have an effect on the expression of genes [14].

To provide the necessary single nucleotides to the nanopore, this sequencing technique employs a DNA exonuclease which cleaves individual bases from the end of a single strand template DNA. The bases are then introduced to the nanopore and their current is measured. Clarke *et al.* describe the procedure and the chemical modifications to the nanopore that are needed for sequencing [22].

Oxford Nanopore Technologies Ltd.[7] developed an array chip to use nanopores in a parallel fashion. The label free detection and the use of an exonuclease promises comparably long reads. At the same time, the signal processing is simpler than an optical detection of bases and the electronics needed can possibly be fabricated cheaply after some further development. Though, a commercial launch is unknown as of June 2011.

In the future, the existing approaches will be developed further and will likely be cheaper and produce longer reads. Maybe new technologies will be invented accelerating the advances even further. However, a sequencing procedure that reads the *whole* DNA of a genome at once can not be expected in the near future. It is therefore still necessary to have methods to acquire the complete sequence of a genome despite of the limitations caused by small reads. These methods are introduced in the next section.

2.3 Genome Sequencing

As mentioned in Section 2.2, all current sequencing technologies face the severe limitation that the produced reads of the DNA molecules are much shorter than even small genomes. It is surprising that complete genomes can be obtained nevertheless. The method to achieve this is called *whole genome shotgun* (WGS) *sequencing*. Figure 2.3 on the next page gives an overview of the three basic steps of WGS sequencing: First the shotgun sequencing (a), then the computational assembly phase (b), and in the end the finishing of the genome (c). These processes are elaborated in more detail in the next three sections.

2.3.1 Shotgun Sequencing

The principle of *shotgun sequencing* [5, 88] is that the DNA molecule to be sequenced is over-sampled by generating reads at random positions. The name stems from the random pattern that shotgun projectiles produce when fired on a target. Overlapping parts of the reads can be used to reconstruct the genome sequence as described in Section 2.3.2.

[7] http://www.nanoporetech.com

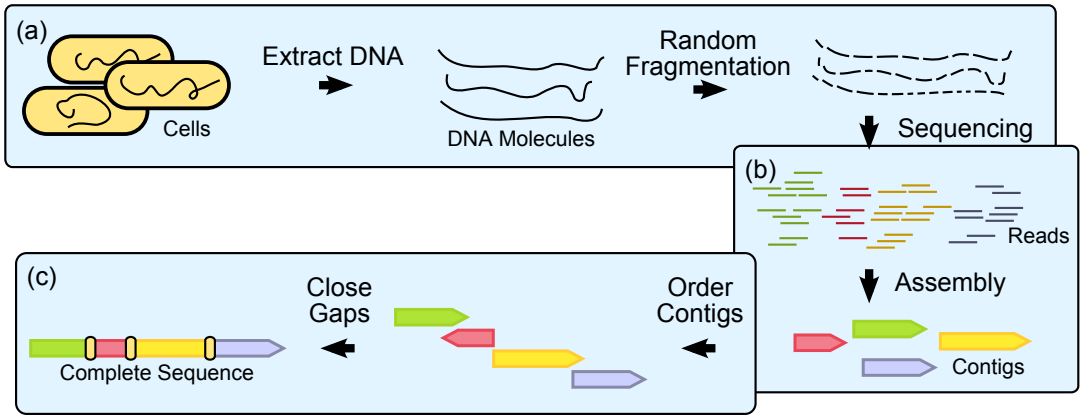

Figure 2.3: Major steps in *whole genome shotgun sequencing*: (a) Generating a set of reads during *shotgun sequencing*, (b) merging the reads to contigs in the *assembly phase*, and (c) completing the genome sequence in the *finishing step*.

To generate the set of overlapping reads, first the DNA has to be extracted and purified from the cells of the organism, see Figure 2.3 (a). Several copies of the genomic DNA are then fragmented – mostly using physical shearing like sonification or nebulization. Following that, the random fragments are sequenced, for example with one of the methods described in Section 2.2. The reads generated this way are ideally evenly distributed over the whole genome and their coverage is high enough such that each base of the genome will be found in several reads. In this context, current high-throughput sequencing techniques fit perfectly since they can cheaply generate a high number of reads and thus provide the desired coverage.

However, shotgun sequencing was invented already earlier in times of Sanger sequencing when each fragment was clonally amplified several times such that the sequencing would work. This was done using cloning vectors which are pieces of DNA that can be copied in host cells like *Escherichia coli*. Foreign DNA that is included into the vector, as a so-called *insert*, is copied along with it.

This cloning technique is sometimes employed even today when sequencing complex genomes: In the *hierarchical shotgun sequencing*, large fragments of the genome are 'stored' in a library of vectors, for example BACs (bacterial artificial chromosomes). The inserts of all those BACs that are needed to cover the whole genome are then sequenced independently with the shotgun approach. This can help to reduce problems in the assembly phase.

Another helpful variation is the *double barreled shotgun sequencing* that employs the so-called *paired end sequencing*. Here, the fragments are sequenced from both ends and the corresponding reads of one fragment are called *mate pair*. If all fragments have specified and uniform lengths, then the mate pair information can be very valuable for the following two phases of WGS sequencing. In the first versions of

the recent high-throughput devices, mate pairs were not available, however most of the companies developed changes in their sequencing protocols to provide them.

2.3.2 Assembly Phase

When the genome is covered sufficiently by reads, their overlaps can be used to reconstruct contiguous stretches of the genomic sequence that are called *contigs*. The process of generating the contigs from a set of reads, as depicted in Figure 2.3 (b), is referred to as the *assembly* of the genome, and it is a typical example for advances in biology gained through bioinformatics. Without the developed assembler programs, the sequencing of a complete genomic sequence would most likely not have been successful until today. While a comprehensive review of the challenges of assembly and how current assemblers deal with them is given in Miller *et al.* [65], we here only give an overview of two distinct classes of assembly algorithms.

First assembly algorithms date back to the times when few but relatively long Sanger reads were predominant. They are called *overlap-layout-consensus* approaches because they compare the reads all-against-all in order to merge overlapping parts into longer contiguous consensus sequences. The merging of reads to contigs is often done in a greedy fashion, sometimes by building an overlap graph. Some well known programs of this era are the *Celera* assembler [66] which was used to assemble one of the first available human genomes, *ARACHNE* [11], or *Bambus* [74].

One of the earliest assemblers that was able to cope with high throughput data, was the *Newbler* assembler [63, Supplem. material] that is shipped with 454 sequencing devices. Subsequently, *SSAKE* [100], *VCAKE* [48], and *SHARCGS* [29] were developed to handle short reads as well. The most problematic issue of these assemblers is the computational time needed for comparing the reads all-against-all.

A different class of assemblers solves this problem elegantly by using a so-called *de Bruijn graph* [17] for assembly [46], or more precisely a subgraph of it. The advantage is that the graph can be built in time linear to the input size, as opposed to the quadratic time that the overlap-layout-consensus approaches need in general. A de Bruijn subgraph consists of nodes representing all substrings of length k, called k-mers, of the reads. The nodes are connected by an edge if two overlapping k-mers occur adjacently in one of the reads. This way, common substrings are condensed and the overlaps of the reads are collected implicitly in the graph. If the sequencing data were perfect – that is without sequencing errors and with reads longer than repetitive regions of the genome – then the de Bruijn graph would reveal the complete genomic sequence: By following an Eulerian path that traverses every edge once, the desired complete sequence could be obtained. The de Bruijn graphs generated from real sequencing data are, however, much more complex, such that heuristics have been developed to cope with the limitations.

The increasing throughput of current sequencing technologies pushed the development of de Bruijn graph based assemblers forward since they are better suited to

handle relatively small reads and can cope with a higher amount of data due to the condensed nature of the graph. An incomplete list of well known de Bruijn graph assemblers is: *Velvet* [108], *EULER-SR* [20], *ALLPATHS* [18], and *SOAPdenovo* [60].

No matter which software is actually used, the assembly ideally results in a single sequence that represents the whole genome. In the case that the organism has several chromosomes, of course each of them should be given by a single contig. However, the desired ungapped sequence of the genome is in practice usually broken into several contigs for a variety of reasons: Fragments of the genome could just by chance not be present during the sequencing run. Also, it might occur that a single stranded fragment binds to itselves creating a hairpin loop that hinders the reading of the bases. Both cases result in a gap of a possible consensus sequence since no (overlapping) reads do exist.

One of the most common reasons for a fragmentation of the consensus sequence are repetitive parts of the genome [89]. Reads originating from different copies of a repetitive region within the genome can not safely be distinguished and thus are assembled to a single contig. We call those *repetitive contigs*. It is often easy to detect a repetitive contig by looking at the coverage of reads that were used to assemble it. A contig is likely to be repetitive, if its coverage is considerably higher than the average (or median) of all contigs. Sometimes, it can even be estimated how often a repetitive contig occurs within the genome by comparing its coverage to the median coverage. Unlike in cases when pieces of sequence are missing, here the information which sequences the repetitive contig belongs to is in general given. The difficulty for the assembler is to untangle the repeating copies because the consensus sequences that connect to non-repetitive contigs are not uniquely distinguishable. However, mate pair information of the reads, if available, can be included to disambiguate the occurrences of the reads and thus to reduce the number of contigs.

2.3.3 Genome Finishing

The assembly phase usually ends up in a set of contigs that need to be connected to obtain the complete sequence without gaps, see Figure 2.3 (c). This phase is called the *finishing* of the genome and, in general, it comes along with additional lab work. According to Nagarajan *et al.* [67], an initial draft assembly of a bacterial genome can be produced in weeks, while its completion currently takes months or even years. The process of finishing the genome is very costly, and thus there have already been debates whether good draft genomes are sufficient for most genomic analyses or if the increased effort for finishing is worth it. An argument in favor of the latter is that finished genomes allow a much richer analysis of the genome, for example in comparative approaches and order based genomic studies. Additionally, the finishing process can also help to improve the quality of the sequences, if for example mis-assembled or low coverage regions are examined closer.

Genome finishing typically consists of three parts which might be applied iteratively. First, the contigs need to be ordered such that gaps between adjacent contigs can be spotted. In the gap closure phase missing sequence information to close potential gaps is sequenced with the aim to merge the corresponding contigs. Finally, and sometimes optionally, the resulting assembly is validated in a polishing phase.

Contig Ordering

In the step of *contig ordering*, information about the order and relative orientation of the contigs is gathered. This is valuable for the gap closure where the gaps are 'selected' by specific primer sequences at the end of the contigs. Pairs of these primers determine a part of the genome that can be amplified and subsequently be sequenced. Given n contigs and no further information about their order, there are $O(n^2)$ possibilities for primer pairs which would amplify differently sized parts of the genome which do not necessarily belong to the missing gap sequences. If, however, the order of the contigs is known, the number of necessary primer pairs to investigate can be reduced to $O(n)$ [79]. A decrease of pairs to be considered results in less time and money for the lab work that is necessary to fill the gaps.

In the following, we refer to information regarding the order and the relative orientation of a set of contigs as their *layout*. Synonyms for layout are *supercontig*, *metacontig* or *scaffold* which might have a subtle difference in meaning but always refer to linking information of the contigs. There is a variety of methods and sources that help to estimate a suitable layout of the contigs. Here, we describe the ones most commonly used:

- Linking information between contigs can be extracted from the output of certain assembly programs like *Newbler* or the *Celera* assembler [67]. If the particular assembly algorithm builds a graph of the reads and their overlaps in order to merge them to contigs, then this graph often implicitly contains hints for a layout of the contigs. The assembled contigs are usually created in a very conservative fashion such that their consensus sequence is supported by a high coverage of reads. However, the graph might also contain faint hints for the adjacency of contigs if overlapping reads do connect them but with a coverage that is too low to be reliable. While the cautious behavior of the assembly algorithms makes sense to avoid misassemblies, in some of the simpler cases there is actually enough information to close the gaps between some contigs *in silico*. Of course, this approach does not help if pieces of the sequence are completely missing after sequencing.

- In the case that mate pairs of the reads are available, some of them can be used to find a layout of the contigs if they span the gap of two contigs. Depending on the gap size as well as the distance of the paired reads these local connec-

tions help to find appropriate contig adjacencies. Unfortunately, sequencing with mate pairs is usually more expensive such that it is often omitted.

- After an initial assembly of the reads, sometimes *fosmid libraries* are employed to accomplish the finishing of a genome. Fragments of the genome with a size between 35 and 40 kbp are used as inserts for the fosmids in order to sequence the ends of each inserted fragment. If those end sequences can be mapped to different contigs, it is possible to infer the distance and orientation of contigs towards each other. Fosmid libraries have the additional advantage that they can be used for primer walking even if the gaps between the contigs exceed the usual size to do primer walking on the genome. But of course this advantage is paid for with a high amount of work to create the library.

- A method to order contigs [56] that works in the global context of a genome is optical restriction mapping [82]. Here, just like in the sequencing approaches, a library of overlapping fragments is created. The fragments are fixed to a glass slide, digested by a restriction enzyme and then fluorescently labeled. The order of the restricted pieces is maintained during this process and their size can be estimated with fluorescent microscopy. Similar to the assembly phase in shotgun sequencing, a map of overlapping restriction sizes can be constructed for the whole genomic sequence. This so-called *optical restriction map* is specific to the employed restriction enzyme that cuts at definite locations. To order contigs, an *in silico* generated restriction map of the contigs can be compared to the optical restriction map. This is especially useful for longer contigs whereas very small contigs might be smaller than the average restriction pieces thus not allowing an ordering of them.

- More and more genomes have already been finished and their sequences are mostly available to the public. If the genome of a related species is at hand, it can be used as a reference to layout the contigs. To this end, the contigs or parts of them are matched onto the related reference allowing to estimate which of the contigs are adjacent. When doing so, one has to bear in mind that the related sequence might have undergone larger rearrangement events like insertions or deletions or even inversions of parts of the genome. Recently, the related reference based ordering has been applied using a single [79] and also multiple [109] references.

In this thesis, we build on the idea of reference based contig layouting. Besides presenting a simple method in Chapter 3 that uses a single reference genome, we also establish a formal way to collect contig adjacency information from multiple references in Chapter 4. Additionally, we show how to integrate the phylogeny of the species into our approach and provide an algorithm to treat repetitive contigs in a special way. However, it is in general advisable to combine different sources of information to find a most likely layout.

Gap Closure

The next step in genome finishing is the closure of the gaps between contigs. As already indicated, this can partially be done *in silico* but mostly this requires additional lab work. If two contigs are known to be adjacent it is possible to amplify the DNA in between the contigs with PCR by designing specific primer pairs that enclose the gap. The pieces of DNA that are copied this way can then be sequenced. Often, the gap is larger than the read length of the applied sequencing procedure. Then, a technique called *primer walking* can be applied. Here, primer pairs are designed for the part of the gap that has been sequenced in the last round and the process is iterated thus 'walking' in steps of the sequencing read length from primer pair to primer pair.

While the contig ordering can be done in little time using computational approaches, the laborious work of gap closure is the main bottleneck with respect to time and money. Therefore, it is especially important that the layout which is estimated in the computational approaches is reliable such that the number of necessary PCR reactions and sequencing steps is reduced.

Polishing

Together with the former steps of contig ordering and gap closing, it is possible to check the consistency and correctness of the current assembly. A special tool for this task is *BACCardI* [9] that uses BAC libraries to validate an assembly. Additionally, mate pairs or matches to related reference genomes might reveal misassemblies where a contiguous sequence from the assembler is in fact not contiguous. In the lab, PCR reactions can amplify regions which had a too low coverage to be assembled in the first place. Besides automatic suggestions, it is often advisable to manually curate an assembly. An established tool that helps in this task is *Consed* [35]. It allows to look at an alignment of the reads that were used to build the contig thus allowing to assess how many reads support single bases and whether there are contigs where repetitive parts of two origins are aligned. In case of mistakes in the assembly, it is possible to modify it by splitting single contigs or joining others.

Chapter 3

Efficient Matching of Contigs

Initial sequencing of a genome usually results in a set of contigs for which their relative order and orientation is not known. To close the gaps it is necessary to know which of the contigs are adjacent. This information is given in a layout of the contigs, and there are several techniques to estimate a suitable layout. Some of them are based on information provided by the sequencing run, others rely on external information. The methods described in this thesis use the genomic sequences of related species to find a layout for the contigs. Our approach is based on finding corresponding regions of contigs and reference sequences, where "corresponding" means in an ideal case that these regions share an evolutionary history and were probably derived from a common ancestor. However, it is not possible to elucidate the true evolutionary history of the DNA molecules. A common circumvention is to use sequence comparison methods to find similar regions between the reference genomes and the contigs.

The general term *sequence comparison* subsumes many different questions and also applications that deal with computing distances or finding similarities between sequences. While we want to give a short overview here, a more detailed introduction to sequence comparison, especially in bioinformatics, can be found in a review by Batzoglou [10].

Our brief historical overview starts in 1950 with the *Hamming distance* that simply counts the differing letters of two equally sized sequences [37]. In 1966, this concept was extended to the *Levenshtein*, or *edit distance* which is defined as the minimum number of *edit operations* – insertions, deletions, and substitutions of letters – transforming one sequence into another [57].

Equivalently to a minimal distance, a maximal *score* can be calculated [84] that penalizes substitutions and rewards matching characters. A first practical algorithm for scoring a pair of sequences was published in 1970 by Needleman and Wunsch [68]. Although their dynamic programming approach was designed to compare protein sequences, it can be applied to sequences in general.

One important aspect of the underlying dynamic programming datastructure is that it allows to derive an *alignment* of the characters that is optimal with respect to the applied scores. Thus it is not only possible to tell that two protein sequences are similar, but also which of the amino acids match and where insertions or deletions could have occurred.

The Needleman-Wunsch algorithm produces *global alignments* that consider *all* characters of the respective sequences. In molecular biology, however, sometimes substrings of high similarity are of interest. These *local alignments* can be found with the Smith-Waterman algorithm that was published in 1981 [87]. Applied to two sequences of length n and m respectively, the algorithms have a time and space requirement of $O(nm)$ [36].

Even though this running time is efficient, it became unpractical in combination with the growing biological sequence databases and the need to find *approximate occurrences* of protein or DNA sequences in these databases. To face this, two kinds of heuristics have been developed: *Seed and extend heuristics* that are typically very fast but might miss relevant matches, and *search space filters* which are runtime heuristics that usually guarantee to find all matches below a specified error rate. Although the latter are extremely fast in the expected case, their worst case complexity stays $O(nm)$.

In this chapter, we will first review different algorithms to find local similarities between DNA sequences. The approach that was chosen to be used for matching contigs is then explained in more detail in Section 3.2. Towards the end of this chapter, we will present our software *r2cat*, the related reference contig arrangement tool, that uses this matching technique to visualize similarities between the contigs and a reference genome and that is able to arrange the contigs with respect to these matches. A more sophisticated approach to find a layout of the contigs based on the information provided by several references will later be given in Chapter 4.

3.1 Finding Local Similarities

Similarities to a reference genome can help to find a layout for a set of contigs. In the following, we will refer to similar regions between contig and reference genome as *matches*, and the process of finding these will be termed *contig matching*.

Please note that usually in computer science a match refers to an exact and complete occurrence of a search pattern in a text. Here, we allow approximate occurrences to account for small scale evolutionary events like insertions, deletions or single nucleotide polymorphisms. Also, we do not demand that the contigs are matched completely on the reference since large scale events like translocations or inversions might have scrambled the sequences. A match is hence equivalent to a local alignment with a high score or, in other words, to a very similar region shared by contig and reference genome.

In biological sequence comparison, the terminology is often to search a "*query* in a *target*" or "*database*", instead of a "*search pattern* in a *text*" as in computer science. In the following, the contigs are the queries that are to be matched onto a target reference genome. Regardless of the different names, all sequences are composed of letters from a given finite and ordered alphabet Σ. We denote by Σ^* the set of all finite strings over Σ, by $|s| := \ell$ the *length* of string $s = s_1 \dots s_\ell$, and by $s[i,j] := s_i \dots s_j$ with $1 \le i \le j \le \ell$ the *substring* of s that starts at position i and ends at position j. For two sequences s and t, a match is a tuple $m = \big((i,j),(k,l)\big)$, if the substrings $s[i,j]$ and $t[k,l]$ have high similarity. An *alignment* is a string over the alignment alphabet $\mathcal{A}(\Sigma) := \left(\Sigma \cup \{-\}\right)^2 \setminus \binom{-}{-}$ where '–' $\notin \Sigma$ is a gap character. Each letter of $\mathcal{A}(\Sigma)$ implies an edit operation: $\binom{-}{a}$ and $\binom{a}{-}$ stand for the insertion or deletion of a character $a \in \Sigma$, and $\binom{a}{b}$ is a substitution that is also called *match* if $a = b$ and *mismatch* otherwise.

3.1.1 Smith-Waterman

The Smith-Waterman algorithm [87] is a well known method to search local alignments. It is a clever variation of the dynamic programming idea that was proposed by Needleman and Wunsch [68] for global alignments. The key datastructure is a *score matrix* S that is used to tabulate the optimal local alignment scores for all pairs of possible prefixes of two given sequences x and y. The entry $S(i,j)$ stores the optimal local alignment score for the prefixes $x[1,i]$ and $y[1,j]$. After computing the complete matrix S, its entry with the highest value reveals the best local alignment. The computation itself is based on the following recurrence:

$$
\begin{aligned}
S(0,0) &:= 0 \\
S(i,0) &:= 0 \\
S(0,j) &:= 0 \\
S(i,j) &:= \max \begin{cases} 0 & (\star) \\ S(i-1,j-1) + score(x[i], y[j]) & (\searrow) \\ S(i-1,j) - \gamma & (\downarrow) \\ S(i,j-1) - \gamma & (\rightarrow) \end{cases} \\
&\text{for all } i,\ 1 \le i \le |x|, \\
&\quad \text{and } j,\ 1 \le j \le |y|
\end{aligned}
$$

where γ is the cost for insertions or deletions of single characters, and $score(a,b)$ is a scoring function for the similarity of the characters a and b. To be sensible, the scores should be chosen such that random sequences on average result in negative global alignment scores.

The three top definitions initialize the first row and column of the matrix and are the base cases for the recursion. The remaining cells of the matrix are usually

computed in a row- or column-wise fashion where each entry $S(i,j)$ is calculated by maximizing over four different cases:

($\star$) Including zero in the maximization prevents a cumulation of negative scores. This allows to skip bad scoring prefixes and start a new alignment from the current sequence indices.

($\searrow$) This case refers to the substitution of $x[i]$ by $y[j]$. If the characters match, this will be rewarded by a positive score. Although mismatches are punished by a negative score, they can also give the maximal value if $S(i-1,j-1)$ contains a high score value.

($\downarrow$) A maximal value in this case indicates that the deletion of $x[i]$ provides the best local alignment up to the current prefixes.

($\rightarrow$) This is similar to the previous case, except for indicating that the insertion of $y[j]$ is the best choice.

Figure 3.1 shows an example of a completed score matrix. The arrows in this picture correspond to those of the above cases providing the maximal value for a cell. By following these arrows backwards, from the highest entry in the matrix to a zero entry, an optimal local alignment can be retrieved. Let $a = x[i]$, $b = y[j]$, and the arrow end in (i,j). Then, a diagonal arrow corresponds to the match $\binom{a}{a}$, if $a = b$, otherwise to a mismatch $\binom{a}{b}$. A horizontal arrow stands for the insertion $\binom{-}{b}$ and a vertical one for the deletion $\binom{a}{-}$.

The described algorithm only finds a best local alignment of two sequences. In our case, we are also interested in other similar regions, for example if parts of the contig occur rearranged in the reference. All local alignments that have a score greater than

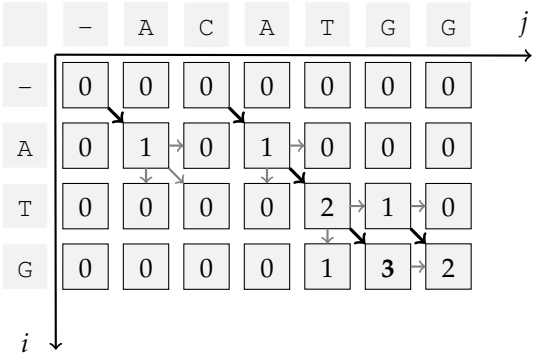

Figure 3.1: Local alignment score matrix for the sequences $x = \texttt{ATG}$ and $y = \texttt{ACATGG}$. Matching characters receive a score of $+1$, while mismatches, insertions and deletions are 'punished' by -1. The arrows indicate score maximizing edit operations where matches are black and all others are gray.

a certain threshold can easily be found by backtracing all appropriate entries of the score matrix. However, care has to be taken on overlapping alignments. To this end, a more advanced algorithm that finds non-overlapping suboptimal alignments was proposed by Waterman and Eggert [101].

The calculation of the score matrix needs $O(|x| \cdot |y|)$ time since the maximum value in each cell can be calculated in constant time. The extraction of the optimal local alignment then needs time proportional to the length of the alignment. Especially, the matrix computation becomes a computational bottleneck if many smaller sequences need to be searched in a larger database. To alleviate the time demand in such cases, heuristics can be applied instead of the exact Smith-Waterman algorithm. The next section introduces the concept of seed and extend heuristics, which trade speed for sensitivity. This means that they might miss some true matches found by the Smith-Waterman algorithm but are usually much faster.

3.1.2 Seed and Extend Heuristics

As the name already suggests, seed and extend heuristics are performed in two phases: First *seeds* are searched which are highly similar regions of query and target sequence. In the second phase, the seeds are *extended* to a local alignment.

In order to efficiently retrieve high scoring seeds, different techniques can be used that in general build an index of one of the sequences. Indexing techniques are called *online* if only the query is indexed. If it is also possible to preprocess the target sequence, they are referred to as *indexed*.

A seed is a pair of substrings of query and target sequence that have high similarity. In the local alignment example in Figure 3.1, a seed corresponds to diagonals of the score matrix with many increasing entries, indicated by the black arrows. In the literature, different kinds of seeds are considered which require to some extent different indexing methods:

- *Exact seeds* refer to an exact occurrence of a part of the query in the target sequence. Usually a minimum length is demanded to avoid unspecific matches. A hash index of fixed size substrings can be used to efficiently lookup matches, as for example applied in *BLAT* [51] to find the seeds. For variable length seeds, suffix arrays or suffix trees can be used, as for instance in *MUMmer* [54]. A space efficient alternative of a suffix tree can be realized using the Burrows-Wheeler transformation, like in *BWA* [58].

- *Spaced seeds*, or *gapped seeds*, are like exact seeds of a fixed size, however with the difference that not all character positions have to match. These seeds are usually longer than exact seeds and contain *don't care* characters that are ignored when comparing query and target sequence. Similarly to exact seeds, spaced seeds can be collected in a hash based index although it needs to be

slightly modified. Algorithms using one or two spaced seeds are implemented in *PatternHunter I* and *II* [61, 59].

- *Approximate seeds* allow that the corresponding sequences are different in any position, although the sequences should have a high similarity score. Again, a modified hash-index can be used that stores for each substring the positions of all strings of the same size having a score above a given threshold t. The choice of t allows a trade-off between speed and sensitivity. If t is high, the index contains fewer entries resulting in fewer seeds found. On the one hand, this makes the extension phase faster, on the other hand, important seeds might be missing such that a match is not found. A low threshold t, on the contrary, will result in many seeds found that have weaker similarities. This causes the extension phase to take more time, although additional matches can be found. When using approximate seeds, mostly the query is indexed since the index would grow too big if t is low and the target is large. Implementations that use approximate seeds are for example *FASTA* [72] and *BLAST* [3, 2].

