German Cancer Research Center Heidelberg
Department of Molecular Biophysics
Head of Department: Prof. Dr. Sándor Suhai

MIRA: An Automated Genome and EST Assembler

Ph.D. Thesis to acquire the title of

Doctor scientiarum humanarum (Dr.sc.hum.)

of

The Medical Faculty of Heidelberg

of

The Ruprecht-Karls-University

presented by
Bastien Chevreux

from Duisburg
2005

Dean: Prof. Dr. Claus R. Bartram
Referee: Prof. Dr. Sándor Suhai

Some of the illustrations and parts of the text in this thesis also appeared in the publications Chevreux et al. (1999), Chevreux et al. (2000) and Chevreux et al. (2004).

Avid readers of David Gerrold will certainly recognise the quotes from his books at the beginning of each chapter of this thesis.

Contents

1 Introduction and Motivation **1**
- 1.1 Genome and EST sequencing projects . . . 1
- 1.2 State of the art assembly . . . 2
- 1.3 Aim of the thesis . . . 3

2 Fundamentals of sequencing and assembly **6**
- 2.1 Definitions and methods . . . 6
- 2.2 Error types and rates . . . 17
  - 2.2.1 Errors of the data acquisition process . . . 17
  - 2.2.2 Errors due to biology . . . 20
- 2.3 Peculiarities of EST sequencing . . . 21
  - 2.3.1 Biological background . . . 21
  - 2.3.2 Implications for assembly projects . . . 22

3 Assembly strategies **25**
- 3.1 Existing strategies . . . 25
- 3.2 Developing a new strategy . . . 28
  - 3.2.1 Tackling misassemblies . . . 29
  - 3.2.2 Focussing on observable data . . . 31
  - 3.2.3 Pattern analysis . . . 31

4 Methods and Algorithms **35**
- 4.1 Data preprocessing and input . . . 35
- 4.2 Read scanning . . . 38
  - 4.2.1 Formalising the problem . . . 38
  - 4.2.2 Present algorithms . . . 40
  - 4.2.3 DNA-Shift-AND algorithm . . . 42
  - 4.2.4 The ZEBRA algorithm . . . 45
- 4.3 Systematic match inspection . . . 54
  - 4.3.1 Improving Smith-Waterman alignment by banding . . . 55
  - 4.3.2 Parametrising Smith-Waterman alignment . . . 59

4.3.3 Attributing an alignment overlap . . . 61
4.3.4 Building a graph . . . 63
4.4 Building contigs . . . 64
4.4.1 Pathfinder and contig interaction . . . 64
4.4.2 Path traversal strategies . . . 66
4.4.3 Contig . . . 68
4.4.4 Contig approval methods . . . 69
4.4.5 Consensus and consensus quality computation . . . 74
4.5 Automatic editing . . . 78
4.6 Finding unknown repeats . . . 79
4.6.1 Repeat types . . . 79
4.6.2 Existing approaches . . . 81
4.6.3 Locating repeats through error pattern analysis . . . 81
4.7 Read extension . . . 84
4.7.1 Intra-contig and extra-contig read extension . . . 85
4.7.2 Extension algorithms . . . 85
4.8 Iterative cycling . . . 87
4.9 Modifications for EST assembly . . . 88
4.9.1 Coverage: meeting both extremes . . . 88
4.9.2 Detection of SNPs in genes and gene families . . . 90
4.9.3 Classification of SNPs . . . 93

5 Results and discussion **95**
5.1 Genome assembly . . . 95
5.2 EST assembly . . . 99
5.3 Discussion . . . 105

6 Conclusion and outlook **108**

A Development details **110**
A.1 Programming environment . . . 110
A.2 Programming approaches . . . 112
A.3 Code statistics . . . 113

B Manual pages **114**
B.1 Synopsis . . . 114
B.2 Description . . . 114
B.3 Working modes . . . 116
B.4 Options . . . 116

B.4.1 -GENERAL (-GE) . . . 117
B.4.2 -ESTGENERAL (-EG) . . . 118
B.4.3 -ASSEMBLY (-AS) . . . 119
B.4.4 -DATAPROCESSING (-DP) . . . 120
B.4.5 -CLIPPING (-CL) . . . 121
B.4.6 -ALIGN (-AL) . . . 123
B.4.7 -CONTIG (-CO) . . . 124
B.4.8 -EDIT (-ED) . . . 126
B.4.9 -DIRECTORY (-DIR, -DI) . . . 127
B.4.10 -FILE (-FI) . . . 127
B.4.11 -OUTPUT (-OUT) . . . 128
B.4.12 Quick mode switches . . . 129
B.4.13 Other switches . . . 129
B.5 Input / Output . . . 130
B.5.1 Filenames . . . 130
B.5.2 Fileformats . . . 131
B.5.3 STDOUT . . . 132
B.5.4 STDERR . . . 132
B.5.5 Logfiles . . . 133
B.6 Tags used in the assembly by MIRA and EdIt . . . 133
B.6.1 Tags read (and used) . . . 133
B.6.2 Tags set (and used) . . . 134
B.7 Requirements . . . 135
B.8 Speed and memory considerations . . . 136
B.9 Usage . . . 137
B.9.1 Assembly from scratch with GAP4 and EXP files . . . 137
B.10 Known Problems / Bugs . . . 138
B.11 Caveats . . . 138
B.12 TODOs . . . 140
B.13 Working principles . . . 140
B.14 License, Disclaimer and Copyright . . . 141
B.15 Authors . . . 142
B.16 Miscellaneous . . . 142
B.17 See Also . . . 143

Literature **144**

References . . . 144
Own publications . . . 155

Index **156**

Curriculum Vitae **160**

Acknowledgements **161**

List of Figures

1 Simplified shotgun sequencing process 9
2 Amplification of DNA through vectors 10
3 Location of forward and reverse reads on plasmids