After a set of seeds has been found, the seeds are postprocessed in the extension phase. Overlapping or nearby seeds are usually merged and subsequently aligned with a Smith-Waterman like alignment procedure.

The whole seed and extend procedure is based on the assumption that each interesting local alignment must contain at least one high scoring seed. However, it is possible that an optimal local alignment computed with Smith-Waterman does not contain an appropriate seed such that heuristics of this kind will miss it. The sensitivity is lower than a Smith-Waterman alignment that is sometimes referred to as full-sensitivity alignment.

One of the most prominent tools for approximate matching of nucleotide or protein sequences is *BLAST*. It is an online algorithm that preprocesses the query. After that, the expected running time for searching a preprocessed query is proportional to the length of the target.

When matching many smaller contigs onto a large reference genome, it can be advantageous to create an index of the reference, since then the expected search time for each contig will only be proportional to the contig's length. The filter algorithms described in the next section commonly rely on such an indexing of the larger target sequence.

3.1.3 Search Space Filtering

To find local similarities between two sequences, a Smith-Waterman score matrix like in Figure 3.1 can be computed. This matrix corresponds to the search space of all possible local alignments between the two sequences. While in this picture the seed and extend approaches strive to find high scoring diagonals, *search space filtering* does in principle the opposite by discarding regions of the matrix that are

too dissimilar. Search space filtering is also known as an *exclusion method* for approximate string matching [36, Ch. 12.3].

Although the matrix is not necessarily explicitly created for filtering the search space, the conceptual idea partitions it into equally sized regions which are possibly overlapping. The goal of a filter algorithm is then to discard as many regions as possible that do not fulfill a certain *filter criterion*. This criterion is usually a necessary, although not sufficient, condition that a region yields a score higher than a given threshold t, or similarly that it has less than e mismatches. The filter is called *lossless* if all such regions are kept and none is falsely discarded.

All remaining parts of the matrix, which were not filtered out, have to be verified in a postprocessing phase that is similar to the extension phase described in the previous section. In contrast to the seed and extend approaches, filtering with a lossless filter will find all regions having a score higher than the specified threshold.

After the index is built, the running-time of exclusion methods for searching is in the expected case typically linear or even sub-linear with respect to the length of the query. In the worst case, however, the complete score matrix has to be verified thus resulting in the same time complexity as the Smith-Waterman algorithm.

3.2 Matching by Filtering with q-Grams

For matching the contigs we chose to use the *SWIFT*[1] algorithm, a lossless search space filter that was developed in Bielefeld [76]. The filter guarantees to find all regions of a specified minimum size that have less than a given error rate. The algorithm is very fast when matching many contigs on a reference genome due to an indexing of the reference sequence. Most information presented in this section is based on the work of Rasmussen *et al.* [76].

3.2.1 General Idea

Filtering methods to find approximate matches usually follow a so-called *q-gram counting strategy*. A *q-gram* is simply a string $s \in \Sigma^q$ of fixed size q; the literature uses frequently *l-tuple* or *k-mer* synonymously. Counting the q-grams that are shared between sequences allows to determine whether an alignment with less than e errors is possible. The so called *q-gram lemma* [71, 98] states that two sequences of length ℓ with Hamming distance e share at least $T(\ell, q, e) := \ell + 1 - q \cdot (e + 1)$ q-grams. If the sequences are completely identical, they share all overlapping $(\ell - q + 1)$ q-grams. With each mismatch, up to q matching q-grams are destroyed. The lemma also holds for the edit distance, if ℓ is the length of the shorter sequence in an alignment.

The q-gram lemma can be used to efficiently discard many dissimilar regions of query sequence and target. In a naive approach that we call the *basic algorithm*, both

[1] The acronym stands for: Sequence Searching and Alignment with Indexing and Filtering

sequences are partitioned into overlapping parts of equal size. This is equivalent to partition an implicit score matrix, like the one in Figure 3.1 on page 26, into overlapping boxes. For each box, the q-grams can be counted that are shared by both sequences. We call these shared q-grams in the following *q-hits*. If the q-hit count of a box is higher than the threshold defined by the q-gram lemma, then this is a necessary condition that an approximate match with less than the specified number of mismatches exists. If the count is below that threshold, there is definitely no such match.

The *SWIFT* algorithm uses further observations to partition the score matrix more cleverly than the above basic algorithm. Additionally, it provides a sophisticated calculation of the partition dimensions and the needed q-hit threshold. Before we explain details of the *SWIFT* filter in Section 3.2.3, we will first address how the q-hits can be found efficiently using an index datastructure.

3.2.2 Building an Index of the Reference Genome

In the above basic algorithmic idea it is crucial to find exact matches of q-grams, the q-hits, within the boxes very quickly. This can be achieved with a so called *q-gram index* which is a simple hash-based index of all q-grams and their positions in the indexed sequence.

In order to access the q-grams efficiently in memory, a hash function maps each possible q-gram to a natural number in the range $\left[0,\, |\Sigma|^q - 1\right]$. A hash number of a q-gram $s \in \Sigma^q$ can be calculated, for example, with the following ranking function:

$$r(s) = \sum_{i=1}^{q} r_\Sigma\big(s[i]\big) \cdot |\Sigma|^{i-1} \tag{3.1}$$

where $r_\Sigma(c)$ maps each character $c \in \Sigma$ to an integer in $\{0, \ldots, |\Sigma| - 1\}$.

With the help of this hash function, an implementation of a q-gram index can be realized with two tables. The *occurrence table* OCC that contains the concatenated lists of occurrences for all q-grams and the second *offset lookup table* OFF which contains for each q-gram the offset where its list of occurrences begins in the first table. The table OCC is of size $O(n)$, and OFF is of size $O(|\Sigma|^q)$. With both tables in memory, all occurrences of a q-gram s can be retrieved quickly by looking at the offsets $\mathsf{OFF}[r(s)]$ to $\mathsf{OFF}[r(s)+1]-1$ in table OCC.

The q-gram index can be built by going two times through the sequence to be indexed. In the first run, each occurring q-gram is counted. Based on the counts, we can specify the offsets in OFF, and add each occurrence of a q-gram during the second run to the appropriate location in OCC.

When computing the hash number of the overlapping q-grams in both above runs with ranking function (3.1), we can efficiently retrieve the rank of a next q-gram that overlaps at $q-1$ positions: If for two overlapping q-grams az and zb with $a, b \in \Sigma$

and $z \in \Sigma^{q-1}$ the rank $r(az)$ is known, then it can be updated in constant time to $r(zb)$ using the following equation:

$$r(zb) = \left\lfloor \frac{r(az)}{|\Sigma|} \right\rfloor + r_\Sigma(b) \cdot |\Sigma|^{q-1}$$

When using this constant-time update, the construction of a q-gram index takes $O(n + |\Sigma|^q)$ time for a sequence of length n.

There are a few points worth mentioning for a practical application of a q-gram index. First of all, indexing large sequences can lead to memory problems. The concatenated list of occurrences OCC needs $(n-q+1)$ times the size of an integer value. Depending on the architecture, the complete list needs about four times the size of the sequence to be indexed. The main memory of current desktop computers might thus be a limiting factor for indexing larger eukaryotic genomes. As a workaround, the index can be built for partitions of the sequence, or may be written to disk. Both will be more time consuming when querying for q-grams.

Besides the memory restriction for the occurrence table OCC, also the use of an integer to point to a location in the original sequence is limited. The programming language Java, for example, allows a maximum value of $2^{31}-1 \approx 2.1 \cdot 10^9$ for an integer. This inhibits to index the human genome that has over $3 \cdot 10^9$ bases.

A second practical issue is a proper choice of q. On the one hand, a high value of q lets the highest hash number for a q-gram ($|\Sigma|^q - 1$) get bigger than the maximal integer value. Then it is not possible to access the OFF array properly. Since also the size of the alphabet Σ matters, this kind of index is better suited for small alphabets like for DNA sequences. If, on the other hand, q is too small, then co-occurring q-grams might not be very specific and are merely observed by coincidence. The expected number of occurrences of a q-gram in a uniformly i.i.d. random sequence of length ℓ is $(\ell - q + 1)/|\Sigma|^q$. If we consider DNA sequences with $|\Sigma| = 4$ and an exemplary size of $\ell = 6 \cdot 10^6$ bases for prokaryotic genomes, a value of $q = 11$ results in each q-gram occurring about once in the expected case.

3.2.3 Filtering for Similarities

Unlike the basic filtering algorithm of Section 3.2.1 that partitions the search space into blocks, the *SWIFT* algorithm partitions the score matrix diagonally into parallelograms. This reflects more appropriately the shape of an area in which high scoring matches appear.

In the simple case of finding exact matches, these parallelograms would have a width of one diagonal and all overlapping q-grams would have to match. For approximate matches, we consider *epsilon-matches* which are basically optimal local alignments where the involved query substring has a length of at least n_0 and the alignment has a maximal *error rate* of ϵ. The latter means that the alignment contains

less than $\lfloor \epsilon \cdot n_0 \rfloor$ insertions, deletions, or mismatches. Rasmussen *et al.* provide a lemma for the dimensions of a parallelogram containing such an ϵ-match:

Lemma 1 (Rasmussen *et al.* [76]) *Let x be a substring of the query of length n_0 or greater that has an ϵ-match to a substring y of the target. Let $U(n, q, \epsilon) := (n+1) - q \cdot (\lfloor \epsilon n \rfloor + 1)$, and assume that the q-gram size q and the threshold τ have been chosen such that*

$$q < \lceil 1/\epsilon \rceil \quad \textit{and} \quad \tau \leq min\{U(n_0, q, \epsilon),\, U(n_1, q, \epsilon)\},$$

where $n_1 = \lceil (\lfloor \epsilon n \rfloor + 1)/\epsilon \rceil$. Then, there is guaranteed to exist a $w \times e$ parallelogram containing at least τ q-hits in x and y, where

$$w = (\tau - 1) + q \cdot (e + 1) \quad \textit{and} \quad e = \left\lfloor \frac{2(\tau - 1) + (q - 1)}{1/\epsilon - q} \right\rfloor .$$

For the choice of n_0 the authors give the corollary that a $w \times e$ parallelogram according to Lemma 1 exists if $n_0 \geq q \left\lceil \frac{\tau+q-1}{1/\epsilon-q} \right\rceil$. Rasmussen *et al.* give the corresponding proofs, together with an efficient algorithm to find all $w \times e$ parallelograms for ϵ-matches of length n_0 or greater. The outline of the algorithm is as follows:

With the dimensions for the parallelograms given, the search of appropriate regions uses a sliding window approach. Therefore the score matrix is separated into overlapping bins, each containing $e+1$ diagonals. Then a window of size w is slid over the query sequence. The intersection of the window and the diagonals of the bins define parallelogram regions for which all q-hits are counted that occur within them. It suffices to add the entering and substract the leaving q-hits of each parallelogram. The positions of query q-grams on the target are retrieved with the q-gram index of Section 3.2.2. If a parallelogram counter reaches τ q-hits, then the parallelogram is reported to be postprocessed. Instead of reporting overlapping parallelograms, they are merged on the fly.

To improve the space requirements of this algorithm, the number of counters needed can be reduced [76, Section 3.2.1]. To this end, $w \times (e+\Delta)$ parallelograms are considered that overlap by e diagonals. Figure 3.2 (a) shows the concept of the $w \times (e+\Delta)$ parallelograms. If Δ is a power of two and greater than e, then the bin indices can be efficiently computed with fast bit shift operations.

The running time of this algorithm can be improved by processing each q-gram of the query only once [76, Section 3.2.2]. Usually, it is processed one time when entering and a second time when leaving the window. The improved version, as illustrated in Figure 3.2 (b), remembers for each parallelogram, besides the q-hit counter, also the minimum and maximum starting position of a q-hit with respect to the query. This allows that only entering q-hits at the current position need to be considered. If a q-hit is more than $w-q$ away from the previous one, then an ϵ-match cannot contain both q-hits. Hence, the maximal parallelogram without this new q-hit, defined by minimum and maximum position and of width e (or

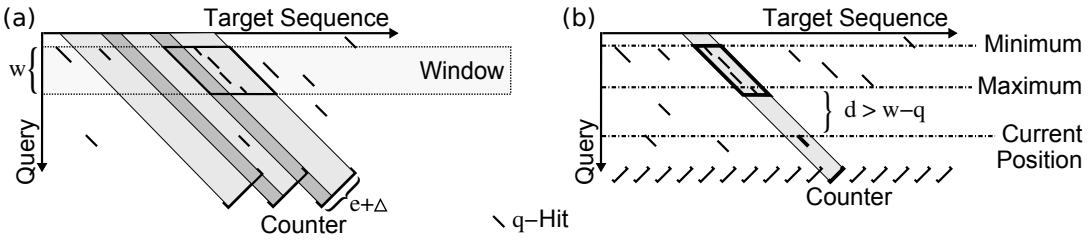

Figure 3.2: Improvements to the SWIFT filter algorithm: (a) Space reduction for the counters due to the use of $w \times (e+\Delta)$ parallelograms, and (b) speed improvement by processing each q-gram of the query only once instead of twice. Figures adapted from unpublished work of K. Rasmussen.

$e+\Delta$ respectively) is reported, if it contains at least τ q-hits. In any case, the q-hit counter is reset and the minimum and maximum positions are set to the current position on the query. When the end of the query is reached, all parallelograms are checked and reported if they contain enough q-hits. The pseudocode of the complete algorithm as well as further considerations are discussed in detail in the aforementioned publication [76].

The next section introduces our implementation of the *SWIFT* filter algorithm that provides the matches to visualize and arrange the contigs.

3.3 *r2cat* – The Related Reference Contig Arrangement Tool

If reference genomes are available, it has to be assessed whether they are related enough to provide information for contig layouting and thus might help in the gap closing phase. A visual inspection of corresponding regions between contigs and reference genome gives first clues in this regard.

We implemented the program *r2cat* (related reference contig arrangement tool) that is able to quickly match a set of contigs onto a related genome and display their similarities in an interactive visualization. The speed of our matching routine is competitive to other established programs, and an automated contig arrangement can be performed that is helpful in the finishing phase of a sequencing project by giving valuable hints on layout of the contigs.

In several sequencing projects, *r2cat* was already successfully applied to help in the finishing process: *Corynebacterium pseudotuberculosis* FRC41 [95], *Rhizobium lupini* (now *Agrobacterium* sp. H13.3) [105], and *Corynebacterium ulcerans* [93]. Additionally many in-house users at the CeBiTec appreciate the capability to quickly generate synteny plots for prokaryotic genomes with *r2cat*.

In the following we explain the steps of matching, visualizing, and arranging contigs with *r2cat* in more detail.

3.3.1 Matching

To assess their relatedness, similar regions between the contigs and a related reference genome have to be determined. We implemented for this task the q-gram filter algorithm described in Section 3.2.

As motivated in Section 3.2.2, we use a default size of $q = 11$ for the q-grams. To calculate the parallelogram dimensions of the filter, we chose a minimum length of $n_0 = 450$ bases. This choice is based on the observation that smaller contigs are often discarded since these do not provide much information and might even be a result of a sequencing error. Furthermore, we set the maximum error rate ϵ to 8% in order to allow mutations to have happened between related sequences, but at the same time keep the filter algorithm fast.

According to Lemma 1, the chosen values yield the threshold $\tau = 44$ q-hits that have to occur in a region of width $e = 64$ bases and height $w = 758$ bases. While filtering we consider $w \times (e+\Delta)$ parallelograms with $\Delta = 128$ as proposed in Section 3.2.3 to reduce space requirements.

Our matching routine is capable of handling multi-chromosomal genomes, provided in multiple FASTA files, and also finds matches for the reverse complement of each contig. To perform the matching, first a q-gram index is created for the reference genome. Only q-grams consisting of the letters `A`, `C`, `G`, and `T` are considered, not distinguishing between lower- and upper-case letters. If the sequences contain other characters, for example of the IUPAC nucleotide ambiguity code, these characters are ignored and q-grams containing them are not indexed.

After building the index, each contig is processed in forward and reverse complementary direction to find all parallelograms fulfilling the filter criterion. The reverse complement matching is necessary since the orientation of the contigs with respect to the reference genome is usually not known, and additionally the genomes might have been shuffled by inversion events during evolution.

Normally, all found parallelograms are aligned with a Smith-Waterman like algorithm to verify whether they are true matches, and to find the local alignments of the sequences. In our implementation, we omit this time consuming step and consider the parallelograms as matches. A disadvantage of this is, that our matches might not start at the exact position of a true match on the reference. The positions may differ by up to Δ bases because of the overlapping $w \times (e+\Delta)$ parallelograms used by the implemented memory improvement. Additionally, it might happen that parallelograms are shorter than 450 bases and also do not strictly fulfill the filter criterion due to the speed improvement mentioned in Section 3.2.3 that we also implemented. However, Rasmussen *et al.* remark for the latter case "that when searching for local alignments in biological sequences the regions triggering such parallelograms are often of great interest anyway" [76]. That is why we keep them.

Despite of these objections, the matches of our filter provide an acceptably accurate impression of the relatedness of two DNA sequences, as Figure 3.3 demon-

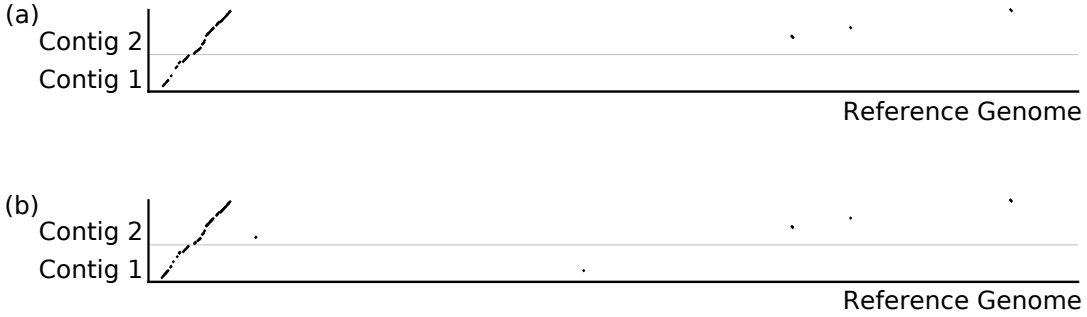

Figure 3.3: Two *Corynebacterium urealyticum* contigs matched on the reference genome *Corynebacterium jeikeium*: (a) with the q-gram filter implemented within *r2cat*, and (b) generated with *BLAST*.

strates. For this figure, we arbitrarily picked two large contigs and matched them onto a related reference genome: first with our matching routine and the second time using *BLAST* [2]. Both plots, which visualize the matches found by the programs, agree largely. Sometimes, *BLAST* finds small matches that our filter misses. On the other hand, our filter frequently shows a single combined match, where *BLAST* has several smaller. In general, we believe that the quality of the matches is acceptable for the purpose of displaying syntenies and also for layouting contigs.

All matches found can be cached in human readable text files for which we chose *.r2c as extension. In these files, also the order of the contigs is stored, if, for example, the automatic arrangement of Section 3.3.3 has been used. Besides loading and saving matches in *r2cat*, it is also possible to parse and modify the files with other programs, or to edit them manually.

To show that the matching implemented in *r2cat* is competitive in running time, we compared it to the three well known matching programs *BLAST* [2], *BLAT* [51] and *MUMmer* [54]. For *BLAST*, we used two different versions: The program *blastn* was completely reprogrammed in C++ and seems to have become faster in some cases than its predecessor *blastall* implemented in C. We changed the output format of both *BLAST* versions from 'pairwise alignment' to the less extensive 'tabular output' that is likewise produced by the other programs. This slightly improved the running time of the *BLAST* programs.

Each program was used on two prokaryotic datasets to match a set of contigs onto a reference genome: The first, small dataset of *Streptococcus suis*, taken from [7], consists of 281 contigs with a total size of 2.1 Mbp that were matched on the genome of another strain of this species. Details of the contigs are listed in Table 3.1, details of the reference genome are given in Table 3.2. The second, larger dataset of *Sinorhizobium meliloti* has 446 contigs and was matched on another strain with a total size of 6.7 Mbp, given in three replicons. Details are given in the tables.

Table 3.1: Contig data that was used to test the matching. Sizes are given in base pairs (bp). The N50 contig size is defined as the size of the largest contig such that at least half of the total size is covered by contigs larger than that contig.

Contig Organism	#Contigs	Total Size	N50 Contig Size	Median Size
Sinorhizobium meliloti SM11	446	7,187,910	34,216	7742
Streptococcus suis	281	2,071,601	32,959	121

Table 3.2: The reference genomes to test the matching. Sizes are given in base pairs (bp).

Reference Genome	Replicon Type	Size	NCBI Number
Sinorhizobium meliloti 1021	chromosome	3,654,135	NC_003047
Sinorhizobium meliloti 1021	megaplasmid pSymB	1,683,333	NC_003078
Sinorhizobium meliloti 1021	megaplasmid pSymA	1,354,226	NC_003037
Streptococcus suis SC84	chromosome	2,095,898	NC_012924

The results for each program and dataset can be found in Table 3.3. It shows the time that was needed for matching and additionally the number of contigs that could not be matched and thus can neither be visualized nor arranged.

Most programs do not give matches for all contigs except for *blastall* that outputs matches as short as 12 bases. It is arguable if those matches can help to find a layout for the contigs since these matches might occur only by chance.

The competing programs find real alignments and thus have a harder task than *r2cat* that only finds parallelograms which could contain a match. However, when looking at Figure 3.3, the reduced work seems to be adequate for a quick impression on the synteny of the sequences. Using our own matching routine has the advantage that *r2cat* has no dependencies on other programs which might be difficult to install and maintain. To use exact matches nevertheless, we wrote an additional *Perl* script that is able to reformat the tab-separated output of *BLAST* in order to create a *.r2c file that can be opened with *r2cat*. In principle, matches of any other program can be used as long as they are provided in a similar tab format.

The matching was primarily implemented to be used on desktop computers and with prokaryotic genomes in mind that typically are smaller than 12 Mbp. On computers with sufficient memory, the matching is also possible for smaller eukaryotic genomes. To demonstrate this, we matched a scaffold of the *Drosophila ananassae* genome (*Dana*, revision 1.3, 13749 sequences with a total size of 231 Mbp) onto the genome of *Drosophila melanogaster* (*Dmel*, revision 5.35, 15 chromosomes with a total size of 169 Mbp). Both genomes were acquired from the websites of the FlyBase project [97] in multiple FASTA format. The scaffold of *Dana* seems to be in draft status and contains many small sequences. In contrast, *Dmel* is completely finished

Table 3.3: Times for matching a set of contigs on a reference genome (average of two consecutive runs). Additionally, the number of contigs having no match at all is given. The experiments were performed on a sparcv9 processor operating at 1593 MHz.

		Streptococcus suis		*Sinorhizobium meliloti*	
Program	Version	Time (s)	Unmatched	Time (s)	Unmatched
blastall	2.2.24	12.3	0	110.9	0
blastn	2.2.24+	4.8	86	121.2	72
blat	15	44.9	94	674.2	84
nucmer	3.07	9.0	109	40.8	92
r2cat		7.7	102	39.3	75

such that it can be used as a reference genome. The matching took 6.9 hours on the same machine that was used for the results in Table 3.3. The peak memory consumption was at 3.1 GByte while creating the q-gram index. We also matched this dataset with *blastall* and surprisingly this was with 6.6 hours a few minutes faster. This might be explained by our observation that *r2cat* found five times as many matches, measured in the total length of contig substrings involved in matches.

3.3.2 Visualization

There is a variety of graphical tools to explore and analyze genomic data. For an overview, see the review of Nielsen *et al.* [69].

We use a *synteny plot* to visualize similar parallelograms of contigs and reference genome. In such a plot, the sequences are represented by the x and y-axis of a coordinate system and matching regions are displayed as diagonal lines. This visualization is related to a *dot plot* that can be derived if in a score matrix, like the one of Section 3.1.1, every score-increasing diagonal is plotted as a dot. Synteny plots usually combine several dots to diagonals and are often flipped vertically.

Figure 3.4 on the next page shows an exemplary synteny plot that was generated with *r2cat*. For this plot, the contigs of *Corynebacterium urealyticum* were matched onto the closely related genome of *Corynebacterium jeikeium* as reference genome. The horizontal axis represents the reference genome, and on the vertical axis all contigs are stacked in the order of the underlying FASTA file.

Each parallelogram that was found by the implemented matching routine of Section 3.3.1 is displayed as a diagonal line. Lines with negative slope correspond to reverse complementary matches.

The horizontal bar at the bottom of the plot helps to assess the coverage of the matches: maximum coverage is displayed in black and fades to light grey with less coverage. Uncovered regions are highlighted in red.

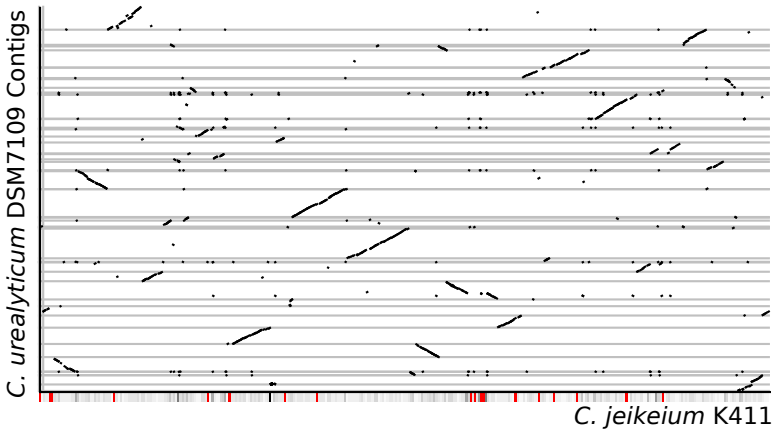

Figure 3.4: Synteny plot generated by *r2cat*. The contigs of *Corynebacterium urealyticum* are matched onto the reference sequence of *Corynebacterium jeikeium*. Details of the sequences are given in Section 6.1.1.

Our implementation features an export of the displayed synteny plots to either bitmap or vector based graphics formats. Since editable formats like scalable vector graphics (SVG) are supported, *r2cat* is well suited to produce high quality synteny plots to be used in publications and other print media.

In the program, the view area is zoomable and panable such that matches can be examined more closely. Matches can be selected in the plot and also displayed in a separate table window. Besides the start and stop positions of the matches in contigs and reference genome, also the number of exactly matching q-grams and an estimate for the repetitiveness of a match is shown. Selected matches are displayed in red and matches belonging to a contig that is selected, appear in orange inside the plot (not shown in Figure 3.4).

Initially, the contigs are stacked in the order as they appear in the FASTA file that was used for matching. There are two possibilities to change their order: Either with the automated approach that is described in Section 3.3.3, or manually in a separate window showing the contigs in a table. In addition to moving contigs per drag and drop in this table, it is also possible to reverse complement a contig if this seems appropriate. After matching, *r2cat* automatically reverse complements a contig if the majority of matches belong to the reverse complement. Contigs that have been reversed are displayed in the plot in blue color.

As an additional table column, *r2cat* shows an estimation for each contig how much of its sequence is repetitive according to the reference genome. Details of the repeat detection are given in Section 4.4.2.

While the main focus of our tool is to arrange a set of contigs, the synteny visualization can also be used to investigate the relationship between two species if,

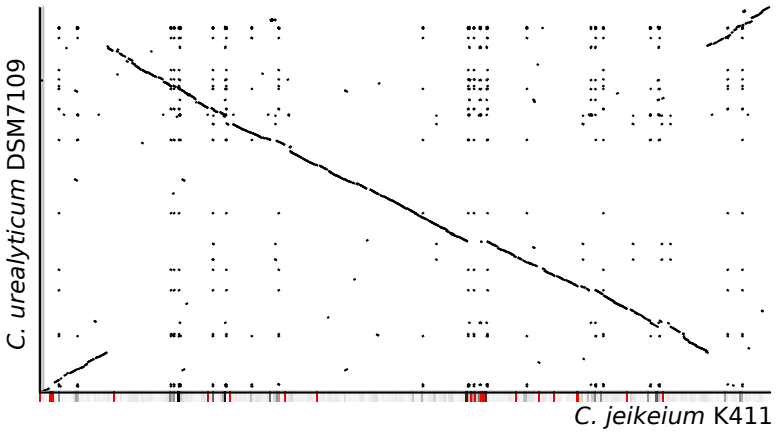

Figure 3.5: Synteny plot of two genomes. The reference is the same as in Figure 3.4, but instead of the contigs of *Corynebacterium urealyticum*, its already finished genome is used as query.

instead of the contigs, the genomic sequence of another genome is chosen for matching. Exemplary, Figure 3.5 shows such a plot where a large scale inversion and several smaller insertions and deletions can be observed. Additionally, repetitive elements can be seen that appear in regular patterns aside from the main diagonals.

3.3.3 Simple Contig Mapping

Alternatively to changing the order of the contigs manually, *r2cat* features an automatic arrangement of the contigs based on their matches to the displayed reference genome. The order of the contigs is inferred by *mapping* them onto the reference.

To perform the mapping, we consider for each contig the coverage of its matches to the reference genome: Let $\mathrm{cov}(c, p)$ be the maximum number of q-hits of a match – from contig c to the reference genome – that covers a single base position p. We use the maximum instead of a sum to avoid counting the q-hits in overlapping matches several times. To find where a contig has the best coverage, we use a sliding window approach. In a window of the size of a contig, we look at the *average window coverage* that is defined as the sum of $\mathrm{cov}(c, p)$ over the positions in that window, divided by its size. By sliding the window over the reference, we calculate the average window coverages for all possible positions. Each contig is then mapped to that window position where it yields the highest average window coverage. Figure 3.6 shows the result of applying this procedure to the matches displayed in Figure 3.4.

After an automated arrangement, the displayed order of the contigs in *r2cat* can be exported to text files where each line gives the identifier of a contig, as well as a '+' or '−' sign, indicating the inferred orientation with respect to the original

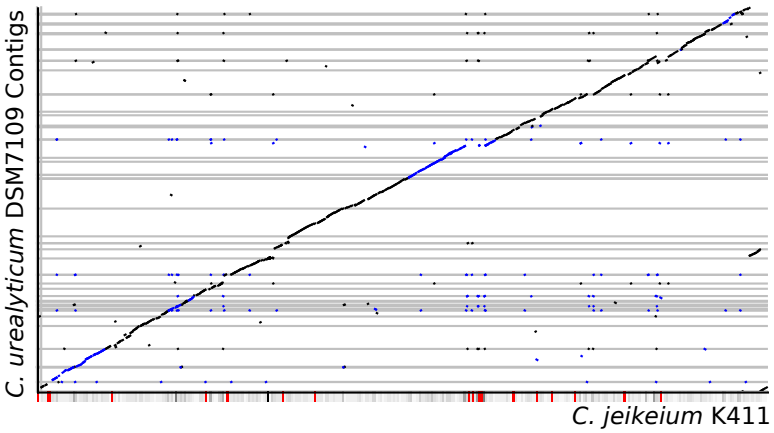

Figure 3.6: The contigs displayed in Figure 3.4 are ordered with the simple mapping approach implemented in *r2cat*. Matches in blue indicate that the contig is reverse complemented.

FASTA file. If the original FASTA file is still available, it is also possible to export the contigs, ordered and oriented, into a copy of it.

In a recent extension of *r2cat*, Yvonne Hermann – one of our bachelor students – integrated an automatic primer design step to aid the gap closing process in the finishing phase of a sequencing project. Primer pairs at the borders of putatively adjacent contigs can be used to amplify the DNA in between, and sequence it subsequently. For the primer design, a variety of relevant sequence features, like for instance the GC content, or the occurrence of homopolymers, are evaluated and scored to find suitable primer candidates. The best candidates are then paired to assure that they have comparable melting temperatures and that they bind to the target sequence instead of to each other.

The matching of contigs to a reference genome can be considered valuable for gap closing purposes, assuming that the corresponding genomes have a high degree of synteny. Still, the results of the simple mapping of this section have to be handled with care. Figure 3.7 shows the contigs of *Corynebacterium urealyticum* in their true order, matched on different reference genomes. The true order was obtained by matching the contigs onto its already finished genome. This figure shows possible causes why a simple reference based approach might not be sufficient:

1. Large scale inversions might suggest that two contigs are adjacent while in fact they are not, like in Figure 3.7 (a). Here, the breakpoints of the inversion, which are the points where the inverted sequence had been cut, coincide with the borders of the contigs and thus cannot be detected. In other cases, like the inversion in the center of Figure 3.7 (b), the breakpoints are detectable, since a single contig has matches on the forward and backward strand of the

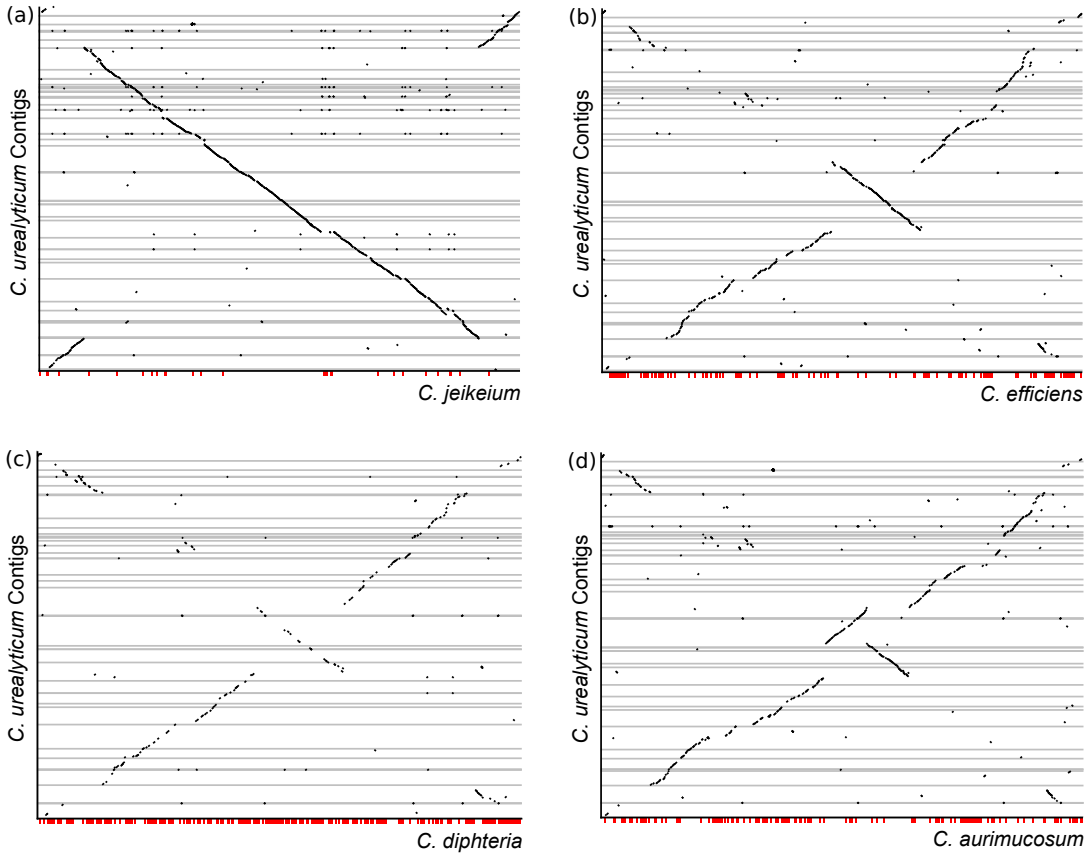

Figure 3.7: Pairwise synteny plots of the contigs of *Corynebacterium urealyticum* and four chosen complete genomes of the *Corynebacteria* genus. Details of the sequences are given in Section 6.1.1.

reference. In simple cases, this might be untangled; however, the influence of several inversions like in Figure 3.7 (d) complicates this task.

2. For a given reference, some contigs cannot be matched reasonably. This happens if the degree of synteny is low, as for example in Figure 3.7 (c). Alternatively, the sequence of a contig might have been deleted in the reference during the course of evolution, or, equivalently, its sequence could have been inserted into the contigs' genome. If a contig is not matched, it cannot be contained in a reference based layout and thus not be considered for closing the gaps between the contigs. Fortunately, if the reference is close enough, only very small contigs cannot be matched at all.

3. Repetitive contigs are mapped to one of their possible occurrences on the reference. The actual placement is merely done by chance when the highest

number of matches is considered. Figure 3.7 (a) shows the typical patterns of repetitive matches.

Some of these obstacles can be avoided, or at least alleviated, if not one but several reference genomes are considered. While the simple mapping approach described in this section is not capable of handling several references, a more sophisticated approach will be described in Chapter 4. There, we show how to incorporate several reference genomes, as well as their phylogenetic relationship, and also give an algorithm to treat repetitive contigs in a special way.

Chapter 4

Advanced Contig Layouting using Multiple Reference Genomes

> I appreciate theoretical work for its elegance, yet find it sterile when it is too detached from practical value.
>
> *(Justin Zobel, Writing for Computer Science)*

A simple mapping of contigs to a single reference genome might not be sufficient to devise a valuable layout of the contigs. Using several related genomes as references can improve the predictions of neighboring contigs for more distantly related genomes [109]. However, conflicting information may arise that complicates to find a layout in which the contigs are uniquely ordered.

In this chapter, we describe a strategy to find a layout of a set of contigs that is capable to employ several related genomes as references. Our approach uses all similarities between the contigs and the reference genomes to collect hints that a pair of contigs is adjacent. The hints are gathered in a weighted graph that is introduced in Section 4.1. The weights in this graph can help in the finishing phase of a sequencing project. To this matter, Section 4.2 describes a fast algorithm for estimating a layout of the contigs with respect to a given graph. After this, we discuss enhancements of the graph creation in Section 4.3, and possible variations of the layouting algorithm in Section 4.4.

4.1 The Contig Adjacency Graph

In this section, we provide the formal notation for the *contig adjacency graph* that we use to collect adjacency information of a set of contigs. Basically, this graph contains edges for all possible adjacencies. These are weighted in a way that the edges between potential neighbors receive high weights. The weighting is done by analyzing the matches of the contigs with respect to a set of reference genomes. Each reference genome can be used independently to compute the weights, and it is straightforward to merge the information of several references into a single graph.

In the following, we will first introduce the notation for the contig adjacency graph, then motivate the edge weighting function and finally, in Section 4.1.3, give an algorithm to construct the graph.

4.1.1 Notation

Recall the notation for strings and matches given in Section 3.1. In this chapter, let $\Sigma = \{\mathtt{A}, \mathtt{C}, \mathtt{G}, \mathtt{T}\}$ be the alphabet of nucleotides such that Σ^* is the set of all possible finite DNA sequences. Suppose we are given a set of contigs $\mathcal{C} = \{c_1, \ldots, c_n\}$, $c_i \in \Sigma^*$, and a set of already finished reference genomes $\mathcal{R} = \{g_1, \ldots, g_{|\mathcal{R}|}\}$, $g_r \in \Sigma^*$.

As already announced, we use matches of the contigs to related reference genomes to infer information about the layout of the contigs. Let $m = \big((s_b, s_e), (t_b, t_e)\big)$ be a match of contig c and reference g. This means that $s = c[s_b, s_e]$ is a substring of the contig, $t = g[t_b, t_e]$ is a substring of the reference genome, and both sequences are similar. The *length* of a match, $|m| := |t| = t_e - t_b + 1$, is defined as the length of the covered substring in the reference genome. For $s_b > s_e$ we define $c[s_b, s_e]$ to be the reverse complement of $c[s_e, s_b]$ and call m a *reverse match*. Further, we assume without loss of generality that $t_b < t_e$ for all $g[t_b, t_e]$. Otherwise we can replace both involved substrings by their reverse complements.

We generate the matches with the q-gram filter described in Section 3.3.1. Thus, for each match m, the number of exactly matching q-grams is provided which can be used as a quality estimation for that match. We refer to this number as $\mathit{qhits}(m)$. The set of matches between a contig $c_i \in \mathcal{C}$ and reference genome $g_r \in \mathcal{R}$ is in the following denoted as $\mathcal{M}_i^r = \{m_1, \ldots, m_s\}$.

Each match $m = \big((s_b, s_e), (t_b, t_e)\big) \in \mathcal{M}_i^r$ can be interpreted as a projection of contig c_i onto the reference genome g_r. The *projected contig* $\pi(m) := \big((t_b - s_b), (t_e + |c_i| - s_e)\big)$ refers to the implied pair of index positions on g_r. For reverse complement matches, the projection can be defined similarly. Note that the size of the projected contig can deviate from the real size of a contig due to insertions or deletions in the match. Figure 4.1 shows an example of two projected contigs as well as their distance, which we define next.

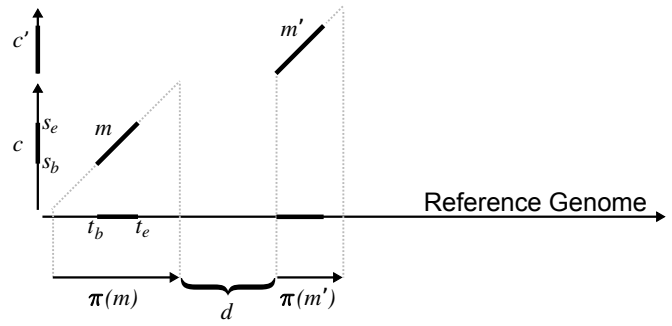

Figure 4.1: Projections $\pi(m)$ and $\pi(m')$ of the contigs c and c' based on their matches m and m'. The distance d reflects the displacement of the projections.

The *distance* of two projected contigs $\pi(m) = (p_b,\ p_e)$ and $\pi(m') = (p'_b,\ p'_e)$ is given by the following function:

$$d\big(\pi(m),\ \pi(m')\big) = \begin{cases} p'_b - p_e & \text{if } p_b < p'_b \\ p_b - p'_e & \text{if } p_b > p'_b \\ -\min\{|m|,\ |m'|\} & \text{if } p_b = p'_b\ . \end{cases} \tag{4.1}$$

If the matches refer to different reference genomes, the distance of their projections is undefined. Note that the term 'distance' is used here in the sense of 'displacement', and d is *not* a metric in the mathematical sense. For example, d is negative if the projected contigs overlap.

Now, we define the edge-weighted *contig adjacency graph* $G_{\mathcal{C},\mathcal{R}} = (V, E)$, which is the central concept of our layouting approach. The contig adjacency graph contains for each contig $c_i \in \mathcal{C}$ two vertices: l_i as the *left connector* and r_i as the *right connector* of contig c_i. The set of vertices V is thus defined as $V = \{l_1, \ldots, l_n, r_1, \ldots, r_n\}$. A function $\mathit{contig}(v)$ refers to the contig for which vertex v represents the left or right connector.

The contig adjacency graph $G_{\mathcal{C},\mathcal{R}}$ is fully connected, that is $E = \binom{V}{2}$, and we term $A = \{\{v, v'\} \in E \mid \mathit{contig}(v) \neq \mathit{contig}(v')\}$ the *adjacency edges* that connect the contigs among each other. The remaining edges are termed the *intra contig edges* $I = \{\{l_1, r_1\}, \ldots, \{l_n, r_n\}\}$. Each intra contig edge $\{l_i, r_i\} \in I$ 'represents' the contig c_i. To ease the understanding of this concept, Figure 4.2 on the following page shows an exemplary graph with four contigs.

By using a left and a right connector per contig, the adjacency edges encode the relative orientation of two contigs. The edge $\{r_a, l_b\}$, for example, states that the right end of contig c_a is adjacent to the left end of contig c_b, which means that both contigs have the same direction. We illustrate this adjacency edge by $\xrightarrow{c_a}\xrightarrow{c_b}$, or equivalently by $\xleftarrow{c_b}\xleftarrow{c_a}$, since the edge is not directed. The adjacency edge $\{l_a, l_b\}$, on the contrary, indicates that one of the contigs is reversed towards the other, depicted

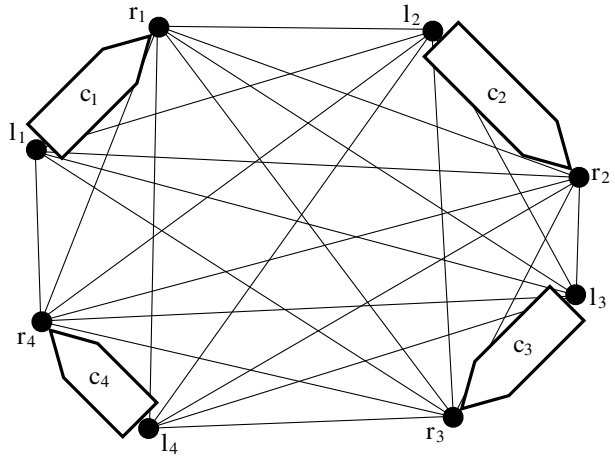

Figure 4.2: Exemplary contig adjacency graph containing four contigs. The intra contig edges I are omitted. Instead of this, the contigs are drawn as pictograms.

as either $\overleftarrow{c_a}\,\overrightarrow{c_b}$, or $\overleftarrow{c_b}\,\overrightarrow{c_a}$. The last possible case of an adjacency of the two contigs would be $\overrightarrow{c_a}\overleftarrow{c_b}$, or $\overrightarrow{c_b}\overleftarrow{c_a}$ respectively, given by the edge $\{r_a, r_b\}$.

The edge weights of the contig adjacency graph are calculated with a function $w\colon E \to \mathbb{R}_0^+$ that will be defined in Section 4.1.2. We are primarily interested in the weights of the adjacency edges A. These shall provide a score of how likely the involved connectors are adjacent with respect to the reference genomes $g_r \in \mathcal{R}$. We call this the *support* of two contigs being adjacent. For convenience, we assume that we can retrieve the support values after their computation from the symmetrical *contig adjacency matrix* W, in which a row (or column) contains all weights involving a particular contig connector:

$$W = \left(\begin{array}{ccc|ccc} 0 & \dots & w(\{r_1, r_n\}) & w(\{r_1, l_1\}) & \dots & w(\{r_1, l_n\}) \\ \vdots & \boxed{\overrightarrow{c_i}\,\overleftarrow{c_j}} & \vdots & \vdots & \boxed{\overrightarrow{c_i}\,\overrightarrow{c_j}} & \vdots \\ w(\{r_n, r_1\}) & \dots & 0 & w(\{r_n, l_1\}) & \dots & w(\{r_n, l_n\}) \\ \hline w(\{l_1, r_1\}) & \dots & w(\{l_1, r_n\}) & 0 & \dots & w(\{l_1, l_n\}) \\ \vdots & \boxed{\overleftarrow{c_i}\,\overleftarrow{c_j}} & \vdots & \vdots & \boxed{\overleftarrow{c_i}\,\overrightarrow{c_j}} & \vdots \\ w(\{l_n, r_1\}) & \dots & w(\{l_n, r_n\}) & w(\{l_n, l_1\}) & \dots & 0 \end{array}\right) \tag{4.2}$$

Each quadrant of this matrix contains that type of adjacency edges as depicted in the central boxes. The weights in the main diagonal are set to zero since they correspond to self-loops of the contig connectors which are not considered in our contig adjacency graph.

We call the sum of the weights of all edges incident to a node $v \in V$ the *total support* of that node, denoted by $\mathcal{S}_v = \sum_{v' \in V} w(\{v, v'\})$. In Matrix (4.2), this is equivalent to

the row- (or column-wise) sum of a contig connector. To estimate how significant an adjacency edge $e = \{v, v'\} \in A$ is for a given contig connector $v \in V$, we consider its *relative support*: $\mathcal{S}_v^{\text{rel}}(e) = \frac{w(e)}{\mathcal{S}_v}$. Intuitively, this fraction tells how specific the connection is for the given contig connector. A single high weighted edge results in a relative support close to 100%, while many equally good connections will lower the value. Note that in general $\mathcal{S}_v^{\text{rel}}(\{v, v'\}) \neq \mathcal{S}_{v'}^{\text{rel}}(\{v, v'\})$.

4.1.2 Weighting the Adjacency Edges

For each intra contig edge $e \in I$, we set the weight $w(e) = 0$ since these do not tell us about the relationship between the contigs. For the other edges, let $e = \{v_i, v_j\} \in A$ be an adjacency between the contigs $c_i = \textsf{contig}(v_i)$ and $c_j = \textsf{contig}(v_j)$. Then, the total weight of this adjacency edge is defined as

$$w(e) = \sum_{g_r \in \mathcal{R}} w_r(v_i, v_j) \tag{4.3}$$

where the (symmetric) function w_r defines the support of this adjacency with respect to a single reference genome g_r. Each support $w_r(v_i, v_j)$ is based on the matches of the involved contigs on that reference:

$$w_r(v_i, v_j) = \sum_{m \in \mathcal{M}_i^r,\, m' \in \mathcal{M}_j^r} s\Big(d\big(\pi(m),\, \pi(m')\big)\Big) \cdot \textsf{qhits}(m) \cdot \textsf{qhits}(m') \ . \tag{4.4}$$

Here, d is the distance between two projected contigs, see Equation (4.1), and $s(d)$ is a suitably defined scoring factor to weight the matches based on the distance of their projections. Note that instead of the number of q-hits as quality measure for the matches, also the *BLAST* bit-score can be used.

The scoring factor $s(d)$ is based on the following observations concerning the distances of projected contigs:

1. Projected contigs that are not adjacent have, in general, a high distance and should obtain a low score. Adjacent contigs should gain a high score for usually having a distance close to zero. However, as illustrated in Figure 4.3 (a), the distance of two projected contigs can reach positive values due to insertions into the reference genome. Similarly, the distances can be negative if the projections overlap, which is the case if there are insertions in the newly sequenced genome. This case is shown in Figure 4.3 (b). Note that an insertion in one genome corresponds to a deletion in the other, and vice versa.

   We model the effect of insertions and deletions in the reference genome with a random variable $X \sim \mathcal{N}(\mu_X, \sigma_X^2)$ that satisfies a Gaussian distribution. The expected value μ_X is zero, and the standard deviation σ_X correlates with the size of insertions and deletions.

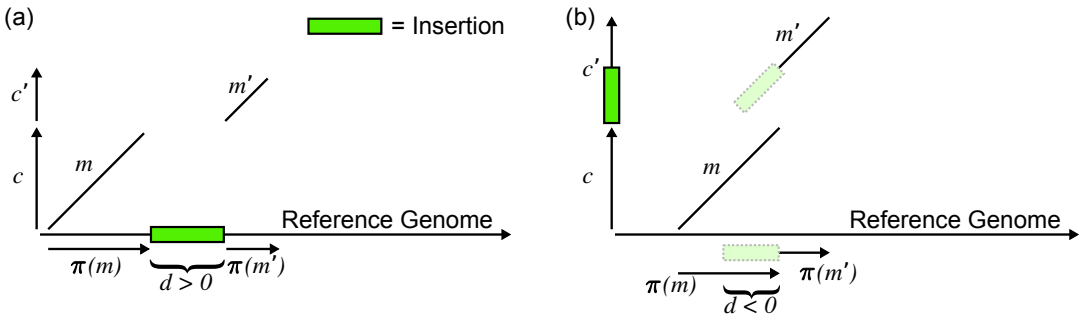

Figure 4.3: (a) An insertion in the reference genome leads to a positive distance, whereas (b) an insertion in a contig leads to a negative distance.

2. In the fragmentation phase of a sequencing project, often fragments disappear such that there are no reads for this fragment. Additionally, the assembler software might discard unreliable reads at the ends of the contigs, which also leads to missing sequence information. If pieces of the newly sequenced genome are missing, the same situation arises as if there is an insertion into the reference genome, which causes positive distances.

   The effect of missing sequences can be modeled according to a random variable $Y \sim \mathcal{N}(\mu_Y, \sigma_Y^2)$ that also obeys a Gaussian distribution. Here, the expected value μ_Y models the average size of the lost fragments, and σ_Y^2 models their variation.

To take the mentioned effects into account, we use a superposition of both Gaussian distributions. Assuming that X and Y are independent, their sum is also distributed according to a Gaussian distribution [27, Ch. 11.2]: $X + Y \sim \mathcal{N}(\mu_X + \mu_Y, \sigma_X^2 + \sigma_Y^2)$. Hence, we set $\mu = \mu_Y$ and $\sigma^2 = \sigma_X^2 + \sigma_Y^2$, and use for our scoring the combined Gaussian distribution $\mathcal{N}(\mu, \sigma^2)$:

$$s(d) := \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{d-\mu}{\sigma}\right)^2} . \tag{4.5}$$

The combined parameters μ and σ can be estimated from already finished sequencing projects, as done in the evaluation in Section 6.1.3. Based on this estimation, we can assume that both $\mu > 0$ and $\sigma > 0$.

4.1.3 Creating a Basic Contig Adjacency Graph

Using the above definitions, a contig adjacency graph can be created for a set of contigs and a set of reference genomes as described in Algorithm 1. For each reference genome, the contigs are matched independently. Based on the projected contigs, the weights of the edges are calculated and integrated into the contig adjacency graph.

Algorithm 1: Basic Contig Adjacency Graph Creation

Input: set of contigs $\mathcal{C}$, set of related genomes $\mathcal{R}$
Output: contig adjacency graph $G_{\mathcal{C},\mathcal{R}}$

```
initialize contig adjacency graph G_{C,R} = (V, E) with w(e) = 0 for all e ∈ E
foreach reference genome g_r ∈ R do
    foreach contig c_i ∈ C do
        find matches M_i^r
        calculate for all matches m ∈ M_i^r the projected contig π(m)
    end
    foreach pair of contig connectors e = {v_i, v_j} ∈ A do
        compute the weight w_r(v_i, v_j) and add it to w(e)
    end
end
```

This procedure is the basic way to create a contig adjacency graph and we will introduce enhancements to it in Section 4.3. These were designed to improve the reliability of the edges if additional information is available. For example, we provide a modified edge weight function that is able to incorporate the phylogenetic distances of the involved species.

Properties of the Graph The weights of a contig adjacency graph created with Algorithm 1 have some noteworthy properties: Strong matches of two contigs that have a high number of q-hits produce a high support of the employed contig connectors if their projected contigs occur close on a reference genome.

Small scale evolutionary events, like mutations or smaller rearrangements, may result in a fragmentation of large matches into several weaker ones. However, our scoring should be robust to this, since the projected contigs of these matches coincide with the projection of the unfragmented match. The distances are thus alike, and consequently the weights of all match fragments contribute to a high weight in total. In contrast, projected contigs of weak matches that occur only by chance usually have large distances to other projections such that the score factor, and thus also the weight contributed to an adjacency support, is low.

The approach of Algorithm 1 assumes that the reference genomes are closely related. In fact, large scale rearrangements can be problematic for the significance of the adjacency support values: A large insertion on a reference, for instance, causes an adjacency of two projected contigs surrounding it to gain a low weight. In such cases it is advantageous that we use several reference genomes. A single reference genome that does not have this insertion is sufficient to also gain a noticeable support for the proper adjacency. If we look at large scale inversions, these can cause high weight edges for contigs that are in fact not adjacent. When using several refer-

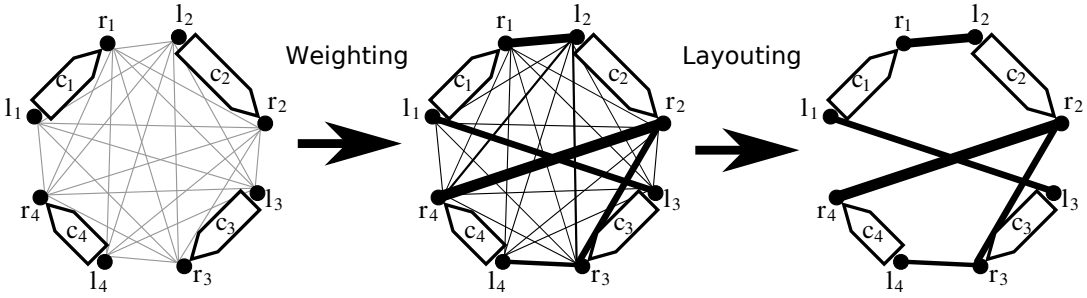

Figure 4.4: Overview of our approach to find a layout for the contigs. First, the edges of the contig adjacency graph are weighted according to Algorithm 1, and then the best adjacencies are extracted to a layout graph, as described in Section 4.2.2.

ence genomes, this can lead to conflicting information for a contig connector since we do not know which of the high weight adjacencies is correct.

Another property of the contig adjacency graph concerns repetitive contigs. These have, in terms of adjacencies, several neighbors. In the adjacency matrix, this becomes apparent by several high entries in a row (or column) if the contigs are also repetitive on at least one of the reference genomes. This means that a repetitive contig usually has not one but several supported adjacencies per contig connector.

To summarize, the contig adjacency graph contains the collected information given by all matches to the references. Figure 4.4 (center) illustrates this property. Although rearrangements and repetitive contigs may cause high weights that have to be handled with care, we can use the calculated edge weights to find a layout of the contigs. The next section focuses on how to extract the most promising edges for this task.

4.2 Finding a Layout of the Contigs

Given a contig adjacency graph, we want to find a subgraph of it that contains all relevant adjacencies in order to ease the gap closure phase of a sequencing project. We call any subgraph with this property a *layout graph* of a set of contigs. Ideally, the edges of a layout graph build a single path that contains each contig once. We call this a *linear layout* of the contigs.

4.2.1 Traveling Salesman Tour Through the Graph

A natural approach to discover a linear layout is to find a tour of maximal weight that contains each contig exactly once, and in a specified direction. With the following minor modifications of the graph, this becomes equivalent to finding a shortest Hamiltonian cycle: All edge weights have to be converted to distances. This is done by replacing each edge weight w by $c-w$ where c is a constant that is not lower than

the maximum weight in the graph. Further, we add an intermediate node between the left and the right connector of each contig to ensure that each contig is incorporated exactly once, and only in one direction. The modified graph is then defined as $G'_{\mathcal{C},\mathcal{R}} = (V', E')$ with $V' = V \cup \{v_i \mid 1 \leq i \leq n\}$ and $E' = A \cup \{\{l_i, v_i\}, \{v_i, r_i\} \mid 1 \leq i \leq n\}$. The distance of all edges that lead to an intermediate node v_i is set to 0.

A shortest Hamiltonian cycle in the modified graph G' defines an order as well as the relative orientation of all contigs. Thus, any algorithm to solve the *traveling salesman problem* (TSP) can be used to find a linear layout of the contigs that is optimal with respect to the weights of the underlying contig adjacency graph. The naive approach, which is to calculate the lengths of all possible tours, is already unfeasible for more than a few nodes since the TSP is NP hard. Besides the naive approach, there are many other algorithms to solve the TSP [6]. These can be divided in runtime heuristics that find an optimal solution while being in the expected case faster than the naive approach, and exactness heuristics that very quickly find a good – although not necessarily optimal – solution.

Runtime heuristics like branch-and-bound algorithms can be used to solve problem instances of up to several nodes. However, in the worst case, they still need exponential time. For hundreds of contigs, the time demand to compute a solution is thus still not feasible.

In comparison to algorithms providing exact solutions, many exactness heuristics are much faster. A very fast greedy algorithm, for instance, is the *multi-fragment heuristic* [13] that proceeds as follows: First the edges of the graph are sorted by increasing distance and then added in this order into an initially empty set of path fragments. Whenever an involved node would exceed the maximal degree of two, or if a path fragment would create a cycle, the edge is skipped. The only exception to the latter is the final Hamiltonian cycle of length n. This "best connection first" procedure creates multiple low distance path fragments which are merged sooner or later.

The multi-fragment heuristic is well suited to find a linear layout of the contigs. Although it produces not necessarily an optimal tour, it gathers high support adjacencies. This local optimization of the adjacencies might in the context of contig layouting be more valuable than spending much time to globally optimize the tour.

In brief, a linear layout of the contigs that is optimal, or near optimal, with respect to the adjacency edge weights, can be computed using a suitable TSP algorithm. However, we found out that a linear layout of the contigs is not necessarily biologically relevant. This is mainly due to an arbitrary placement of repeated or rearranged regions. A method that provides a unique layout where possible, but also points out alternative solutions where necessary, may be more useful in practice. In the next section, we present our approach to tackle this problem.

4.2.2 Fast Adjacency Discovery Algorithm

Our approach to discover relevant adjacencies from a contig adjacency graph is based on the multi-fragment heuristic introduced in the previous section. We chose this greedy heuristic because it seems natural to first incorporate those adjacencies into a layout that are most promising to be investigated for gap closure.

As already indicated, repeating or rearranged regions may prohibit an unambiguous linear layout of the contigs. Repeating contigs create cycles in a possible path, and rearrangements can lead to conflicting adjacencies of a contig. To account for this, we relax the constraints of the multi-fragment heuristic: First, we allow cycles that could appear due to repetitive contigs. Secondly, when inserting an edge, we permit one of the incident nodes, but not both, to exceed a degree of two. This allows to also include conflicting information into a layout.

Our procedure to extract a layout graph is formally described in Algorithm 2. The input is a contig adjacency graph, for instance created with Algorithm 1. We start with the most promising edges – those with the highest support – and integrate them one by one into the initially empty layout graph, except if both of the involved contig connectors have already been integrated. When an adjacency edge is integrated, we also say the edge is *realized* in the layout. To avoid that too faint edges are realized in the final layout, one can check if the relative support of an edge to be integrated with respect to both contig connectors is above a certain threshold.

The result of our greedy algorithm is a layout graph, as exemplified in Figure 4.4 (right). There, the adjacency $\overset{c_4}{\longrightarrow}\overset{c_2}{\longleftarrow}$, having the highest weight, is realized first. The edge $\{r_3, r_2\}$ is introduced later, although conflicting, because it has a higher weight than $\{l_4, r_3\}$, and thus r_3 is not occupied yet. Finally, the adjacency $\overset{c_4}{\longleftarrow}\overset{c_3}{\longleftarrow}$ is realized, since l_4 is the last free connector.

The resulting layout graph usually does not describe a linear layout, and in general the graph is not necessarily connected. However, it contains many of the strongly supported adjacencies of the contigs and includes at least one edge for each contig connector. The best edges are realized first and then padded with possibly conflicting information such that all contig connectors are included in the layout.

All unambiguously incorporated contigs can be helpful in the finishing process to guide the primer design for gap closure. Yet, knowing about conflicting edges can also be contributive since these indicate possible rearrangements or show the influence of repetitive contigs. So, instead of pinning the result down to a single, possibly wrong, linear layout of the contigs, we prefer to output the best possibilities. Nonetheless it should be kept in mind that rearrangements can cause seemingly good adjacencies that do not belong to a correct layout.

Algorithm 2 is our basic approach to layout contigs. Section 4.4 shows variations of this layouting, for example a special treatment for repetitive contigs.

Algorithm 2: Basic Contig Adjacency Discovery

Input: contig adjacency graph $G_{\mathcal{C},\mathcal{R}} = (V, I \cup A)$
Output: layout graph L of the contigs

```
create empty layout graph L = (V_L, E_L) with V_L = ∅ and E_L = ∅
foreach adjacency edge e = {v, v'} ∈ A, sorted by decreasing weight w(e) do
    if |V_L ∩ {v, v'}| ≤ 1 then
        V_L = V_L ∪ {v, v'}
        E_L = E_L ∪ e
    end
end
E_L = E_L ∪ I
```

4.3 Enhancements of the Graph Creation

So far, we introduced the contig adjacency graph and its creation, as well as a layouting algorithm to extract the interesting edges. This section explains how additional information can be used in the graph creation phase to improve the reliability of the calculated edge weights.

4.3.1 Including Phylogenetic Distances

The reference genomes used for layouting are typically related to different degrees to the contigs genome. Sometimes, a phylogenetic tree of the species is available that contains phylogenetic distances of the species towards each other. If not, such a tree can be generated even if some genomes are not completely assembled yet, for example, from the highly conserved 16S ribosomal RNA.

Given a phylogenetic tree, we can use it to weight the matches according to the relatedness of the reference genome. Assuming that between closer related species less rearrangements have taken place, this weighting also helps to avoid contradicting edges in the contig adjacency graph that can be caused by rearrangements.

When a phylogenetic tree $\mathcal{T}$ of the involved species is available, we can use the contained evolutionary distances to change the score factor $s(d)$ from Equation (4.5) at page 48 to:

$$s_{\mathcal{T}}(d, d_{\mathcal{T}}) := \frac{1}{d_{\mathcal{T}} \cdot \sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{d-\mu}{d_{\mathcal{T}}\cdot\sigma}\right)^2} \tag{4.6}$$

where $d_{\mathcal{T}}$ is the tree distance to the particular reference genome.

As illustrated in Figure 4.5, a higher tree distance $d_{\mathcal{T}}$ allows larger insertions and deletions, but scores the reliability of the matches to more distantly related genomes to a lesser degree. To use the score factor of Equation (4.6) as displayed, it

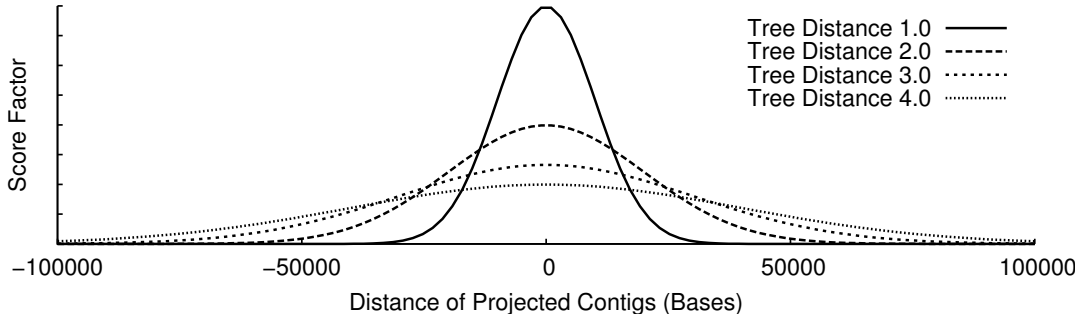

Figure 4.5: Influence of the phylogenetic tree distance given in Equation (4.6). Parameters $\sigma = 10,000$ and $\mu = 0$ arbitrarily chosen for illustration.

is advisable to normalize the phylogenetic distances such that the closest reference species has a distance of one and less related genomes have a higher distance.

The basic layouting algorithm, together with this phylogenetic tree enhancement has been published [41], and the implementation is known as *treecat*, the "phylogenetic tree based contig arrangement tool". Later, in the evaluation in Chapter 6, we will refer by this name to the corresponding algorithms and enhancements.

4.3.2 Integrating Additional Information

In sequencing projects, often additional information occurs which can be helpful to layout the contigs. Section 2.3.2 mentions for example mate pairs, fosmid libraries, or optical restriction maps. These pieces of information can be included into our approach by modifying the weights of the contig adjacency graph after their computation, which then influences the predicted adjacencies in a layout graph. If expert information indicates that two contigs are not adjacent, it suffices to set the appropriate edge weight to zero. This contig connection will not occur in the result afterwards. On the contrary, if for example fosmid end sequencing shows that two contigs are adjacent and quite close, the incorporation of that edge into the layout graph can be forced by setting the corresponding edge weight to the maximum weight of the graph.

4.4 Variations of the Contig Layouting

In applications of the basic layouting algorithm to real data, it became clear that rearrangements in the reference genome as well as repetitive contigs are the main reason for misleading predictions of adjacent contigs in the layout graph. In the next section, we discuss the detection and handling of rearrangements. After that, we deal with repetitive contigs and give an algorithm to include them more appropriately in Section 4.4.2.

4.4.1 Handling Rearrangements

Rearrangements that happened between the newly sequenced genome and the reference genome can cause false or missing adjacencies in a computed layout of the contigs. We first consider rearrangements due to insertions and deletions, and then examine how to deal with large scale inversions.

An insertion in one of the genomes, for example caused by horizontal gene transfer, can not be distinguished from a potential deletion of that sequence block in the other. In reference based contig layouting, both cases can thus be treated equivalently. If the size of a respective sequence block is rather small, the rearrangement is sufficiently handled by our scoring factor for the distances of projected contigs, as described in Section 4.1.3.

On a larger scale, it depends which part of the contig was inserted or deleted: If the whole contig is affected, then there is no information how to layout this contig. The only chance in this case is to use more, or other related references in the hope that they contain the necessary information. If only an inner part, or either side of the contig is inserted or deleted, the remaining parts that match do contribute to the weight of the adjacency edge. The support is then proportional to the size of the corresponding matches.

Another type of large scale rearrangements that occur frequently in prokaryotic genomes are inversions [26]. In this case, using more reference genomes likely introduces even more conflicting edges. While simple inversion scenarios might successfully be untangled by examining the matches, a series of overlapping inversions can be impossible to resolve.

One suggestion to detect which of the contigs show misleading edges due to inversions is to cluster the matches of a contig to find their main diagonals, for example with a clustering by linear regression [34]. Contigs that contain significant forward as well as reverse complementary matches can be marked to warn that the corresponding edges may be unreliable.

4.4.2 Repeat-aware Layouting

As already stated, a meaningful biological order of the contigs might differ from a linear layout. Our concept of a layout graph withdraws the restriction of a linear layout by allowing alternative adjacency edges. This is in particular utilized by Algorithm 2 that allows cycles which can be caused by repetitive contigs. Still, this basic algorithm generates a layout graph in which each contig is incorporated only once. However, a repetitive contig has to be included several times into an adequate layout, since its sequence occurs several times in the genome.

While other approaches often try to avoid repetitive contigs completely (see Section 6.1.4), we found that ordering non-repetitive contigs first and adding connections to repeats later seems to be a good strategy. Unfortunately, the information

about repetitive contigs is not directly accessible from the contig adjacency graph. Therefore, we describe how to infer which contigs are repetitive based on their matches to a reference genome. Afterwards, we introduce an enhanced layouting of the contigs that integrates repetitive contigs as often as necessary.

Repeat Detection

There are several ways to detect repetitive contigs. One possibility is to find known repetitive sequences on the contigs that are stored in a database as done, for example, by *RepeatMasker* [86]. However, we aim at a de-novo repeat detection that is based on the provided sequence data. Therefore, we chose to use the matches to a reference genome in order to distinguish between repetitive and non-repetitive contigs. We call the latter for the sake of a shorter notation from now on *regular contigs*.

Using the matches to detect repeats assumes that repeating regions are conserved between closely related species. Surely, this is a very strong assumption, and we will discuss its sensibility in more detail later in Section 6.2.3. In accordance with this assumption, we consider for the repeat-aware layouting, instead of several references, only a single reference genome that is most closely related.

Given a set of matches $\mathcal{M}_i^r$ of contig $c_i \in \mathcal{C}$ to a reference genome $g_r \in \mathcal{R}$, we first determine which matches are repetitive, and from this we derive whether the whole contig can be considered as repetitive.

We call $m = \big((s_b, s_e), (t_b, t_e)\big) \in \mathcal{M}_i^r$ a *repetitive match* if there exists another match $m' = \big((s'_b, s'_e), (t'_b, t'_e)\big) \in \mathcal{M}_i^r$ such that

(i) the contig substring of m is included in the substring of m':
$s_b \geq s'_b$ and $s_e \leq s'_e$, and

(ii) the match positions on the reference are not overlapping:
$\{t_b, \ldots, t_e\} \cap \{t'_b, \ldots, t'_e\} = \emptyset$.

Since the exact positions of the matches may vary, depending on the matching procedure that is used, we allow for condition (i) a slack of ρ_1 times the length of m. By default we use a value of 10% for ρ_1.

Based on this, we speak of a *repetitive contig* if the contig has at least one repetitive match m of sufficient length. Sufficient means that at least ρ_2 percent of the contig is covered by the repetitive match: $s_e - s_b \geq \rho_2 \cdot |c|$. As default we set ρ_2 to 90%. In the following let $\mathcal{C}_R \subset \mathcal{C}$ be the set of repetitive contigs. We consider all contigs that are not repetitive to be regular.

An advantage of this approach is that the matches which are needed to create the contig adjacency graph of Section 4.1 can also be used for detecting repeats. However, contigs that are repetitive on the newly sequenced genome, are not necessarily repetitive on the employed reference genome.

To extend, as well as verify, the prediction of repetitive contigs, one can use the read coverage information obtained in the assembly phase that is discussed in Section 2.3.2. This even allows to estimate how often a repetitive contig has to be included. Another possibility to verify the repetitive contigs is to study them in the tangle structure of the de Bruijn (sub-)graph of all reads (or contigs) of the genome to be assembled [4].

Repeat-aware Layout Algorithm

Having the repetitive contigs identified, we show how to use this information to compute an appropriate layout of the contigs. To this end, we adopt Algorithm 2 from page 53 to be aware of repetitive contigs and include them appropriately. The overall strategy is to distinguish between regular and repetitive contigs and to process both sets one after another. The absence of repetitive contigs in the first set implies that most contigs should have exactly two neighbors. Following this observation, we will begin to create a simple linear chaining of the contigs. After that, we explore how the repetitive elements can be integrated into this initial layout in a meaningful way.

Layouting the Non-repetitive Parts of the Genome To create an initial layout graph of the regular contigs, we realize their edges in a two pass procedure. The first pass is basically a variation of Algorithm 2 that creates linear chains. In the second pass we realize additional edges to contigs that would be missed otherwise.

The complete algorithm to create an initial layout graph is listed in Algorithm 3. The single parts are explained in the following: In line 1 of Algorithm 3, the contig adjacency graph is created. Note that it is built for repetitive *and* regular contigs, thus the procedure starts with matching *all* contigs onto the given reference genome. Like in Algorithm 1, all pairwise matches are used to calculate the adjacency support values. However, we introduce a slight modification that helps to reduce misleading edges for regular contigs. As mentioned before, contigs as well as matches can be repetitive. It can happen that a regular contig has repetitive matches if they are small enough, for example contig ends are often flanked by repeats. Such matches are ignored in the weight calculation in order to avoid misleading edges with high support that are based on unreliable repetitive matches. Of course, for repetitive contigs all matches are used.

After the contig adjacency graph has been built for all contigs, we create an initial layout graph of the regular contigs. Starting at line 2 of Algorithm 3, we proceed similarly to Algorithm 2. The difference is that an edge is realized if *both* of the contig connectors are not included yet. Algorithm 2 requires only that *one* contig connector is free. Thus, we generate multiple fragments of good adjacencies that are in general joined to larger chains during the course of the algorithm. This matches the expectation that since we do not resolve repetitive contigs at this stage, the result

Algorithm 3: Simple Chaining and Extension

Input: set of contigs $\mathcal{C}$, set of repetitive contigs $\mathcal{C}_R \subset \mathcal{C}$, reference genome g
Output: initial layout graph G_L of the regular contigs

1 create contig adjacency graph $G_{\mathcal{C},g} = (V, E)$, as in Algorithm 1, omitting repetitive matches for regular contigs
2 create empty layout graph $G_L = (V_L, E_L)$ with $V_L = \emptyset$ and $E_L = \emptyset$
3 $E_{\text{regular}} = \{\{v, v'\} \mid \textit{contig}(v), \textit{contig}(v') \notin \mathcal{C}_R \text{ and } \textit{contig}(v) \neq \textit{contig}(v')\}$
// find regular edges
4 **foreach** $e = \{v, v'\} \in E_{\text{regular}}$, sorted by decreasing weight $w(e)$ **do**
5 **if** $|V_L \cap \{v, v'\}| = 0$ **then**
6 $V_L = V_L \cup \{v, v'\}$
7 $E_L = E_L \cup \{e\}$
8 **end**
9 **end**
// find additional regular edges
10 **foreach** $e = \{v, v'\} \in E_{\text{regular}} \setminus E_L$, sorted by decreasing weight $w(e)$ **do**
11 **if** e has exactly one vertex u in V_L **and** $\mathcal{S}_u^{\text{rel}}(e) > \tau_1$ **then**
12 $V_L = V_L \cup \{v, v'\}$
13 $E_L = E_L \cup \{e\}$
14 **end**
15 **end**
16 $E_L = E_L \cup \{l_i, r_i\}$ for the left and right connectors l_i, r_i of all $c_i \notin \mathcal{C}_R$

should be a set of linear chains of the contigs which can also be present in the form of one or several cycles.

Up to line 9, we find appropriate neighbors for most regular contigs. However, if very small contigs lie between two large contigs, then we sometimes observe a *shadowing effect*, as illustrated in Figure 4.6: The adjacency edge between the large contigs can have a higher support that shadows the edge weights to the small contig. Thus, the algorithm would not realize the weaker edges with the consequence that the small contig is not included in the layout graph. This behavior is generally unwanted, but, as we will see in Algorithm 4, it can be advantageous for small repetitive contigs. That is why we do not abandon the effect, for example by ignoring the size of the matches in the weight function. Instead, we compensate the shadowing effect for the affected regular contigs by integrating them into the initial layout as good as possible. Starting in line 10 of Algorithm 3, we look at all edges not yet realized and see if they can append an unintegrated contig connector to the initial layout. Although the shadowing edge stays in the layout, in most cases the correct

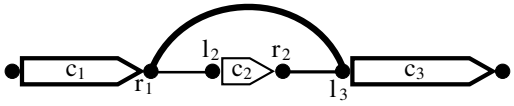

Figure 4.6: Shadowing effect: If a small contig c_2 is on a reference genome located between the larger contigs c_1 and c_3, in the contig adjacency graph the correct edges to c_2 can have a lower weight than the adjacency $\xrightarrow{c_1}\xrightarrow{c_3}$.

edges from the small contig will also be realized, resulting in a triangle shape of connected contigs. To control that only very specific edges are incorporated, we test whether the additional edge has a high relative support $\mathcal{S}^{\text{rel}}$ of at least τ_1.

Adding the Repetitive Contigs Starting with the initial layout graph constructed by Algorithm 3, the task is now to include the repetitive contigs into the layout.

Knowing the layout of the regular contigs helps to close the gaps in between. Repetitive contigs are, in contrast, not well suited for a primer-based closing of gaps since primers in repetitive sequences will bind unspecifically to several regions on the genome. This results in unspecific PCR products and should thus be avoided. Nonetheless, we believe that it is very helpful in the finishing phase of a sequencing project for a researcher to be informed whether repetitive contigs interrupt a gap of two regular contigs, or not. In the case of several repetitive contigs in a gap, their order plays, to our opinion, only a secondary role because this information cannot directly help in the finishing process: If both primers are based on repetitive contigs, this will produce even more unpredictable results.

Our idea in Algorithm 4 is therefore to place each repetitive contig as often as necessary between the corresponding regular contigs into the initial layout graph. Consequently, the important edges that we want to integrate in our initial layout are those which connect a repetitive contig with a regular one, see line 1 of Algorithm 4. We demand that the relative support of these edges with respect to the repetitive contig is higher than a threshold τ_2. This avoids the incorporation of arbitrarily weak edges. The edges between repetitive contigs are not considered in this approach, as motivated above.

For the interesting edges, we try to find for each involved regular contig connector a suitable counterpart that is also connected to the other end of the repetitive contig, as shown in lines 4 to 10 for the left connectors. This procedure is based on the following observation: As illustrated in Figure 4.7, a repetitive contig $c \in \mathcal{C}_R$ usually has several good edges for its right and its left connector leading to different regular contigs. The problem is to determine which edges belong to a particular repeat occurrence on the reference genome. The shadow effect, which was an obstacle for regular contig ordering, becomes here an advantage. In the example of Figure 4.7, the adjacency $\xrightarrow{c_1}\xrightarrow{c_5}$ has a high support, if the contigs c_1 and c_5 are only separated by the occurrence of the relatively small repeating contig c. Thus, the objective is to

Algorithm 4: Integration of Repetitive Contigs

Input: set of contigs $\mathcal{C}$, set of repetitive contigs $\mathcal{C}_R \subset \mathcal{C}$, contig adjacency graph $G = (V, E)$, initial layout graph G_L
Output: repeat-aware layout graph G_L of the contigs

1 let $E_{\text{rep}} = \{\{v, v'\} \mid \textit{contig}(v) \in \mathcal{C}_R, \textit{contig}(v') \notin \mathcal{C}_R \text{ and } \mathcal{S}_v^{\text{rel}}(\{v, v'\}) > \tau_2\}$
2 **foreach** edge $e \in E_{\text{rep}}$, sorted by decreasing weight $w(e)$ **do**
3 **if** $e = \{v_1, l\}$ contains the left connector l of a contig $c \in \mathcal{C}_R$ **then**
4 let r be the right connector of contig c
5 **if exists** $v_2 = \arg\max_{v \in V} \{w(\{v_1, v\}) \mid \{r, v\} \in E_{\text{rep}}\}$ **then**
6 duplicate l and r to l' and r'
7 $V_L = V_L \cup \{v_1, l', r', v_2\}$
8 $E_L = E_L \cup \{\{v_1, l'\}, \{l', r'\}, \{r', v_2\}\}$
9 remove $\{v_1, l\}$ and $\{r, v_2\}$ from E_{rep}
10 **end**
11 **else** *// $e = \{r, v_2\}$ contains the right connector r of a contig $c \in \mathcal{C}_R$*
12 perform lines 4 to 10 analogously
13 **end**
14 **end**

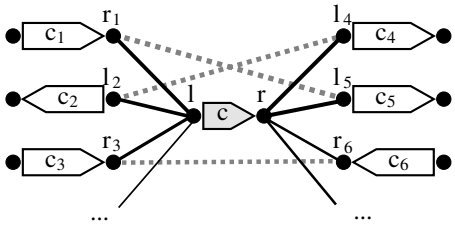

Figure 4.7: Typical scenario for the adjacency edges of a repetitive contig $c \in \mathcal{C}_R$. The dashed lines depict the best edge from a contig connector on the right to a contig connector on the left.

search for any regular node that is connected to either side of a repetitive contig, a suitable counterpart that is connected to the other side, such that the edge from the node to the counterpart has the highest weight.

This way, we find for each significant occurrence of a repetitive contig the two surrounding regular contigs with respect to the reference genome. For all occurrences, we add two new connectors of the repetitive contig and the appropriate edges to the layout graph.

Assuming that a reference genome is highly related, the variation of the basic layouting algorithm introduced in this section helps in two ways: First, it detects contigs that are repetitive and marks them in the output to avoid misinterpretation. Second, Algorithm 4 includes the repetitive contigs into a previously generated

initial layout such that a copy will be placed between two regular contigs for which the contig adjacency graph suggests a repeat occurrence. Unfortunately, it is not possible with this approach to propose an order of the repetitive contigs, if several are included between two regular ones.

Dealing with the repetitive contigs is a hard task and usually other algorithms filter them in order to not get confused. In our basic layouting algorithm, we do not filter repetitive contigs. However, they lead to conflicting information in the resulting layout graph. With the variation described in this section, we aim at resolving these conflicts. The proposed algorithms include repetitive contigs more appropriately and find a linear layout for the regular contigs. The contents of this section have been published [43], and we refer to the implementation of the algorithms in the evaluation chapter as *repcat*, the "repeat-aware contig arrangement tool".

We hope that the presented layouting algorithms of this chapter are helpful in a gap closing process, although we are aware that they do not produce a perfect layout of the contigs. This might in some cases even be impossible when relying only on related reference genomes to find the layout. That is why we want to appeal to the common sense of a researcher to see the process of layouting more like an experiment in a lab instead of an impeccable algorithm that finds the true layout of the contigs. The algorithms propose likely adjacencies that are based on the data, but they may fail to give a unique linear layout of the contigs. Nevertheless, when comparing our implementation with other related programs in Chapter 6, we show that our predictions provide competitive, and often even better results.

Chapter 5

Realization of the Software

The software containing the ideas and algorithms of Chapters 3 and 4 evolved gradually until it reached the current state. Instead of just describing this state, we want to sketch the most important steps of this development. To our opinion, this explains best how the current features emerged and which design decisions have been taken. The second part of this chapter briefly introduces some external programs and libraries that are employed within, or in combination with, our software.

5.1 Implementational Milestones

In the beginning of this Ph. D. project, the idea arose to use an existing C++ implementation of the *SWIFT* algorithm (see Section 3.2.3) to match contigs onto a related reference genome to aid in the process of genome finishing. This implementation was faster than *BLAST*, and it permitted to also match very large contigs. Naturally, the need came up to visualize the matches in order to inspect the reliability of the reference genomes, and to infer information about the order and orientation of the contigs. This need eventually led to the birth of *r2cat*.

5.1.1 *r2cat*

For the implementation of *r2cat*, we chose the programming language Java, mainly because of two reasons: Firstly, Java is a highly platform independent language such that our software runs on Windows, Linux, Mac OS, and also on Solaris which is the CeBiTec default environment. The second reason is that Java features default graphical capabilities due to the included graphical user interface (GUI) library Swing.

Accordingly, a first prototype of *r2cat* used Swing to provide a basic dot plot visualization for matches that were previously generated using the above mentioned *SWIFT* C++ implementation that was developed by Kim Rasmussen. At this stage, we also devised and implemented a rudimentary ordering of the contigs based on

their 'center of mass' which is in principle the mean of all matches on a reference genome. However, this concept had a few flaws and such we adopted an idea of Jochen Blom which resulted in the simple contig mapping as explained in Section 3.3.3.

The next step in the development was driven by the wish to create a stand-alone application for matching and visualization that was independent of calling external programs. Consequently, the matching idea of *SWIFT* was reimplemented in Java.

At this point in time, we decided to license the software under the general public license (GPL) that allows users to obtain and modify the source code free of charge. This decision is based on the belief that the scientific community profits from sharing and reusing code. Additionally, providing also the code allows other researchers to comprehend and also test our approaches more deeply than with an executable application alone.

Early users of the software requested additional features, like for example the possibility to export the ordered contigs into FASTA files, or to save pictures of the displayed synteny plots. These and several other usability features, like the ones introduced in Section 3.3, have been implemented over time. For exporting the synteny plots, we used existing code of the open source project FreeHEP that will be introduced in Section 5.2.1. Exemplary plots were already shown in Section 3.3.2.

The complete application of *r2cat*, including the FreeHEP graphics code, can be packed in a single Java Archive (JAR) file with a size of less than one megabyte. Using this archive, the program can be started with the Java Web Start Technology without the need for an installation.

In 2009, we wrote an applications note [42] introducing *r2cat*, and made the implementation available on the Bielefeld University Bioinformatics Server (BiBiServ).[1] Later, one of our bachelor students, Yvonne Herrmann, extended *r2cat* with code to automatically design primer pairs. The code is based on a *Perl* script that was developed by Jochen Blom and Christian Rückert to aid the finishing of in-house sequencing projects.

5.1.2 *treecat*

Simultaneously with the latest developments of *r2cat*, we also worked on the ideas described in Chapter 4. Here, it was helpful that the already implemented matching routine of *r2cat* could be reused to compute the matches to the reference genomes. Until mid 2009, we implemented the basic contig adjacency graph creation and the first layouting algorithms. Additionally to the layouting heuristic proposed in this thesis, we also implemented an exact branch-and-bound method which, however, becomes too slow for more than a dozen of contigs.

Subsequently, the ideas of *treecat* were published [41], and the implementation was made available to be started with Java Web Start from the BiBiServ.[2]

[1] http://bibiserv.techfak.uni-bielefeld.de/r2cat
[2] http://bibiserv.techfak.uni-bielefeld.de/treecat

The software *treecat* features a basic GUI where the user can select a FASTA file of the contigs, and for several related reference genomes. Additionally, a phylogenetic tree in Newick format can be specified that contains the distances of the species. The matches to the reference genomes are cached to avoid another time consuming matching if the algorithm is rerun, for example with different parameters. After matching, the contig adjacency graph is constructed like described in Section 4.1, and the layouting is performed using Algorithm 2 on page 53.

To visualize the generated layout graphs, we initially applied the Graphviz package (see Section 5.2.2). To this end, the computed layout graph was output in a textual format – the Graphviz DOT language – and then converted to a graphical representation using an external program.

However, the Graphviz visualization has some severe drawbacks: An excess of contigs and connections results in overlapping edges and nodes, which renders the graph unreadable since the output is static. Consequently interactive visualizations have been developed in two bachelor projects with the goal to make the *treecat* layout graphs more accessible and user friendly:

1. Christian Miele developed an integration of the prefuse graph drawing toolkit (see Section 5.2.4) as a replacement of the external Graphviz visualization. A layout graph created by *treecat* can thus interactively be assessed: Nodes can be moved, edges can be selected, and the graph can be zoomed and panned.

2. Annica Seidel implemented a local view to navigate through the complete contig adjacency graph. To this end, a single contig is displayed as center, and on both sides the adjacency edges to other contigs are shown, sorted by their adjacency support. A user can interactively traverse through the graph by clicking on the contigs. Additionally, edges can be marked as promising.

Both extensions were finished in early 2011 but are only tentatively attached to *treecat*. The successor *htscat* (see Section 5.1.4) includes them in a common framework.

5.1.3 *repcat*

By analyzing the results of *treecat*, we realized that repetitive contigs pose a problem in layouting. Consequently, we started in the beginning of 2010 to implement ideas how to handle repetitive contigs. To this end, we modified the existing code of *treecat* which resulted in a prototype of *repcat*. In fact, most code is very similar, the changes affect mainly the layouting and partially the contig adjacency graph creation. The ideas of *repcat* were published [43], but to date the implementation is not officially released as a tool. Reasons for this decision can be found in the evaluation of *repcat* in Section 6.2.3.

5.1.4 *htscat*

In late 2010, we started to combine the existing tools into a uniform application: The software *htscat*, the "high throughput sequencing contig arrangement toolsuite", is designed to be an extensible framework that includes different methods and algorithms to layout a set of contigs in order to help in the finishing process of prokaryotic genome sequencing projects.

The modular framework simplifies a possible extension with further code of other developers. Initial efforts were taken to integrate a repeat resolving approach (Ph. D. project of Patrick Schwientek) that is *not* based on related references but on information already provided in the assembly phase. Currently, the source codes of both projects are joined in a single repository,[3] and some matching parts of *r2cat* have been integrated into the other project.

Until now, *htscat* combines the functionality of *r2cat* and *treecat* in a single desktop application. The organigram shown in Figure 5.1 gives an overview about the incorporated components. The *r2cat* code base provides the matching routine and the visualization of synteny plots. Due to the *treecat* sources, we add the capabilities to handle multiple reference genomes simultaneously. The bachelor projects support this with an interactive visualization of the layout graphs. Besides prefuse and FreeHEP as external libraries, we use in *htscat* the NetBeans Platform framework that will be introduced in Section 5.2.3. It contributes greatly to the usability and user friendliness of the GUI and also supports to write modular code for a simplified extensibility of the application.

A screeenshot of the running application in Figure 5.2 on page 68 shows the most important components: The panel displayed on the left helps to organize different sequencing projects. In a folder like view, for each contig set, the reference genomes are shown on which the contigs were matched. Besides creating new projects in a wizard dialog, new reference genomes can be specified for any opened project.

The matches to each reference genome are visualized with the *r2cat* code, as displayed in the center of the application window. One or several references of a project can be selected to create a contig adjacency graph. The resulting layout graph produced by the *treecat* code is visualized with the implementations of the two bachelor projects that were mentioned above. In the given Figure 5.2 on page 68, these visual components are shown on the right and at the bottom of the window.

All components of the window can be freely reordered to adapt to the user's needs. For dual display use, each component can be undocked to an independent window. Thanks to the NetBeans Platform, all changes are remembered and restored when the application is started.

[3] http://htscat.svn.sourceforge.net/

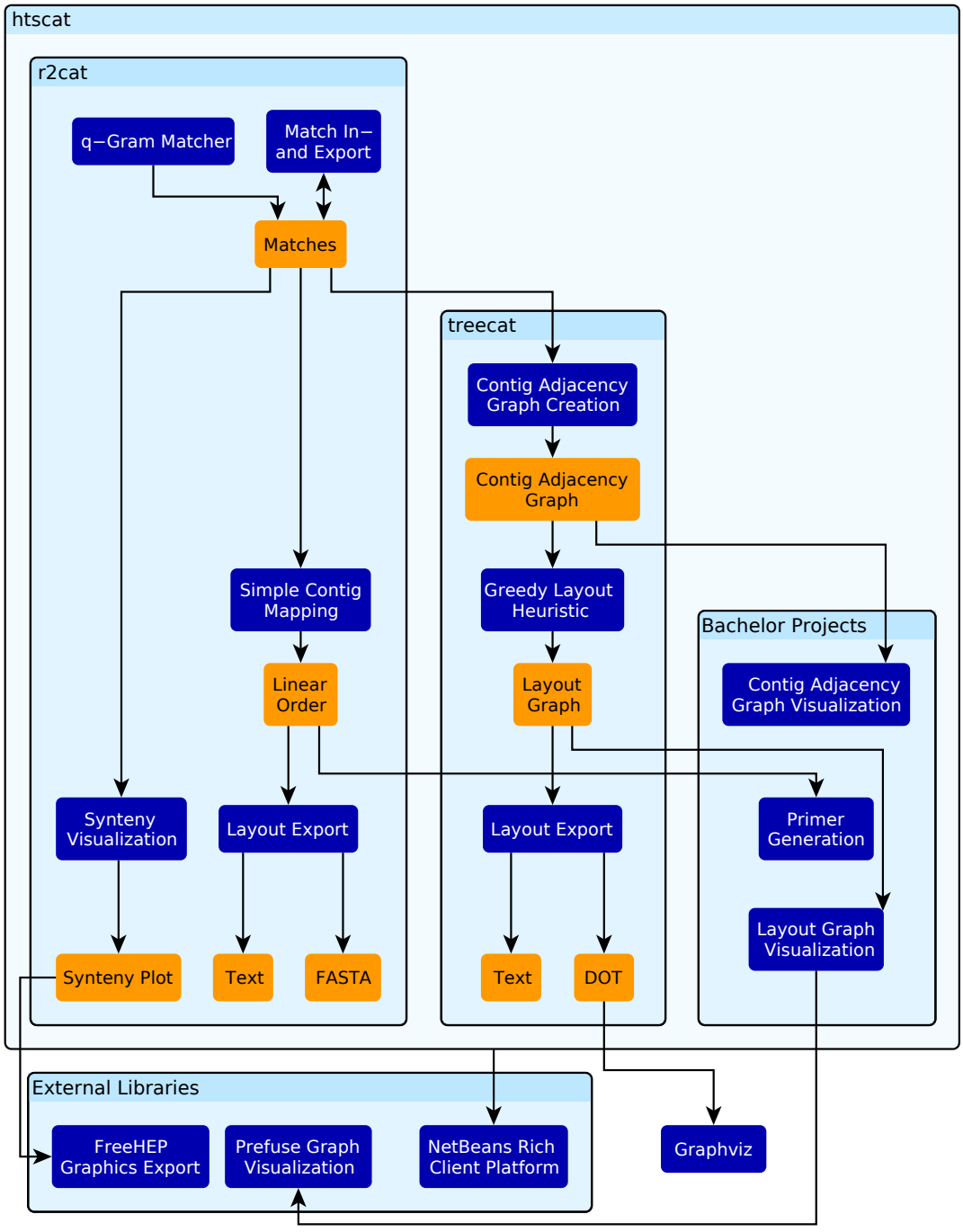

Figure 5.1: Overview of the *htscat* software structure. Program components are depicted with dark blue boxes, orange boxes refer to results or output. Arrows indicate dependencies or processing.

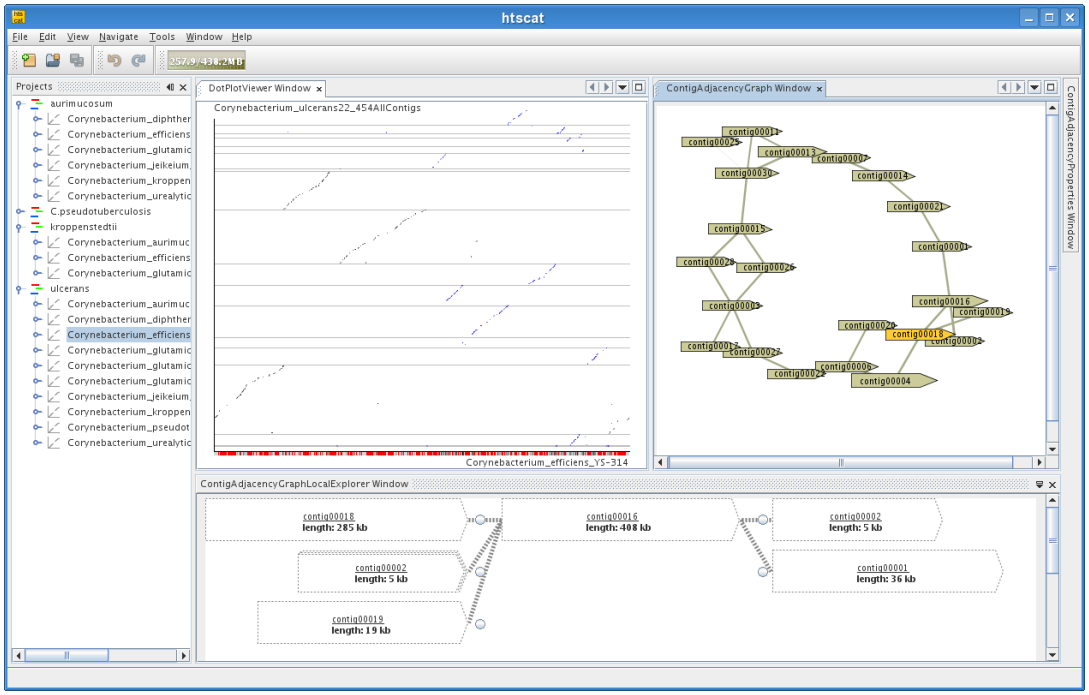

Figure 5.2: Screenshot of *htscat*. Top, from left to right: Project management, *r2cat* synteny plot visualization, layout graph created by *treecat*. Bottom: Local visualization of the contig adjacency graph.

5.2 External Software and Libraries

In this section, we describe the libraries and external software packages that we use within the *htscat* framework. Most information about the mentioned features was gathered from the specified project websites.

5.2.1 FreeHEP Graphics Export

FreeHEP[4] is a Java library that is licensed under the terms of the lesser general public license (LGPL) such that it can be shared and reused. Although the library was actually developed for the very specific subject of high energy physics (HEP), it also contains source code with a broader scope. In particular useful for our project is the vector graphics package that allows to export Swing components to a variety of different graphics formats. These include the vector based formats EPS, PDF, PS, and SVG, but also bitmap images like GIF or PNG are provided. There is even an export to TEX that draws a given component with PSTricks. We integrated that part of the FreeHEP code into our project which allows to export synteny plots to the above mentioned formats. Especially the SVG format is useful since it is easily

[4] http://java.freehep.org/

editable and provides an excellent possibility to create high quality, vector based graphics for publications.

5.2.2 Graphviz

The open source software package Graphviz[5] can be used to visualize graphs [33] that are provided in a textual representation, the so called DOT language. This language eases the description of graph structures which can be specified by simply providing all nodes and edges. The programs of the Graphviz package then perform a layouting to create a visual representation of the graph. Note that in this context, layouting refers to moving the contents of the graph such that it ideally has no overlapping edges or nodes. With this aim, different programs of the package are specialized for different graph types. The most prominent is perhaps *dot* which layouts hierarchical data represented in directed graphs. For our purpose, however, *neato* is better suited that layouts undirected graphs according to a spring model. An example of such a graph is shown in Figure 6.9 on page 89.

5.2.3 NetBeans Platform

The NetBeans Platform[6] is a framework that was originally programmed for the NetBeans Integrated Development Environment (IDE), but can be used more generically in the development of Swing applications. Thanks to a modular architecture, many functions can be reused in own desktop applications which improves the usability of the application without adding much overhead. The main reasons why we decided to use the NetBeans Platform for *htscat* are an advanced window management within the application, a powerful lookup concept for intra process communication, existing code for project management, and a module management with loose coupling of the components that allows to easily develop and distribute extensions to the software.

5.2.4 Prefuse Graph Visualization

The prefuse visualization toolkit[7] contains a library to display graph like data within Java programs [39]. In our software, we use it as an interactive replacement of the Graphviz visualization. It supports panning and zooming into the graph and is extensible enough to visualize custom nodes. Additionally, several different procedures can be used to place the nodes, like for example a force based layouting comparable to the spring model in Graphviz.

[5] http://www.graphviz.org/
[6] http://netbeans.org/features/platform/
[7] http://prefuse.org/

Chapter 6

Layouting Corynebacteria Contigs

This chapter demonstrates the performance of our proposed methods on real sequencing data. After introducing the datasets and discussing preparatory steps in Section 6.1, we evaluate and compare the capabilities of several layouting programs using different contig sets in Section 6.2. More specifically, *r2cat* is compared with other single reference based layouting approaches, and *treecat* is compared with a related approach that also uses multiple references. In the end, we assess the ability of *repcat* to cope with repetitive contigs.

6.1 Background and Preparatory Steps

This section gives a detailed description of the employed test data, explains the generation of a reference layout for the contigs, describes the estimation of the parameters for our contig adjacency graph creation, and finally introduces other related programs for contig layouting.

6.1.1 Description of the Datasets

For our evaluation, we used data from sequencing projects conducted at the CeBiTec. The assembled contigs that we used for our studies were kindly provided by Andreas Tauch and his group. All genomic sequences involved in this evaluation belong to the genus *Corynebacterium* that is comprised of Gram-positive rod-shaped eubacteria which typically have a high GC content.

Though the habitats and properties of the Corynebacteria are diverse, they are intensively investigated for two major reasons: Firstly, the genus contains several pathogen species, for example *Corynebacterium diphtheriae* which causes the severe disease diphtheria in humans. The second motivation that drives the research in this field is the industrial utilization of Corynebacteria, for instance in the production of

Table 6.1: The three contig sets used in the evaluation experiments. The number of repetitive contigs, as defined in Section 4.4.2, was determined with *r2cat* by matching the contigs onto their already finished genomes.

Contig Organism	# Contigs (# Repetitive)	Total Length (bp)	N50 Contig Size (bp)
C. aurimucosum ATCC 700975	73 (15)	2,736,233	82,833
C. kroppenstedtii DSM 44385	6 (1)	2,434,342	546,376
C. urealyticum DSM 7109	69 (15)	2,294,755	86,391

amino acids and nucleotides. *Corynebacterium glutamicum* is a prime example that is used in the mass production of glutamate for the food industry.

Several species of the *Corynebacterium* genus have been completely sequenced by now. Due to their diversity, however, there are also many ongoing sequencing projects. Corynebacteria genomes are thus perfectly suited to deliver test datasets for our evaluation: There are enough closely related finished genomes available as references, and new sequencing data are produced regularly.

For the evaluation, we prepared three datasets, each consisting of a set of contigs to be layouted and a set of reference genomes which are related to the contigs' genome. The three contig sets were sequenced and assembled from the species of *C. aurimucosum* [94], *C. urealyticum* [91], and *C. kroppenstedtii* [90], respectively. The complete genomes of these species have already been finished and are available from the NCBI website.

We discarded all contigs of the original assembly with a size of less than 500 base pairs, resulting in the sets shown in Table 6.1. This step is a common practice and was taken since small contigs that consist of only two or three reads are not very informative, but can be very confusing in the mapping process. The N50 contig size that is a more robust characterization for the size distribution of contig sets than the mean or median, does not change due to this action. As a reminder, the caption of Table 3.1 on page 36 contains the definition of the N50 contig size.

While *C. aurimucosum* and *C. urealyticum* consist of a few dozens of contigs, and required some effort to be closed, *C. kroppenstedtii* is a special case. Here, the initial sequencing and assembly resulted instantly in five very large contigs, ranging from 150–850 kbp. In this case, the finishing was rather straightforward, and reference based methods were not necessary. Nevertheless, we included this dataset in the evaluation to show that seemingly simple tasks for biologists might be more complex for computer programs when relying on the given data only.

As reference genomes for layouting the contigs, the three above mentioned finished genomes were used and were extended by choosing four additional publicly available Corynebacteria genomes, *C. diphtheriae*, *C. efficiens*, *C. glutamicum*, and *C. jeikeium*, that we downloaded from the NCBI website. The complete set of refer-

Table 6.2: Overview of the reference genomes that we employed in our evaluation. All sequences belong to the *Corynebacterium* genus.

Reference Genome	Replicon Type	Length (bp)	NCBI Number
C. aurimucosum ATCC 700975	chromosome	2,790,189	NC_012590
C. diphtheriae NCTC 13129	chromosome	2,488,635	NC_002935
C. efficiens YS-314	chromosome	3,147,090	NC_004369
C. glutamicum ATCC 13032	chromosome	3,282,708	NC_006958
C. jeikeium K411	chromosome	2,462,499	NC_007164
C. jeikeium K411	plasmid pKW4	14,323	NC_003080
C. kroppenstedtii DSM 44385	chromosome	2,446,804	NC_012704
C. urealyticum DSM 7109	chromosome	2,369,219	NC_010545

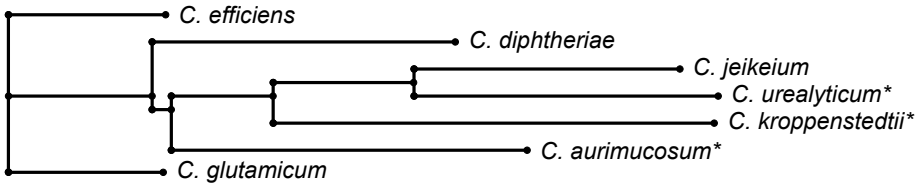

Figure 6.1: Phylogenetic tree of the involved Corynebacteria. For all species marked with an asterisk (*) the underlying contig data were available. The tree was calculated with EDGAR [15] and visualized with TreeVector [73].

ence genomes, including their accession numbers, is shown in Table 6.2. Note that the DNA molecules of all mentioned reference sequences occur as circular replicons in the bacteria.

To assess the evolutionary relationships of the involved species, we generated a phylogenetic tree of the species with the EDGAR framework [15] that applies Neighbor Joining [81] to distances of a set of *core* genes. The resulting tree is shown in Figure 6.1. For a more detailed illustration for the varying degree of rearrangements and synteny between the employed species let us reconsider some of the synteny plots that were already given in Chapter 3: Figure 3.7 on page 41 exemplarily shows four synteny plots involving the *C. urealyticum* contigs. While Figure 3.7 (a) shows a high degree of synteny and only few rearrangements to the *C. jeikeium* genome, Figure 3.7 (d) shows low synteny combined with many major rearrangements in the *C. aurimucosum* genome. Figures 3.7 (b) and 3.7 (c) show a similar inversion pattern but differing levels of synteny with respect to the reference genomes of *C. efficiens* and *C. diphtheriae*.

6.1.2 Determining a Reference Layout

The availability of the finished genomic sequences of the contig sets enables us to compute a *reference layout* which can be used as a 'standard of truth' when comparing the layouts generated by the different programs.

The reference layout for each set of contigs was devised by mapping them with *r2cat* onto their corresponding finished genome. Three of the *C. aurimucosum* contigs, with a size of 28 kbp in total, did not match on the finished genome and could thus not be included into the reference layout. The explanation for this is that the sequences belong to the *C. aurimucosum* plasmid pET44827.

In the process of mapping, *r2cat* revealed that all contig sets contain repetitive contigs; their quantity is given in Table 6.1. Due to these repetitive contigs, a reference layout created by simple mapping is not necessarily reliable. In general, repetitive contigs map non-uniquely to multiple locations on the genome. However, their actual placement in the linear layout of *r2cat* is merely by chance since a repetitive contig is placed on a single occurrence where it has the most matches. Whether an adjacency of two regular contigs is interspersed with repetitive contigs is thus only a matter of chance.

To account for this, and to ease the evaluation, we relabeled the contigs of each dataset. We prefixed all regular contigs with a 'c' and all repetitive contigs with an 'r'. The regular contigs were then numbered consecutively in their true order such that adjacencies can easily be seen. Repetitive contigs were numbered too, but only to distinguish them. Their arbitrary numbers do not contain any information about adjacencies.

For the following evaluation, we created two multiple FASTA files for each contig set: The first file contains all regular contigs of the reference layout in their original order and orientation, however with the FASTA identifiers renamed as stated above. This allows to examine whether the programs predict the correct adjacencies of the regular contigs. At the same time, the file does not contain any further hints due to pre-ordered or already oriented contigs. Since the repetitive contigs are removed, the programs ideally recover the true layout which is c00, c01, c02, c03, ..., and so forth. The second file is created likewise, but it also contains the repetitive contigs. This file was used in the evaluation of *repcat* that is specifically designed to handle these contigs.

6.1.3 Parameter Estimation for the Contig Adjacency Graph

To compute a layout for a set of contigs based on multiple reference genomes, we utilize the contig adjacency graph introduced in Section 4.1. The weights in this graph depend on two parameters of the scoring function given in Equation (4.5): The mean μ, and the standard deviation σ of the Gaussian distribution that models the distances of projected contigs.

Table 6.3: Contigs and finished genome that we used to estimate parameters for the contig adjacency graph creation.

Contig Organism	# Contigs (# Repetitive)	Total Length (bp)	N50 Contig Size (bp)
C. pseudotuberculosis FRC41	87 (1)	2,315,337	50032

Finished Genome	Replicon Type	Length (bp)	NCBI Number
C. pseudotuberculosis FRC41	chromosome	2,337,913	NC_014329

In order to find suitable values for these parameters, we acquired another contig set that is explicitly omitted in the evaluation experiments such that the estimation is largely independent of the evaluation contigs. We used contigs of *Corynebacterium pseudotuberculosis*, and again filtered them for a minimum length of 500 bases. Table 6.3 contains more detailed information about the resulting contig set, and also about the complete genome which has recently been finished [95].

With the mentioned genomic sequences, we estimated μ and σ as follows: First, we devised a reference layout of the contigs and created a FASTA file of the regular contigs as described in the previous section. In the next step, the contigs were matched on all genomes of Table 6.2. Then, for each genome, the pairwise distances of projected contigs were calculated. Since we know from the reference layout which contigs are adjacent, we can investigate all distances of projected contigs that belong to truly adjacent contigs. The histogram of Figure 6.2 shows these distances, and we can observe that, besides outliers, most projected contigs have a small distance. Reasons for outliers are mainly rearrangements in the reference genomes, repetitive substrings in the contigs, or unspecific matches occurring by chance.

While the main fraction of the distances follows approximately a Gaussian distribution, the outliers hinder a proper estimation of μ and σ. Consequently, we try to exclude them by discarding all distances larger than the third quartile ($Q_3 = 20,906$ bases) of the data. To also remove outliers on the other side, while accounting for the asymmetry of the histogram, we clip the data symmetrically with respect to the median ($Q_2 = 4,808$ bases) of the distances. Thus, all values smaller than $Q_2 - Q_3 = -16,098$ are discarded. The result of the described outlier reduction is shown as histogram in Figure 6.3.

On the trimmed data, we performed then a maximum-likelihood fitting to a Gaussian distribution. Rounded to integers, the fitting resulted in a mean value of $\mu = 3,000$ and a standard deviation of $\sigma = 6,926$. The corresponding Gaussian function with the estimated parameters is also plotted in Figure 6.3.

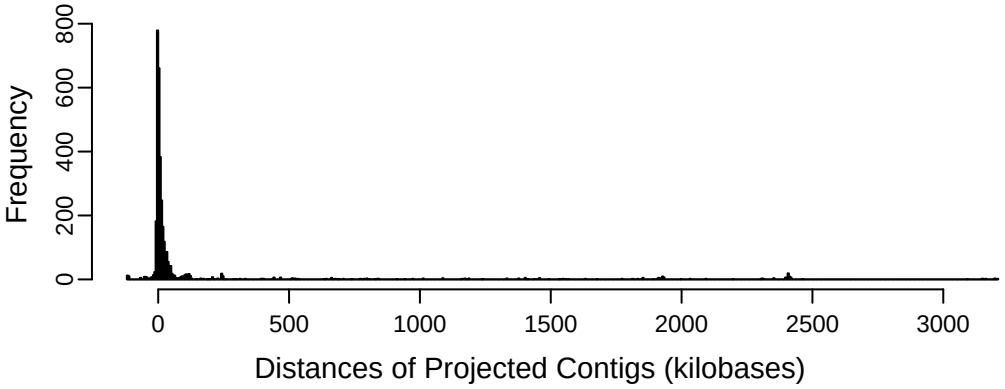

Figure 6.2: Complete histogram over the distances of projected contigs that are adjacent according to the reference layout.

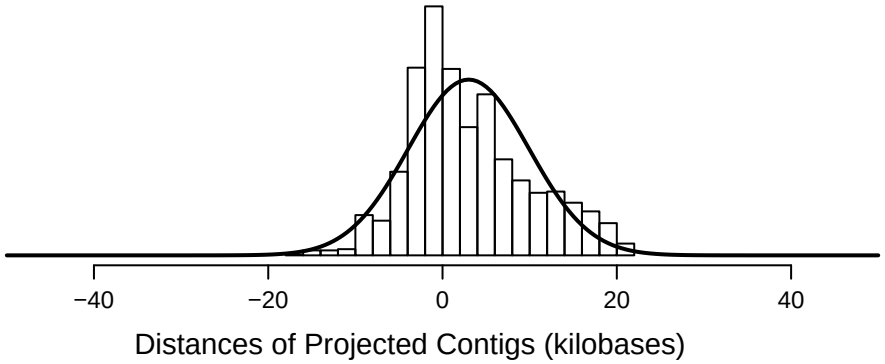

Figure 6.3: Range selected for parameter estimation and resulting Gaussian function with $\mu = 3{,}000$ and $\sigma = 6{,}926$.

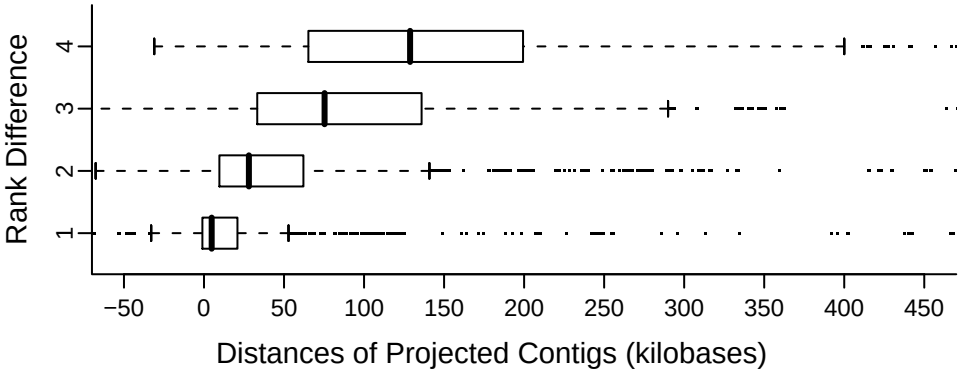

Figure 6.4: Box-plots of the distances of projected contigs grouped by the difference of the rank of the contigs with respect to the reference layout.

Certainly, the choice of μ and σ has an effect on the contig adjacency graph, and therefore also on the edges predicted. The difficulty to find sensible parameters is illustrated in Figure 6.4. It shows box-plots of the distances of projected contigs – not only for truly adjacent contigs that have a difference in their rank in the reference layout by one – but also for contigs that have a rank difference of two, three, and four. The latter correspond to *approximate adjacencies* of contigs that have one, two, or three contigs in between. Note that the box-plot for rank one and the histogram in Figure 6.2 represent the same data, however, the box-plot displays only the reduced range of the distances from -50 to 450 kilobases to emphasize the inter quartile ranges.

An ideal scoring function has the property that rank one distances are separated from those of all other ranks, meaning that immediate neighbors receive high scores and the latter low scores. As can be seen in the box-plots for rank one and two of Figure 6.4, this is only possible to a certain degree on this data, since the inter quartile ranges of both distributions overlap.

In general, the graph creation and the layouting is relatively robust to smaller changes of the parameters. If, however, the width σ is increased too much, then a wider range of distances get high scores. This results in a stronger shadowing effect since for example a part of the rank two distances will also get high scores. On the contrary, a very small value for σ considers only a small range of distances and thus scores only a fraction of the correct distances adequately.

The choice of μ depends on the expectations how big gaps between the contigs are. If, for example, small contigs are removed from a contig set, the gap sizes change, and consequently μ should be increased. However, too large values for μ favor adjacencies between contigs that actually have a small contig in between.

We experimented with the estimated values of $\mu = 3,000$ and $\sigma = 6,926$ a little and found out that they actually yield quite reasonable results. Therefore, we decided to use the derived parameters for all applications of *treecat* in the evaluation experiments in Section 6.2.

6.1.4 Other Software for Contig Layouting

We found several programs in the literature that are able to compute a layout of a set of contigs based on related genomes. In the following, we introduce the programs which we used to compare our layouting performance with. First, the single reference based approaches are described in chronological order, followed by the only other approach to handle multiple references.

Single Reference Based Methods

Projector 2 The web service *Projector 2* [40] maps contig ends onto a template genome using *BLAST* or *BLAT*. Features of *Projector 2* are an optional repeat masking

for contig and template sequences, a visualization of the mapping, and an automated primer design step for gap-closing purposes. Prior to the automated primer design, difficult regions for primer walking are removed. These include sequences with an unbalanced GC content, or repetitive sequences like phage DNA, IS elements or gene duplications.

OSLay The program *OSLay* [79] takes a set of *BLAST* or *nucmer* matches between the contigs and a reference sequence or scaffold, and computes from these a layout of the contigs. To this end, the algorithm minimizes height differences of so-called local diagonal extensions, which are basically matches from the border of a contig to the reference sequence. If the reference is a scaffold, then the program optionally layouts both sets of sequences simultaneously. A resulting layout is visualized and can be imported into a Consed [35] project to aid gap closure.

ABACAS The acronym *ABACAS* [7] stands for "algorithm-based automatic contiguation of assembled sequences". The authors refer by *contiguation* to the process of aligning, ordering, and orienting a set of contigs. These tasks are mainly done with tools of the *MUMmer* package: The aligners *nucmer* or *promer* of this package are used for matching, and then the *delta-filter* and *show-tiling* programs perform the ordering and orienting of the contigs. Before matching, the reference FASTA sequence is checked that it contains only a single chromosome which hinders to use *ABACAS* for multi-chromosomal reference genomes. After contiguation, the Artemis Comparison Tool (*ACT*) [19] can be used to visualize and manually reorder the contigs. Additionally, the *Primer3* [53] program can be run to find suitable primer pairs to close the gaps.

MCM The Mauve Contig Mover (*MCM*) [80] is integrated into the genome alignment and visualization system *Mauve* [24]. *MCM* proposes the relative order of the contigs based on a complete or draft reference genome. To this end, it uses the *Mauve* progressive aligner [25] to identify local collinear blocks (LCBs). The ordering exploits that placing contigs in a correct order will merge LCBs, thus the program strives to minimize their number. The process of matching and ordering is iteratively repeated until the order does not change anymore. After ordering the contigs, the *Mauve* visualization can be used to inspect potential misassemblies, or also rearrangements between reference genome and contigs.

Multiple References

PGA Zhao *et al.* [109] present a method to find a layout for a set of contigs using several related sequences as references. For each reference genome a *fitness matrix* is computed giving distances between the contigs based on their *BLAST* matches.

All matrices are combined into a single fitness matrix to search a (near) optimal path of contig connections with their heuristic *PGA* (pheromone trail-based genetic algorithm). It is noteworthy that the randomized algorithm potentially proposes different adjacencies each time it is run.

6.2 Evaluation on Real Sequencing Data

In the remainder of this chapter, we address the outcome of three different experiments: At first, we evaluate the performance of single reference based contig layouters, each time with the closest genome as reference sequence. Secondly, we compare *PGA* and *treecat* using all reference genomes except for the one to be finished. Finally, we assess the effect of repeats by applying *treecat* and *repcat* on a contig set including repetitive contigs.

In each experiment, some of the introduced programs, as well as our implementations, were run to devise a layout of the contigs. The output was then compared to the corresponding reference layout. As quality measurement, we take the following four values that we also state in the result tables of each experiment:

TP We count all proposed connections that also occur in the reference layout as *true positive* predictions.

FP Predictions that do not appear in the reference layout are *false positives*.

TPR From the values of TP and FP, we calculate the *true positive rate* (also *sensitivity*):

$$\mathrm{TPR} := \frac{\mathrm{TP}}{\mathrm{P}} \ ,$$

with P being the total number of achievable correct connections.

PPV The *positive predictive value* (also *precision*) tells the percentage of all predictions that were actually correct:

$$\mathrm{PPV} := \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}}$$

Note that when counting the true or false positives we do not consider the orientation of the contigs since *PGA* does not always provide this information. In consequence, it is possible that we count too many true positives. However, we think that this evaluation procedure is fair since all programs are treated equally and we presume that no program gains a particular benefit.

Besides the true and false positives of a predicted layout, we also measured the running time of the different approaches. All experiments were performed on a Sun-Fire-V440 Solaris server with four sparcv9 processors operating at 1,593 MHz. During the experiments, we were the only user logged in such that effects by other processes on the running time are minimized. Further, we measured the *user time*

of the processes which excludes the time for system calls like for example I/O operations. Obviously, the times of the web service *Projector 2* are not comparable.

6.2.1 Single Reference Based Ordering

This first experiment was designed to compare the layouting of other single reference based approaches with *r2cat*. To this end, we applied the single reference programs of Section 6.1.4 on the contig sets using the closest phylogenetic neighbor as reference sequence. According to the phylogenetic tree, shown in Figure 6.1, this is *C. glutamicum* for the contigs of *C. aurimucosum*, and *C. jeikeium* for the other two contig sets.

As contig sets, we used the FASTA files containing the renamed regular contigs. For any adjacency estimated by a program we could thus determine whether it is a true positive or a false positive prediction. Besides the obvious adjacencies of c_i and c_{i+1}, we considered the first and the last contig of each set as adjacent since the genomes are circular. The number of possible true positive adjacencies (P) is thus 55 for *C. aurimucosum*, 5 for *C. kroppenstedtii*, and 54 for *C. urealyticum*.

We describe now first, how the different programs were used and how the estimated adjacencies were extracted, and then discuss the results given in Table 6.4 on page 82. Except otherwise stated, we used the default parameters of each program.

- The *Projector 2* results were generated using its web service. There, we first selected the appropriate reference genome for each dataset from a given list. For the *C. jeikeium* genome either the chromosome, or the plasmid pKW4 was selectable, but not both, so that we used the chromosome without the plasmid. The next step was to upload the contigs to the web server. On the following parameter page, we kept the default settings, except that we selected to download a zip file containing the generated files. As default, the matching was performed by running *BLAT* on the server. The running times for *Projector 2* were estimated from the provided start and stop times of the contig ordering jobs. According to the website, the server runs on a quad core AMD Opteron with 2.0 GHz, thus, the running times on the server are not directly comparable to the other programs that we ran locally. We give the times nevertheless to provide a rough impression. The adjacencies of the contigs were extracted from the `mapped_sort_1.csv` files found in the downloaded zip files.

- To generate the *OSLay* results, we first had to match the contigs onto the corresponding reference genome since, unlike for all other programs, this is not done by the program. The matches that we generated with *nucmer* (v. 3.07), as well as the original FASTA files, could then be provided to *OSLay* which computed an optimal syntenic layout of the contigs. To this end, we used the standard parameters of the implementation and selected to export the syntenic layout. The adjacencies were extracted from the `supercontigsList.x.txt`

files. The running time of *OSLay* in the following results table is only the time for matching with *nucmer*. The layouting time could not reasonably be measured since the necessary files have to be provided manually in the graphical user interface. As an estimation, the layouting takes between two to four additional seconds.

- The program *ABACAS* refused to use reference genomes with more than one sequence. Thus, we provided the *C. jeikeium* reference without its plasmid pKW4. For the matching, the user can specify that either *nucmer* or *promer* is called. After a rapid matching with *nucmer*, we discovered that the program had no results for two of the datasets, and only false positive predictions for the third. The explanation is that *ABACAS* relies on the *show-tiling* program of the *MUMmer* package which produced an empty file for the two datasets when using the *nucmer* matches. Consequently, we decided to match also with the much slower *promer* which yielded better results. The adjacencies were in both cases extracted from the generated `<fasta>.tab` files, where `<fasta>` is the filename of the original contig file.

- The *Mauve* extension *MCM* iteratively matches and reorders the contigs. For each iteration, a new folder is created that contains the sequences and the estimated layout. We used the batch option to start *MCM* from the command line. The matching was performed with the *progressiveMauve* program, of the *Mauve* package version 2.3.1, which we compiled for the Solaris operating system. The resulting adjacencies were extracted from the folder created in the last iteration. There, the file `<base>_contigs.tab` contains the proposed adjacencies of the contigs, with `<base>` being the prefix of the contigs' FASTA filename up to the first dot.

- Finally, we used *r2cat* as described in Section 3.3. After sorting the contigs, the result was written to a file.

Table 6.4 gives the results of applying the different programs on the three datasets. It shows that the pioneering approaches *Projector 2* and *OSLay*, which mainly use matches at the end of the contigs, clearly find less true positive adjacencies than newer algorithms considering all matches. The more recent programs *ABACAS* (with *promer* matches) and *MCM* predict more correct adjacencies but also more false positive connections. The running times of these programs are dominated by the matching: For *ABACAS*, the matching with *promer* takes several *days*. In view of the few seconds of some other programs this is an unacceptable amount of time. Also the hours that *MCM* needs for the iterative matching are not desirable.

Our implementation *r2cat* shows on these datasets a reasonable performance. Matching and simple ordering are performed in seconds and the layout result contains high numbers of correct adjacencies and a moderate number of false positives.

Table 6.4: Results of the single reference based layouting programs using the phylogenetically closest genome as reference sequence. The running times for *OSLay* contain only the matching, those of *Projector 2* are not comparable to the other times since they were measured on a different CPU.

Program	TP	FP	TPR	PPV	Time (s)
C. aurimucosum contigs with *C. glutamicum* reference					
Projector 2	7	14	0.13	0.33	(26)
OSLay	0	1	0.00	0.00	16
ABACAS with *nucmer*	0	0	0	*undef.*	32
ABACAS with *promer*	26	12	0.47	0.68	92,769
MCM	22	33	0.40	0.40	9,526
r2cat	33	11	0.60	0.75	13
C. kroppenstedtii contigs with *C. jeikeium* reference					
Projector 2	3	2	0.60	0.60	(25)
OSLay	0	0	0.00	*undef.*	12
ABACAS with *nucmer*	0	0	0	*undef.*	24
ABACAS with *promer*	3	1	0.60	0.75	79,415
MCM	2	3	0.40	0.40	684
r2cat	2	3	0.40	0.40	8
C. urealyticum contigs with *C. jeikeium* reference					
Projector 2	16	15	0.30	0.52	(24)
OSLay	6	2	0.11	0.75	23
ABACAS with *nucmer*	0	7	0.00	0.00	44
ABACAS with *promer*	23	15	0.43	0.61	180,047
MCM	25	29	0.46	0.46	6,327
r2cat	26	22	0.48	0.54	11

Let us take a closer look at the *C. kroppenstedtii* contigs. Figure 6.5 shows the synteny to the closest reference genome with the contigs ordered and oriented according to the reference layout. With the simple contig mapping of *r2cat*, the contig c03 is placed between c00 and c01, as indicated in the figure. This obstructs three true positive adjacencies.

The example shows that a single misplaced contig can produce up to three false positive connections. The measurement of false positives is thus very stringent in this evaluation, and one might argue that it is better to know about approximate adjacencies in which maybe a smaller contig can be missing between two larger ones. Especially in the next experiment, the programs provide more false positives, and often these are such approximate adjacencies.

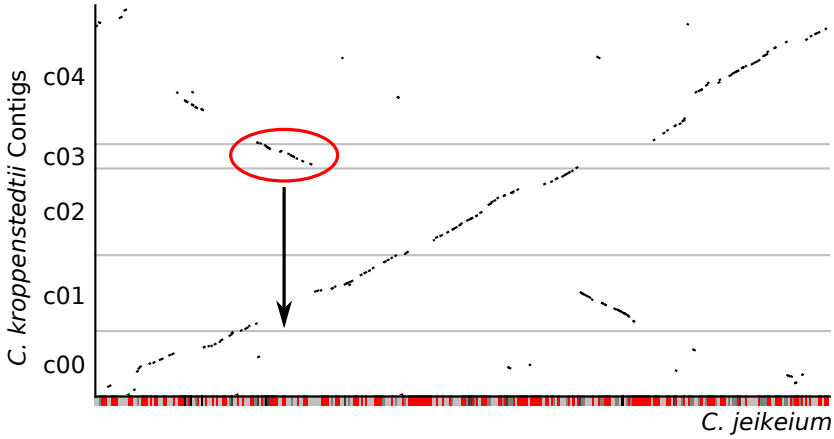

Figure 6.5: Synteny plot of the contigs of *C. kroppenstedtii* matched on its closest reference genome *C. jeikeium*. The contigs are ordered and oriented according to the reference layout.

6.2.2 Multiple Reference Based Layouting

In this experiment, we want to investigate whether the use of multiple reference genomes can improve the adjacency prediction. Several reference genomes most likely increase the information that can be collected from matches of the contigs. This information can, however, also be conflicting and thus lead to false positive predictions. Approaches that are not restricted to output a single linear layout, as for example *treecat* with its layout graph, will even accumulate such false positives. Nevertheless, we believe that showing several likely possibilities is better than to output a single linear layout. The latter can be misinterpreted as containing the only correct solution. Giving alternatives in conflicting cases expresses better that a result should be handled with caution.

In the following, we compare the multiple reference based layouting programs *PGA* and *treecat* by applying them on the contig sets with all genomes of Table 6.2 as references. Of course, whenever a genome is to be reconstructed from its set of contigs, this genome is removed from the dataset of reference genomes. Like in the previous experiment, we used the contig sets without repetitive contigs. We first describe how both programs were used and which parameters have been chosen, and then present and discuss the results of Table 6.5 on page 85.

Although we already introduced *PGA* briefly, we want to provide some further details. This helps to better understand some aspects of the results and of the application of *PGA*.

A layout computation with *PGA* can be divided into two parts: In the first part, a *fitness matrix* is created which is comparable to our contig adjacency graph. To this end, the set of contigs is matched onto each reference genome using *BLAST*. The

corresponding *Perl* script discards contigs smaller than 3.5 kbp before matching and masks repetitive matches. Additionally, too short, as well as overlapping matches are filtered. Based on the remaining matches, the fitness matrix is created such that it contains pairwise distances between the contigs. The distances range from 0 to 20, where a value close to 0 indicates that the contigs are potential neighbors. In the last step of this part, the matrices derived from all reference genomes are combined into a single matrix.

In the second part of applying *PGA*, a genetic algorithm is used to calculate a linear layout of the contigs that aims to minimize the distances given in the combined fitness matrix. For the application of the algorithm, we used the default parameters that are also given in the publication [109]: We set the "pheromone trail persistence" $\rho = 0.8$, the "relative importance of [the] pheromone trail and the visibility" $\beta = 3$, the "probability for pseudo-random-proportional selection" $q_0 = 0.8$, and the maximal number of iterations to $10,000$.

The underlying genetic algorithm is a randomized algorithm such that the layouts of each computation may differ. This is probably the reason why the C++ implementation of the genetic algorithm computes *five* linear layouts each time it is run, which are then combined into a single result. In a combined result, for each predicted adjacency the percentage is given how often it occurred in one of the five layouts. Despite of this attempt to produce more robust results, the combined layouts are still subject to variation. That is why we applied the layouting part of *PGA* 20 times for each contig set, resulting in 100 computed layouts. In the comparison in Table 6.5, we give the mean values of the 20 runs, and additionally the *best* result in parentheses. As best, we define that combined result yielding the highest true positive rate. In case of several equally good layouts, we break ties using the positive predictive value.

Next, we applied *treecat* on the data as described in Section 5.1.2. To compute the adjacency support values, we used the phylogenetic tree of the Corynebacteria, and set $\mu = 3,000$ and $\sigma = 6,926$, as discussed in the section on parameter estimation.

The results of both programs are listed in Table 6.5. The comparison shows that our *treecat* achieves equal or better results than *PGA* while at the same time being much faster. The running times given for the matching mainly compare the corresponding routines in *r2cat* and *BLAST*, as already done in Section 3.3.1. However, the running times for the layouting impressively demonstrate that *treecat* is 200-fold faster than *PGA*, on average for the three datasets.

Another advantage of *treecat* is that it has only two parameters that influence the layouting, and nevertheless produces quite robust results. In contrast, *PGA* depends, besides the already mentioned four parameters, on further 'default values': For instance, an initial population size, default mutation and crossover probabilities, as well as min and max values for the population fitness are stated in the publication.

Table 6.5: Results of applying *PGA* and *treecat* to layout the contig sets with the help of the remaining genomes of Table 6.2 as references. *PGA*s results are averaged over 20 runs, its best run (highest TPR with best PPV) is given in parentheses.

					Times (s)	
Program	TP	FP	TPR	PPV	Matching	Layouting
			C. aurimucosum contigs			
PGA	28.1 (31)	76.0 (79)	0.51 (0.56)	0.27 (0.28)	214.1	92.3
treecat	31	36	0.56	0.46	34.6	0.3
			C. kroppenstedtii contigs			
PGA	3.0 (3)	2.0 (2)	0.60 (0.60)	0.60 (0.60)	98.0	15.6
treecat	3	4	0.60	0.43	30.5	0.2
			C. urealyticum contigs			
PGA	15.5 (18)	75.9 (75)	0.29 (0.33)	0.17 (0.19)	221.9	80.9
treecat	28	40	0.52	0.41	34.2	0.3

We now examine the results for of all three contig sets in more detail and provide some additional observations that cannot be derived from the table alone.

Corynebacterium aurimucosum For this contig set, *PGA* found in the best run the same number of 31 true positive connections as *treecat*. But looking at the best result of *PGA* alone might distort the general impression of its layouting quality. The box-plots in Figure 6.6 clarify that this result is not typical for *PGA*, since often worse results are predicted.

Staying at this dataset, a comparison of *treecat* with *r2cat* shows that using more reference genomes does not automatically provide better results. While *r2cat* pre-

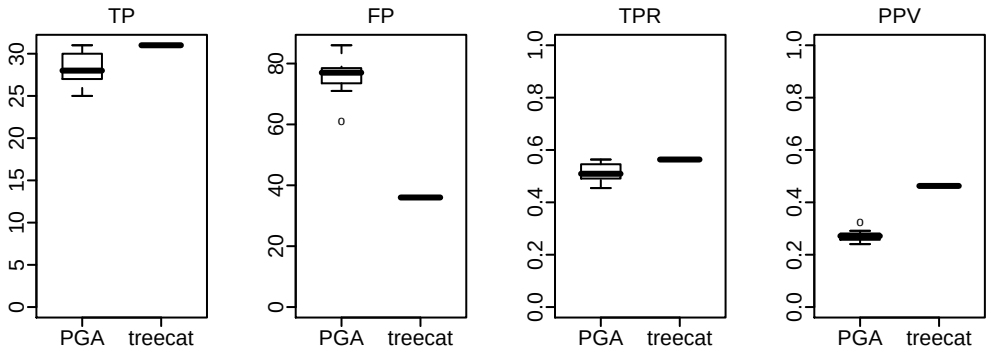

Figure 6.6: Box-plots of the varying results of 20 *PGA* runs for the *C. aurimucosum* dataset. The steady result of *treecat* is shown by a plain bar.

dicts 33 correct adjacencies, *treecat* finds two less, although using the information of five additional reference genomes. Fourteen of *treecat*'s false positive predictions might nevertheless be helpful in the gap closing process, since they are approximate adjacencies having a rank difference of one in the reference layout.

When *treecat* is run with *C. glutamicum* as single reference genome, like in the experiment with *r2cat*, it yields 35 true positives and 25 false positive predictions. This shows that additional genomes can introduce high weight edges to the contig adjacency graph that hinder the extraction of true adjacencies in the layouting phase.

The advice is therefore to first assess the relatedness of the genomes to be used for layouting, for example with synteny plots. Then, the most promising ones should be chosen to be the basis of the contig adjacency graph, following the guideline "quality before quantity". Our *htscat* suite supports such a workflow by allowing to investigate the synteny plots of the contigs and to select those references that are most related to build the contig adjacency graph.

Corynebacterium kroppenstedtii This contig set consists only of five regular contigs, but it seems to be hard to layout. None of the contig ordering programs predicted more than three of the five possible true positive adjacencies. To investigate this further, we take a look at the contig adjacency matrix W from page 46. The complete matrix devised by *treecat* for the *C. kroppenstedtii* contigs is shown in Figure 6.7. Each row (or column) contains the integer rounded adjacency support values for a certain contig connector. The adjacencies of the reference layout

$$\underleftarrow{\text{c00}}\ \underrightarrow{\text{c01}}\ \underrightarrow{\text{c02}}\ \underleftarrow{\text{c03}}\ \underleftarrow{\text{c04}}$$

are underlined in the matrix. In the circular genome, c04 and c00 are also adjacent. As a reminder, the matrix is symmetric such that each adjacency support appears

$$W = \begin{array}{c} \\ r_{00} \\ r_{01} \\ r_{02} \\ r_{03} \\ r_{04} \\ l_{00} \\ l_{01} \\ l_{02} \\ l_{03} \\ l_{04} \end{array} \begin{array}{c} \begin{array}{cccccccccc} r_{00} & r_{01} & r_{02} & r_{03} & r_{04} & l_{00} & l_{01} & l_{02} & l_{03} & l_{04} \end{array} \\ \left(\begin{array}{cccccccccc} 0 & 0 & 0 & 0 & 3 & 0 & 2 & 0 & 0 & \underline{\mathbf{6}} \\ 0 & 0 & 23 & 0 & 0 & 2 & 0 & \underline{\mathbf{294}} & 0 & 0 \\ 0 & 23 & 0 & \underline{\mathbf{20}} & \mathbf{377} & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & \underline{\mathbf{20}} & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 3 & 0 & \mathbf{377} & 0 & 0 & 4 & \mathbf{394} & 0 & \underline{37} & 0 \\ 0 & 2 & 1 & 1 & 4 & 0 & \underline{42} & 0 & \mathbf{43} & 8 \\ 2 & 0 & 0 & 0 & \mathbf{394} & \underline{42} & 0 & 0 & 2 & \mathbf{15} \\ 0 & \underline{\mathbf{294}} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \underline{37} & \mathbf{43} & 2 & 0 & 0 & 0 \\ \underline{\mathbf{6}} & 0 & 0 & 0 & 0 & 8 & \mathbf{15} & 0 & 0 & 0 \end{array} \right) \end{array}$$

Figure 6.7: Symmetrical weight matrix of Equation (4.2) filled with integer rounded weights computed by *treecat* for the *C. kroppenstedtii* contigs. True adjacencies are underlined, edges predicted by *treecat* are printed in bold face.

twice. The adjacencies extracted by *treecat* in its layouting process are printed in bold face in the matrix. Consequently, all bold faced entries that are underlined are true positive predictions, and those not underlined are false positives.

In the given matrix, we can observe that the highest adjacency support (394) for the connection of the left connector of `c01` to the right connector of `c04` actually does not belong to a correct adjacency. The high weight is caused by an inversion that is also documented in the synteny plot in Figure 6.5 for the case of the closest reference genome.

This example shows that in general the weights in the contig adjacency graph just collect the information given by the matches. If the matches are biased or unreliable, then the weights are so, too. Consequently, all results derived with the help of reference genomes have to be judged in the context of their reliability.

Corynebacterium urealyticum On this contig set, our approach *treecat* found the most true positive adjacencies compared to all other approaches. This success is slightly diminished by the 40 false positive connections that are also contained in the resulting layout graph. Nevertheless, this number is much lower than the 75.9 false positives that *PGA* predicted on average.

In the following, we want to visually compare the layouts generated by *PGA* and *treecat*. To this end, we exemplarily show the resulting layout graphs computed by both programs for this dataset in Figures 6.8 and 6.9. The graphs are visualized with the program *neato* of the graph visualization package Graphviz, as described in Section 5.2.2. For *PGA* we used the best layout as defined above.

The nodes in both graphs are labeled with the renamed contig names that indicate the correct adjacencies. An optimal layout would therefore be a single circle showing all contigs in order $\texttt{c00}, \texttt{c01}, \ldots, \texttt{c54}$. In the *treecat* graph in Figure 6.9, a node label additionally gives the size of that contig. We have drawn in gray all nodes of contigs smaller than $3,500$ bases, as well as the incident edges, to improve the comparability of both graphs, since *PGA* filters all such contigs before matching and thus cannot contain them in the layout.

It is arguable whether the missing contigs might be a disadvantage for *PGA* or not; however, we like to note that neither the size of the contigs to be removed can be specified, nor the removal can be switched off completely in their software. Further, in the *treecat* layout, the small contigs contributed only to four true positive connections but at the same time introduced ten false positives.

The edge weights in *PGA*'s layout graph in Figure 6.8 tell in how many of the five layouts this connection was proposed. To increase the readability of this graph, we draw all edges in gray that belong to a connection occurring only once. The edge labels in our *treecat* graph give the relative support, as defined in Section 4.1.1. A value close to 100% indicates that this is the only relevant adjacency of the incident contig connector.

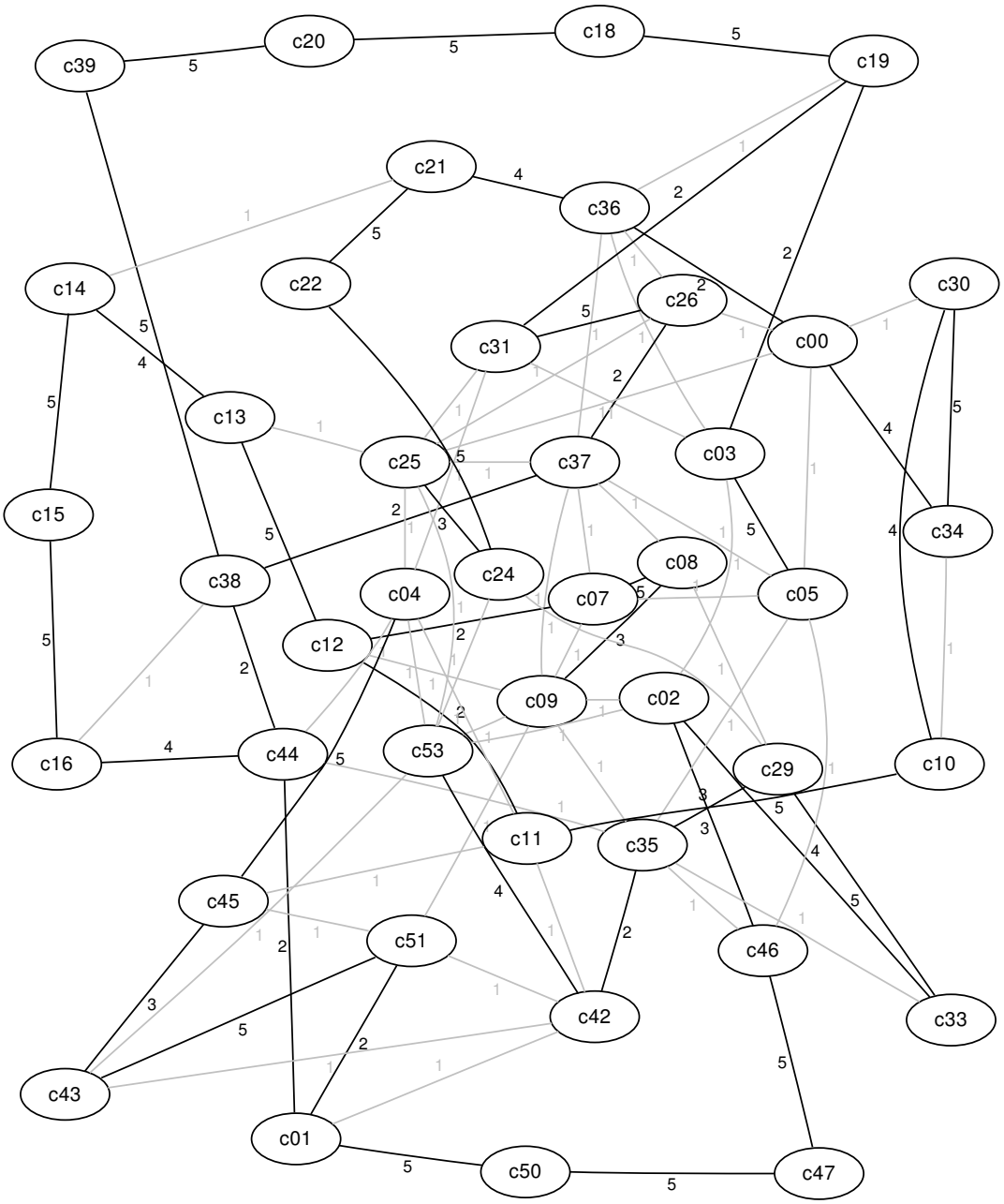

Figure 6.8: The best layout (TP 18, FP 75, TPR 0.33, and PPV 0.19) that *PGA* generated for the *C. urealyticum* contigs in 20 runs when using all other genomes as reference sequences. The contig nodes are numbered according to the reference layout. The edge labels tell how often an adjacency occurred in one of the five layouts. To improve the readability, all 50 edges occurring only in a single layout are drawn in gray.

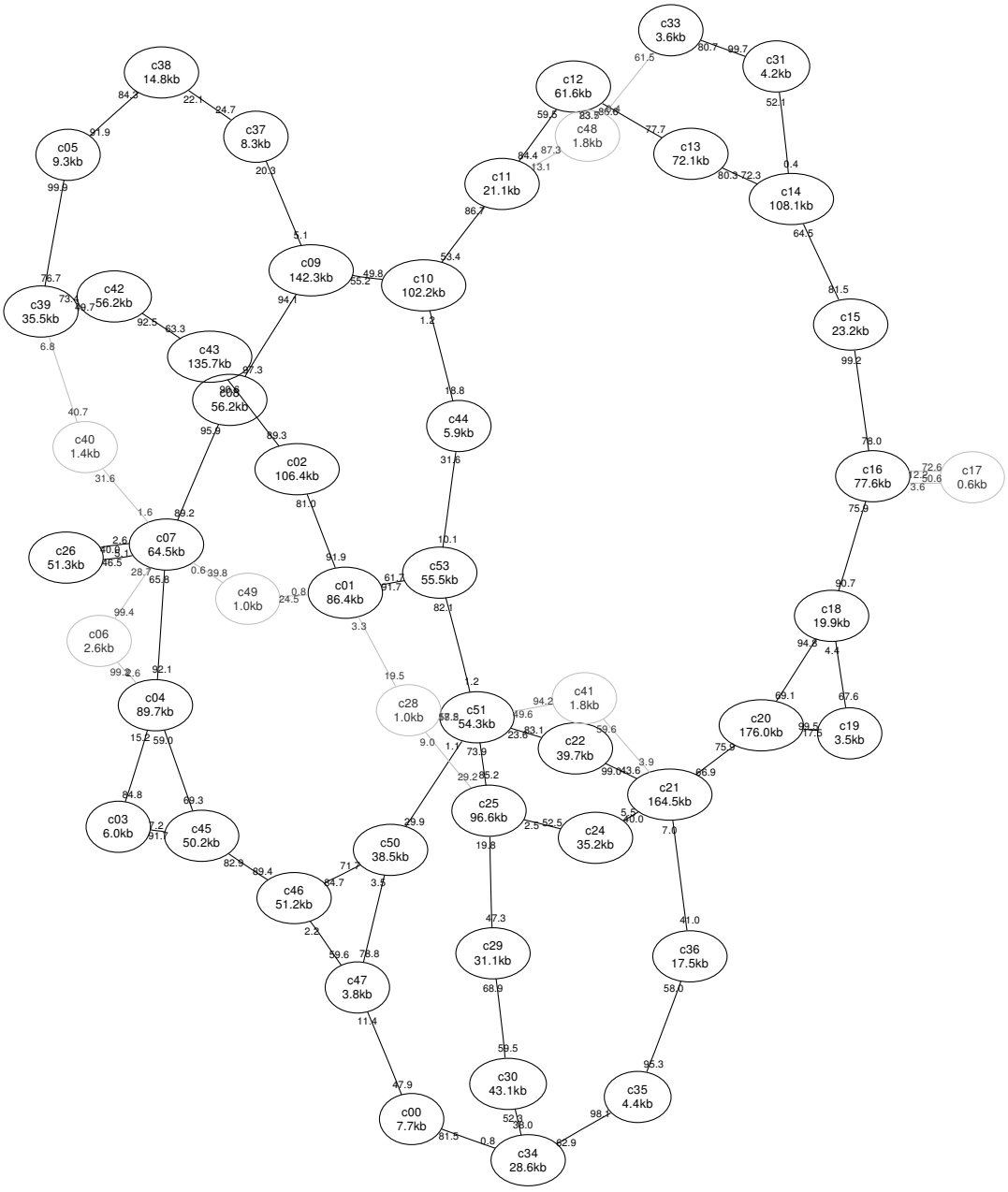

Figure 6.9: *C. urealyticum* contig adjacencies predicted by *treecat* when using all other genomes as reference sequences. The contig nodes are numbered according to the reference layout. Contigs smaller than 3.5 kbp have gray nodes. The edge labels show the relative support with respect to the contig connector nearby.

In a comparison of the displayed graphs, the *treecat* result looks less cluttered and it seems that our approach is able to more robustly interpret the information given by the matches to the reference genomes.

Further investigation of the *treecat* layout graph in Figure 6.9 reveals that for the contigs c07 to c22 an almost unique path can be observed which orders most of the inner contigs correctly. However, our graph contains also connections like from c02 to c43 that have a high (relative) support but are not correct. The high support is due to the big inversion that can be observed in the synteny plot in Figure 3.7 (a) on page 41. There, the reference genome is *C. jeikeium* which is the next phylogenetic neighbor to *C. urealyticum* and thus has a high influence on our result.

The results show that using several reference genomes can be advantageous, but this is not necessarily the case. In particular, we found an example where the use of a single reference instead of several would improve the results of *treecat*. In general, the true positive rate of *treecat* is among the best compared to all other approaches. The idea of *treecat* to display also conflicting information in a layout graph, however, comes with the disadvantage of producing more false positives than approaches predicting a single linear layout. Yet, in contrast to *PGA* the resulting positive predictive value is mostly better. Additionally, we would like to emphasize that several false positives of *treecat* are approximate adjacencies. Such a prediction can be considered helpful for the finishing process in practice, despite of being a false positive in theory.

6.2.3 Layouting Repetitive Contigs

The aim in the last section of this evaluation is to study how good repetitive contigs can be integrated into a contig layout. In the previous experiments these were excluded since none of the involved programs was intended to handle repetitive contigs properly. Some approaches even actively try to remove such contigs.

Here, we exemplify the integration of repetitive contigs on the *C. urealyticum* dataset. In the following experiment, we thus employed all 69 contigs as listed in Table 6.1, including the 15 repetitive contigs. We want to remind that we have prefixed all regular contigs with a 'c' and numbered them in their true order. The repetitive contigs are prefixed with an 'r' and are numbered arbitrarily.

To know where instances of the repetitive contigs can be placed correctly, we manually inspected their matches on the finished *C. urealyticum* genome with *r2cat*. For each pair of adjacent regular contigs we noted which repetitive contigs have an occurrence between them. The list contains for example that the repetitive contig r05 can be placed between the regular contigs c19 and c20 but as well between c50 and c51. We observed that some repetitive contigs occur several times in tandem between the same pair of regular contigs.

We applied two programs to find a layout of the contigs. The first program was *treecat* that is among the best programs in the previous experiments, and the second

was *repcat* which was created to correctly place the repetitive contigs between the regular contigs allowing them to appear more than once.

Both programs were run two times using a single genome as reference: The already finished genome of *C. urealyticum* as a *perfect reference*, and the genome of *C. jeikeium* as a more realistic reference that is still very closely related. Details to these genomes are given in Table 6.2.

In the case of *treecat*, we used the same parameters as in the previous experiment. For the application of *repcat*, however, we tweaked the parameters of the contig adjacency graph creation to better match the employed references: We experimentally adjusted μ to $2,000$ and σ to $2,000$ as well. This decrease of both parameters yielded more reasonable results and can be justified by the high relatedness to the reference genomes. Closer references have less insertions and deletions and, additionally, the repetitive contigs are included this time which also decreases the expected gap size. Further, we set two thresholds for *repcat* that were introduced in Section 4.4.2: We set the additional edges threshold $\tau_1 = 90\%$ and the threshold for incorporating repetitive contigs to $\tau_2 = 0.1\%$.

The manually annotated list of repetitive contigs between regular contigs serves as reference layout in these experiments. As motivated in Section 4.4.2, the interesting connections are those from a repetitive contig to a regular contig. Therefore, we extracted those connections from the output of both programs and compared them with our manually annotated list. If a repetitive contig is present in between two adjacent regular contigs, we count the connections to the corresponding regular contigs as true positive. If such an adjacency is not given in our list, we count this as a false positive prediction. Consider the following list where all regular contigs are printed in bold face for a better distinction:

$$\ldots \xrightarrow{} \textbf{c27} \xrightarrow{} \textbf{c28} \xrightarrow{} \text{r09} \xrightarrow{} \textbf{c29} \xrightarrow{} \text{r03} \xrightarrow{} \text{r13} \xrightarrow{} \text{r12} \xrightarrow{} \textbf{c30} \xrightarrow{} \textbf{c31} \xrightarrow{} \textbf{c32} \xrightarrow{} \ldots$$

In this example, a predicted connection from r09 to c28 or c29 is naturally counted as true positive. Also, we count a connection from r13 to c29 or c30 as true positive since it is present in between the regular contigs, although not being directly adjacent. A predicted connection from r13 to c31, however, is counted as false positive because the repetitive contig is not found somewhere next to c31.

We ran both programs on the datasets and counted the true and false positive predictions as described above. Our manually annotated list revealed that the 15 repetitive contigs occurred in 79 instances on the genome. For each occurrence, two true positive connections could be predicted, so the sum of all positive predictions (P) is 158. This number is needed to calculate the true positive rate which is given in Table 6.6, together with the other layout quality measurements.

When comparing *treecat* and *repcat*, it has to be noted that the latter was specifically designed to include repetitive contigs several times into a layout. While *treecat* will stop to add edges if both connectors of a repetitive contig have been considered,

Table 6.6: Results for layouting the repetitive contigs of the *C. urealyticum* dataset. Unlike the previous result tables, not the adjacencies of regular contigs were counted, but the proper placement of a repetitive contig next to a regular one.

	Perfect Reference				*C. jeikeium* Reference			
Program	TP	FP	TPR	PPV	TP	FP	TPR	PPV
treecat	29	1	0.18	0.97	23	7	0.15	0.77
repcat	139	8	0.88	0.95	61	50	0.39	0.55

repcat can predict two true positive connections for many occurrences of a repetitive contig, stopping when the relative support becomes less than τ_2. It would be desirable to verify the number of included repetitive contigs, for example with the read coverage information discussed in Section 4.4.2, however this information is not always given so that we are not using it here. Nevertheless, *repcat* has a clear advantage in this experiment.

As expected, *repcat* recovers more repeat occurrences than *treecat*. With the perfect reference, almost 90% of the possible connections are found. Most missing connections are due to repetitive contigs that appear several times in a single gap. Here, *repcat* only predicts a single occurrence and thus misses possible true positives.

While in the result table only connections from repetitive to regular contigs were considered, we show the complete layout predicted by *repcat*, for the case of the perfect reference, in Figure 6.10. This graph is comparable to the layout graphs of Figures 6.8 and 6.9, though it also contains instances of repetitive contigs which are depicted by rectangular nodes. A repetitive contig is further labeled with an occurrence count in parentheses. The shown graph illustrates that the ideas of the *repcat* algorithm in principle do work. It is observable how the regular contigs form a circle and the repetitive contigs are attached in between.

However, it is apparent in Table 6.6 that using the more realistic *C. jeikeium* reference decreases the true positive rate, as well as the positive predictive value. The tendency is observed for both programs although *treecat* is affected more drastically.

At this point, we need to come back to the assumption stated in Section 4.4.2. The *repcat* algorithm assumes that repeating regions are conserved between closely related species. It might be true that such regions in one genome are usually also repetitive on a related genome. Our results, however, indicate that the placement of these regions in different genomes is very fragile. The idea of *repcat* seems to work only with the perfect reference that holds the very stringent assumption of conserved repeat patterns.

Frankly, we have to advise against using a related genome as a reference to layout repetitive contigs. This outcome is rather unfortunate since repetitive regions are one of the biggest obstacles on the way to a finished genome. Fortunately, there are other methods to handle repeats that do not rely on reference genomes. For

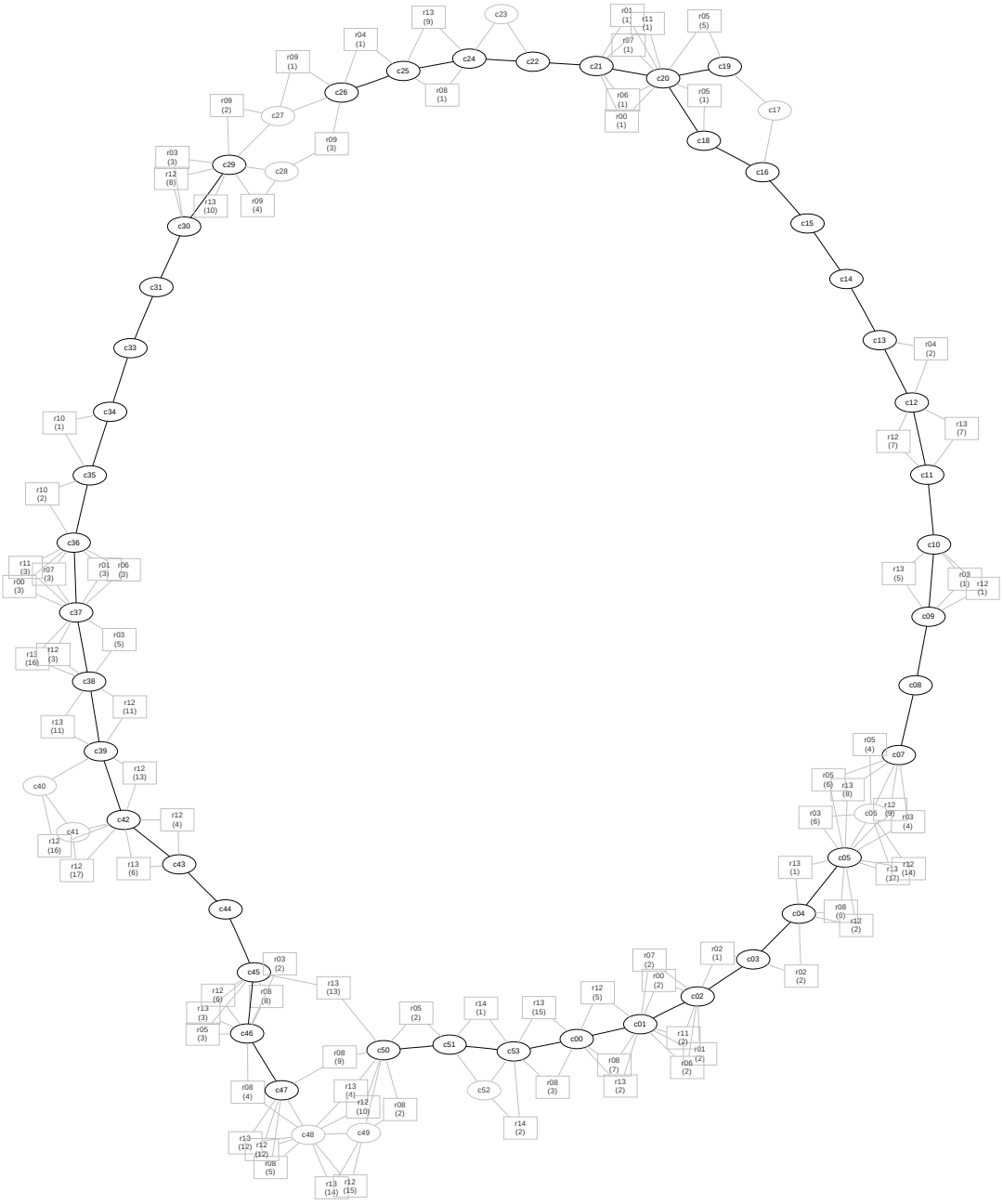

Figure 6.10: *C. urealyticum* contig connections generated by *repcat* using the finished genome as reference. Contigs smaller than 3.5 kbp are drawn in gray. Repetitive contigs, depicted with rectangular nodes, can appear more than once. Their occurrence count is given in parentheses.

example, Wetzel *et al.* [103] recently proposed a two-tiered approach where an initial sequencing and assembly is subsequently augmented with mate-pairs that are fine-tuned to the repeat structure of the genome.

Reference based layouting of contigs has been improved over the years. First approaches used only matches of the contig ends on a reference genome to devised a relative order of them. More recent approaches, including our *r2cat*, use all matches to a reference, and our evaluation shows that they provide better results compared to the initial approaches. Equally important, the single reference layouting approaches differ in their running times. In contrast to several hours that other approaches need to compute a reasonable layout, certainly *r2cat* has with a few seconds a top rank in this field.

If only few contigs match on a reference genome due to a more distant relationship, it can be advantageous to use several of them, if this is possible. Contigs not matching on one reference might match on another genome that is provided. In a comparison to the single other available approach that handles multiple reference genomes, *treecat* shows a convincing performance: Time demand for layouting is superior, and the layout quality is at least equal, often even better than *PGA*'s.

The last experiment of this chapter shows the limitations of reference based layouting. Repetitive contigs are problematic since they should be placed in several instances into a proper layout. Although repetitive contigs often also occur repetitively on a related genome, the adjacencies of the repetitive regions do not seem to be conserved too well between species, and thus a reference based placement is unlikely to be successful.

As a conclusion of this whole evaluation, we would recommend a combination of *r2cat* and *treecat*. If a closely related reference is available – and the relatedness can easily be assessed with the synteny plots of *r2cat* – then the simple mapping produces reasonable results. If the synteny plots indicate that the references are more distantly related, then a layout graph can be computed which shows the most promising adjacencies that were collectable from the data. Our suite *htscat* contains all necessary components supporting such a workflow.

Chapter 7

Summary and Outlook

In this thesis, we introduced bioinformatic approaches to aid the process of genome finishing. One result of our efforts is the program *r2cat* – introduced in Chapter 3 – which can be used to quickly match a set of contigs onto a related reference genome, and to inspect the matches in a synteny plot visualization. The matches can then be used to arrange the contigs in a linear layout according to the reference genome. For adjacent contigs, primer pairs can be designed to facilitate the amplification and sequencing of the gaps in between. Through its availability and its ease of use, this tool has already been engaged in several sequencing projects conducted at the Center for Biotechnology (CeBiTec) of Bielefeld University.

While *r2cat* is mainly a software contribution, the program *treecat* also contains novel ideas and concepts: In Chapter 4, we devised an approach to simultaneously include the information given by multiple reference genomes. To this end, we use an advanced scoring function to collect hints on which of the contigs are adjacent based on matches to the reference genomes. The scoring function can even utilize evolutionary distances of the reference genomes given by a phylogenetic tree. All information is gathered in a *contig adjacency graph* which represents all possible adjacencies of the contigs. After building the contig adjacency graph, we use a fast heuristic to extract the most promising edges into a layout graph. This is a more flexible and powerful output compared to the linear layout that most other programs provide. In an evaluation with real sequencing data, we found out that *treecat* provides equivalent or better results while being much faster than other related approaches from the literature.

The concepts of *treecat* were extended to also handle repetitive contigs which are ideally included into a proper layout as often as they occur in the genome. Although promising in theory, we discovered in the evaluation of the resulting program *repcat* that the repetitive parts of a genome do not behave as expected for realistic data. Thus, unfortunately, a desired reference based integration of repetitive contigs appears not to be reliable.

All our implementations are bundled in the open source software project *htscat* that we initiated. As described in Chapter 5, it provides an extensible modular framework which currently combines the features of *r2cat* and *treecat*. We hope that it will become valuable in many sequencing projects.

To our opinion, a well-balanced collaboration between biology and bioinformatics is the engine that drives the research in our field. Due to different scientific 'languages' that computer scientists and biologists use, this can become complicated. Yet, many successful joint projects show that the effort is worth it. Also our project would not have been fruitful without constant feedback and suggestions, and we are grateful for the possibility of these collaborations within the CeBiTec.

The goal in our research was to help in the sequencing of prokaryotic genomes. Though, this help was never meant to be a replacement of biological expertise or knowledge. On the contrary, we want to generally warn against trusting automated 'analyses' of bioinformatic software without restrictions. Hence, also a layout computed by one of our programs should not be seen as ground truth, but more as a support for a proper finishing of the genome.

Any predicted layout has to be critically questioned and it should be clear that the layout graph only reflects the information given by the data. If different reference genomes provide conflicting information, then a devised layout graph most likely shows misleading adjacencies. In our evaluation, particularly inversions happened to be a major reason for introducing conflicts. A possible improvement of our methods would therefore comprise a detection and possibly also an untangling of these inversions, or of rearrangement events in general.

To investigate conflicting information, it might be desirable to know which of the reference genomes contributed to which proposed adjacencies. One idea is thus to create for each reference genome an own contig adjacency graph instead of collecting the information in a single one. All graphs could subsequently be compared, and if one adjacency is favored in one graph, but not in the others, then this shows a conflict to be studied.

Another possibility is to try to untangle the inversions, already before the contig adjacency graph is computed from the matches. In the recent literature we observed an approach that tried exactly this [28] by detecting so called *inversion signatures*. Given these signatures, it is under some circumstances possible to untangle the inversion events; however, this is not possible in general.

Ultimately, we have to face the question whether further endeavor is necessary for reference based layouting of contigs. Certainly, approaches have been improved over the years. But is further development still required?

The answer is ambivalent. On the one hand, there is still room for improvement as demonstrated above. On the other hand, we have to admit that, most likely, one day the layouting of contigs will be gratuitous. By the time when sequencing approaches can either obtain very large reads of several hundred kilobases, or even

read complete chromosomes at once, the assembly and finishing of genomes will become a minor matter. Consequently, also approaches to layout contigs will vanish.

Nevertheless, we think that some of our ideas and implementations even then can be considered useful. Very large reads can also be matched on a reference genome with *r2cat*. The resulting synteny plots give an impression about rearrangements and also about repetitive regions of both genomes, and additionally the reads can be arranged according to the reference using the simple mapping procedure. Moreover, complete genomes can also be compared visually, as already discussed in Section 3.3. Due to an open source licensing, the corresponding parts of our software to match genomes or to display synteny may be integrated in future projects as well.

Even the contig adjacency graph, which is specialized for contig layouting and was solely designed to do this in the context of multiple reference sequences, can possibly be reused in the field of comparative analysis of genomes: A *'gene' adjacency graph* could be employed to discover genes that occur in different related species in near proximity and thus are perhaps functionally associated. Besides marginal changes of the parameters, we believe that the current implementation can be used with a FASTA file containing genes instead of contigs. The gene adjacency graph has maybe even advantages over existing approaches that rely on a prior homology assignment. While these commonly only accept sequences of uniquely labeled genes as input, our graph additionally processes the distances between homologous regions and also includes incompletely matching genes implicitly.

This last example shows that some concepts and methods devised in this thesis are more generally applicable. Possibly they will contribute to other important subjects in bioinformatics, or even beyond. We are very excited where this might lead to in the future.

Acknowledgments

I am very grateful to all the people who supported and encouraged me in the last five years. Thanks for cheering me up in hard times and sharing nice moments in good times. The following people, I would like to acknowledge by name.

At first, I want to thank Jens Stoye for his support and for giving me the opportunity to work in his group. You were not only an adviser and boss, but also a good friend.

Many thanks go also to Andreas Tauch for his willingness to review this thesis and for supporting the collaboration with his group.

During my Ph. D. years in Bielefeld, I enjoyed being a member of the Genome Informatics group. In particular, I remember various retreats and social events where we had fun and experienced science. I want to thank all members, past and present, that I was in contact with for an amicable atmosphere. Especially, Roland Wittler, my good friend and office mate, made this time very pleasant for me. Thanks for sharing a passion for photography and espresso making with me. Furthermore, Roland often gave valuable hints and contributed to my work by helping to refine concepts. I also want to recognize our secretary Heike Samuel as the soul of our group who keeps everything together.

On the professional side, I am thankful to all people who helped me with ideas and discussions, who gave hints or support, especially Jochen Blom, Alexander Goesmann, and Tim Nattkemper. Thanks to Christian Rückert, Susanne Schneiker-Bekel, Eva Trost, and Daniel Wibberg for their collaboration, their biological expertise, and for providing datasets. Additionally, I want to thank several people for proofreading parts of this thesis: My sister Monika Hare, and my colleagues Pina Krell and Patrick Schwientek. Roland Wittler even read the whole manuscript.

Not only was social and professional encouragement important; I also want to acknowledge the financial support I received. Thank you, Jens, for the position in your group. I am also grateful to the 'International NRW Graduate School in Bioinformatics and Genome Research'. Besides providing travel grants for conferences and workshops, like BREW 2007, ISMB 2008, and WABI 2009, the graduate school gave me the possibility to meet the Dalai Lama in September 2007 in Münster.

Ending these acknowledgments, I dedicate the last words to my family and my loved ones. I am grateful for the constant support and care from my parents Eva and Franz Husemann, and my sisters Monika Hare and Uta Sander. Finally, I thank my fiancée Svea Stork for her patience and encouragement.

Bibliography

[1] C. Adessi, G. Matton, G. Ayala, G. Turcatti, J. J. Mermod, P. Mayer, and E. Kawashima. Solid phase DNA amplification: characterisation of primer attachment and amplification mechanisms. *Nucleic Acids Res.*, 28(20):e87, 2000.

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

[3] 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.

[4] J. A. Amgarten Quitzau and J. Stoye. Detecting repeat families in incompletely sequenced genomes. In *Proc. WABI*, vol. 5251 of *LNBI*, 342–353. 2008.

[5] S. Anderson. Shotgun DNA sequencing using cloned DNase I-generated fragments. *Nucleic Acids Res.*, 9(13):3015–3027, 1981.

[6] D. L. Applegate, R. E. Bixby, V. Chvátal, and W. J. Cook. *The Traveling Salesman Problem: A computational study*. Princeton University Press, 2006.

[7] S. Assefa, T. M. Keane, T. D. Otto, C. Newbold, and M. Berriman. ABACAS: algorithm based automatic contiguation of assembled sequences. *Bioinformatics*, 25:1968–1969, 2009.

[8] O. T. Avery, C. M. MacLeod, and M. McCarty. Studies on the chemical nature of the substance inducing transformation of pneumococcal types. *J. Exp. Med.*, 79(2):137–158, 1944.

[9] D. Bartels, S. Kespohl, S. Albaum, T. Drüke, A. Goesmann, *et al.* BACCardI—a tool for the validation of genomic assemblies, assisting genome finishing and intergenome comparison. *Bioinformatics*, 21(7):853–859, 2005.

[10] S. Batzoglou. The many faces of sequence alignment. *Brief. Bioinform.*, 6(1):6–22, 2005.

[11] S. Batzoglou, D. B. Jaffe, K. Stanley, J. Butler, S. Gnerre, E. Mauceli, B. Berger, J. P. Mesirov, and E. S. Lander. ARACHNE: a whole-genome shotgun assembler. *Genome Res.*, 12(1):177, 2002.

[12] D. A. Benson, I. Karsch-Mizrachi, D. J. Lipman, J. Ostell, B. A. Rapp, and D. L. Wheeler. Genbank. *Nucleic Acids Res.*, 28(1):15–18, 2000.

[13] J. L. Bentley. Fast algorithms for Geometric Traveling Salesman Problems. *Informs. J. Comp.*, 4(4):387–411, 1992.

[14] A. Bird. DNA methylation patterns and epigenetic memory. *Genes Dev.*, 16(1):6–21, 2002.

[15] J. Blom, S. P. Albaum, D. Doppmeier, A. Pühler, F.-J. Vorhölter, and A. Goesmann. EDGAR: a software framework for the comparative analysis of microbial genomes. *BMC Bioinformatics*, 10:154, 2009.

[16] I. Braslavsky, B. Hebert, E. Kartalov, and S. R. Quake. Sequence information can be obtained from single DNA molecules. In *Proc. Natl. Acad. Sci. USA*, vol. 100, 3960–3964. 2003.

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

[18] J. Butler, I. MacCallum, M. Kleber, I. A. Shlyakhter, M. K. Belmonte, E. S. Lander, C. Nusbaum, and D. B. Jaffe. ALLPATHS: de novo assembly of whole-genome shotgun microreads. *Genome Res.*, 18(5):810–820, 2008.

[19] T. J. Carver, K. M. Rutherford, M. Berriman, M.-A. Rajandream, B. G. Barrell, and J. Parkhill. ACT: the Artemis Comparison Tool. *Bioinformatics*, 21(16):3422–3423, 2005.

[20] M. J. Chaisson and P. A. Pevzner. Short read fragment assembly of bacterial genomes. *Genome Res.*, 18(2):324–330, 2008.

[21] E. Chargaff. Chemical specificity of nucleic acids and mechanism of their enzymatic degradation. *Experientia*, 6(6):201–209, 1950.

[22] J. Clarke, H.-C. Wu, L. Jayasinghe, A. Patel, S. Reid, and H. Bayley. Continuous base identification for single-molecule nanopore DNA sequencing. *Nature Nanotech.*, 4(4):265–270, 2009.

[23] S. T. Cole, R. Brosch, J. Parkhill, T. Garnier, C. Churcher, *et al.* Deciphering the biology of *Mycobacterium tuberculosis* from the complete genome sequence. *Nature*, 393(6685):537–544, 1998.

[24] 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.

[25] A. E. Darling, B. Mau, and N. T. Perna. progressiveMauve: Multiple genome alignment with gene gain, loss and rearrangement. *PLoS ONE*, 5(6):e11 147, 2010.

[26] A. E. Darling, I. Miklós, and M. A. Ragan. Dynamics of genome rearrangement in bacterial populations. *PLoS Genet.*, 4(7):e1000 128, 2008.

[27] F. Dekking, C. Kraaikamp, H. Lopuhaä, and L. Meester. *A Modern Introduction to Probability and Statistics: Understanding Why and How*. Springer, 2007.

[28] Z. Dias, U. Dias, and J. C. Setubal. Using inversion signatures to generate draft genome sequence scaffolds. In *Proc. ACM Conference on Bioinformatics, Computational Biology and Biomedicine*. 2011.

[29] J. C. Dohm, C. Lottaz, T. Borodina, and H. Himmelbauer. SHARCGS, a fast and highly accurate short-read assembly algorithm for de novo genomic sequencing. *Genome Res.*, 17(11):1697–1706, 2007.

[30] J. Eid, A. Fehr, J. Gray, K. Luong, J. Lyle, *et al.* Real-time DNA sequencing from single polymerase molecules. *Science*, 323(5910):133–138, 2009.

[31] M. Fedurco, A. Romieu, S. Williams, I. Lawrence, and G. Turcatti. BTA, a novel reagent for DNA attachment on glass and efficient generation of solid-phase amplified DNA colonies. *Nucleic Acids Res.*, 34(3):e22, 2006.

[32] R. E. Franklin and R. G. Gosling. Molecular configuration in sodium thymonucleate. *Nature*, 171(4356):740–741, 1953.

[33] E. R. Gansner and S. C. North. An open graph visualization system and its applications to software engineering. *Softw. Pract. Exper.*, 30:1203–1233, 1999.

[34] A. J. Gonzalez and L. Liao. Clustering exact matches of pairwise sequence alignments by weighted linear regression. *BMC Bioinformatics*, 9:102, 2008.

[35] D. Gordon, C. Abajian, and P. Green. Consed: a graphical tool for sequence finishing. *Genome Res.*, 8(3):195–202, 1998.

[36] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[37] R. W. Hamming. Error detecting and error correcting codes. *Bell System Tech. J.*, 29:147–160, 1950.

[38] T. D. Harris, P. R. Buzby, H. Babcock, E. Beer, J. Bowers, *et al.* Single-molecule DNA sequencing of a viral genome. *Science*, 320(5872):106–109, 2008.

[39] J. Heer, S. K. Card, and J. A. Landay. prefuse: a toolkit for interactive information visualization. In *Proc. SIGCHI conference on Human factors in computing systems*, CHI, 421–430. 2005.

[40] S. A. F. T. van Hijum, A. L. Zomer, O. P. Kuipers, and J. Kok. Projector 2: contig mapping for efficient gap-closure of prokaryotic genome sequence assemblies. *Nucleic Acids Res.*, 33:W560–W566, 2005.

[41] P. Husemann and J. Stoye. Phylogenetic comparative assembly. *Algorithms Mol. Biol.*, 5(1):3, 2010.

[42] P. Husemann and J. Stoye. r2cat: synteny plots and comparative assembly. *Bioinformatics*, 26(4):570–571, 2010.

[43] P. Husemann and J. Stoye. Repeat-aware comparative genome assembly. In *Proc. GCB*, vol. P-173 of *LNI*, 61–70. 2010.

[44] C. A. Hutchison, III. DNA sequencing: bench to bedside and beyond. *Nucleic Acids Res.*, 35(18):6227–6237, 2007.

[45] E. D. Hyman. A new method of sequencing DNA. *Anal. Biochem.*, 174(2):423–436, 1988.

[46] R. M. Idury and M. S. Waterman. A new algorithm for DNA sequence assembly. *J. Comp. Biol.*, 2:291–306, 1995.

[47] M. Imelfort and D. Edwards. De novo sequencing of plant genomes using second-generation technologies. *Brief. Bioinform.*, 10(6):609–618, 2009.

[48] W. R. Jeck, J. A. Reinhardt, D. A. Baltrus, M. T. Hickenbotham, V. Magrini, E. R. Mardis, J. L. Dangl, and C. D. Jones. Extending assembly of short DNA sequences to handle error. *Bioinformatics*, 23(21):2942, 2007.

[49] J. H. Jett, R. A. Keller, J. C. Martin, B. L. Marrone, R. K. Moyzis, R. L. Ratliff, N. K. Seitzinger, E. B. Shera, and C. C. Stewart. High-speed DNA sequencing: an approach based upon fluorescence detection of single molecules. *J. Biomol. Struct. Dyn.*, 7(2):301–309, 1989.

[50] J. Kalinowski, B. Bathe, D. Bartels, N. Bischoff, M. Bott, *et al.* The complete *Corynebacterium glutamicum* ATCC 13032 genome sequence and its impact on the production of L-aspartate-derived amino acids and vitamins. *J. Biotechnol.*, 104(1-3):5–25, 2003.

[51] W. J. Kent. BLAT – the BLAST-like alignment tool. *Genome Res.*, 12(4):656–664, 2002.

[52] R. Knippers. *Molekulare Genetik*. Thieme, Stuttgart, 2001.

[53] T. Koressaar and M. Remm. Enhancements and modifications of primer design program Primer3. *Bioinformatics*, 23(10):1289–1291, 2007.

[54] 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.

[55] E. S. Lander, L. M. Linton, B. Birren, C. Nusbaum, M. C. Zody, *et al.* Initial sequencing and analysis of the human genome. *Nature*, 409(6822):860–921, 2001.

[56] P. Latreille, S. Norton, B. Goldman, J. Henkhaus, N. Miller, *et al.* Optical mapping as a routine tool for bacterial genome sequence finishing. *BMC Genomics*, 8:321, 2007.

[57] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.

[58] H. Li and R. Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

[59] 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.

[60] R. Li, H. Zhu, J. Ruan, W. Qian, X. Fang, *et al.* De novo assembly of human genomes with massively parallel short read sequencing. *Genome Res.*, 20(2):265–272, 2010.

[61] B. Ma, J. Tromp, and M. Li. PatternHunter: faster and more sensitive homology search. *Bioinformatics*, 18(3):440–445, 2002.

[62] A. Magi, M. Benelli, A. Gozzini, F. Girolami, F. Torricelli, and M. L. Brandi. Bioinformatics for next generation sequencing data. *Genes*, 1(2):294–307, 2010.

[63] M. Margulies, M. Egholm, W. E. Altman, S. Attiya, J. S. Bader, *et al.* Genome sequencing in microfabricated high-density picolitre reactors. *Nature*, 437(7057):376–380, 2005.

[64] D. W. Meinke, J. M. Cherry, C. Dean, S. D. Rounsley, and M. Koornneef. *Arabidopsis thaliana*: a model plant for genome analysis. *Science*, 282(5389):662–682, 1998.

[65] J. R. Miller, S. Koren, and G. Sutton. Assembly algorithms for next-generation sequencing data. *Genomics*, 95(6):315–327, 2010.

[66] E. W. Myers, G. G. Sutton, A. L. Delcher, I. M. Dew, D. P. Fasulo, *et al.* A whole-genome assembly of *Drosophila*. *Science*, 287(5461):2196–2204, 2000.

[67] N. Nagarajan, C. Cook, M. Di Bonaventura, H. Ge, A. Richards, K. A. Bishop-Lilly, R. DeSalle, T. D. Read, and M. Pop. Finishing genomes with limited resources: lessons from an ensemble of microbial genomes. *BMC Genomics*, 11(1):242, 2010.

[68] 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.

[69] C. B. Nielsen, M. Cantor, I. Dubchak, D. Gordon, and T. Wang. Visualizing genomes: techniques and challenges. *Nature Methods*, 7:S5–S15, 2010.

[70] P. Nyrén and A. Lundin. Enzymatic method for continuous monitoring of inorganic pyrophosphate synthesis. *Anal. Biochem.*, 151(2):504–509, 1985.

[71] O. Owolabi and D. McGregor. Fast approximate string matching. *Softw. Pract. Exper.*, 18(4):387–393, 1988.

[72] W. R. Pearson and D. J. Lipman. Improved tools for biological sequence comparison. In *Proc. Natl. Acad. Sci. USA*, vol. 85, 2444–2448. 1988.

[73] R. Pethica, G. Barker, T. Kovacs, and J. Gough. TreeVector: Scalable, interactive, phylogenetic trees for the web. *PLoS ONE*, 5(1):e8934, 2010.

[74] M. Pop, D. S. Kosack, and S. L. Salzberg. Hierarchical scaffolding with Bambus. *Genome Res.*, 14(1):149–159, 2004.

[75] D. Pushkarev, N. F. Neff, and S. R. Quake. Single-molecule sequencing of an individual human genome. *Nat. Biotechnol.*, 27(9):847–852, 2009.

[76] K. R. Rasmussen, J. Stoye, and E. W. Myers. Efficient q-gram filters for finding all ϵ-matches over a given length. *J. Comp. Biol.*, 13(2):296–308, 2006.

[77] T. D. Read, S. N. Peterson, N. Tourasse, L. W. Baillie, I. T. Paulsen, *et al.* The genome sequence of *Bacillus anthracis* Ames and comparison to closely related bacteria. *Nature*, 423(6935):81–86, 2003.

[78] D. C. Richter. *Algorithms and Tools for Genome Assembly and Metagenome Analysis*. Ph.D. thesis, Eberhard-Karls-Universität, Tübingen, 2009.

[79] D. C. Richter, S. C. Schuster, and D. H. Huson. OSLay: optimal syntenic layout of unfinished assemblies. *Bioinformatics*, 23(13):1573–1579, 2007.

[80] A. I. Rissman, B. Mau, B. S. Biehl, A. E. Darling, J. D. Glasner, and N. T. Perna. Reordering contigs of draft genomes using the mauve aligner. *Bioinformatics*, 25(16):2071–2073, 2009.

[81] N. Saitou and M. Nei. The neighbor-joining method: a new method for reconstructing phylogenetic trees. *Mol. Biol. Evol.*, 4(4):406–425, 1987.

[82] A. Samad, E. Huff, W. Cai, and D. Schwartz. Optical mapping: a novel, single-molecule approach to genomic analysis. *Genome Res.*, 5:1–4, 1995.

[83] F. Sanger, S. Nicklen, and A. R. Coulson. DNA sequencing with chain-terminating inhibitors. In *Proc. Natl. Acad. Sci. USA*, vol. 74, 5463–5467. 1977.

[84] P. H. Sellers. On the theory and computation of evolutionary distances. *SIAM J. Appl. Math.*, 26(4):787–793, 1974.

[85] J. Shendure, G. J. Porreca, N. B. Reppas, X. Lin, J. P. McCutcheon, A. M. Rosenbaum, M. D. Wang, K. Zhang, R. D. Mitra, and G. M. Church. Accurate multiplex polony sequencing of an evolved bacterial genome. *Science*, 309(5741):1728–1732, 2005.

[86] A. F. A. Smit, R. Hubley, and P. Green. RepeatMasker Open-3.0. http://www.repeatmasker.org, 1996–2010.

[87] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *J. Mol. Biol.*, 147(1):195–197, 1981.

[88] R. Staden. A strategy of DNA sequencing employing computer programs. *Nucleic Acids Res.*, 6(7):2601–2610, 1979.

[89] H. Tang. Genome assembly, rearrangement, and repeats. *Chem. Rev*, 107(8):3391–3406, 2007.

[90] A. Tauch, J. Schneider, R. Szczepanowski, A. Tilker, P. Viehoever, *et al.* Ultrafast pyrosequencing of *Corynebacterium kroppenstedtii* DSM44385 revealed insights into the physiology of a lipophilic corynebacterium that lacks mycolic acids. *J. Biotechnol.*, 136(1-2):22–30, 2008.

[91] A. Tauch, E. Trost, A. Tilker, U. Ludewig, S. Schneiker, *et al.* The lifestyle of *Corynebacterium urealyticum* derived from its complete genome sequence established by pyrosequencing. *J. Biotechnol.*, 136(1-2):11–21, 2008.

[92] J. Thompson and P. Milos. The properties and applications of single-molecule DNA sequencing. *Genome Biol.*, 12(2):217, 2011.

[93] E. Trost, A. Al-Dilaimi, P. Papavasiliou, J. Schneider, P. Viehoever, *et al.* Comparative analysis of two complete *Corynebacterium ulcerans* genomes and detection of candidate virulence factors. *BMC Genomics*, 12(1):383, 2011.

[94] E. Trost, S. Götker, J. Schneider, S. Schneiker-Bekel, R. Szczepanowski, *et al.* Complete genome sequence and lifestyle of black-pigmented *Corynebacterium aurimucosum* ATCC 700975 (formerly *C. nigricans* CN-1) isolated from a vaginal swab of a woman with spontaneous abortion. *BMC Genomics*, 11(1):91, 2010.

[95] E. Trost, L. Ott, J. Schneider, J. Schröder, S. Jaenicke, *et al.* The complete genome sequence of *Corynebacterium pseudotuberculosis* FRC41 isolated from a 12-year-old girl with necrotizing lymphadenitis reveals insights into gene-regulatory networks contributing to virulence. *BMC Genomics*, 11(1):728, 2010.

[96] G. Turcatti, A. Romieu, M. Fedurco, and A. P. Tairi. A new class of cleavable fluorescent nucleotides: synthesis and optimization as reversible terminators for DNA sequencing by synthesis. *Nucleic Acids Res.*, 36(4):e25, 2008.

[97] S. Tweedie, M. Ashburner, K. Falls, P. Leyland, P. McQuilton, *et al.* FlyBase: enhancing *Drosophila* gene ontology annotations. *Nucleic Acids Research*, 37:D555–D559, 2009.

[98] E. Ukkonen. Approximate string-matching with q-grams and maximal matches. *Theor. Comput. Sci.*, 92(1):191–211, 1992.

[99] J. C. Venter, M. D. Adams, E. W. Myers, P. W. Li, R. J. Mural, *et al.* The sequence of the human genome. *Science*, 291(5507):1304–1351, 2001.

[100] R. L. Warren, G. G. Sutton, S. J. M. Jones, and R. A. Holt. Assembling millions of short DNA sequences using SSAKE. *Bioinformatics*, 23(4):500–501, 2007.

[101] M. S. Waterman and M. Eggert. A new algorithm for best subsequence alignments with application to tRNA-rRNA comparisons. *J. Mol. Biol.*, 197:723–728, 1987.

[102] J. D. Watson and F. H. C. Crick. A structure for deoxyribose nucleic acid. *Nature*, 171(4356):737–738, 1953.

[103] J. Wetzel, C. Kingsford, and M. Pop. Assessing the benefits of using mate-pairs to resolve repeats in de novo short-read prokaryotic assemblies. *BMC Bioinformatics*, 12(1):95, 2011.

[104] D. L. Wheeler, C. Chappey, A. E. Lash, D. D. Leipe, T. L. Madden, G. D. Schuler, T. A. Tatusova, and B. A. Rapp. Database resources of the national center for biotechnology information. *Nucleic Acids Res.*, 28(1):10–14, 2000.

[105] D. Wibberg, J. Blom, S. Jaenicke, F. Kollin, O. Rupp, *et al.* Complete genome sequencing of *Agrobacterium* sp H13-3, the former *Rhizobium lupini* H13-3, reveals a tripartite genome consisting of a circular and a linear chromosome and

an accessory plasmid but lacking a tumor-inducing Ti-plasmid. *J. Biotechnol.*, 2011.

[106] Wikimedia. http://commons.wikimedia.org/wiki/File:Difference_DNA_RNA-EN.svg. Creative Commons License BY-SA 3.0. File accessed on July 13th, 2010.

[107] M. H. F. Wilkins, A. R. Stokes, and H. R. Wilson. Molecular structure of deoxypentose nucleic acids. *Nature*, 171(4356):738–740, 1953.

[108] D. R. Zerbino and E. Birney. Velvet: algorithms for de novo short read assembly using de Bruijn graphs. *Genome Res.*, 18(5):821–829, 2008.

[109] F. Zhao, F. Zhao, T. Li, and D. A. Bryant. A new pheromone trail-based genetic algorithm for comparative genome assembly. *Nucleic Acids Res.*, 36(10):3455–3462, 2008.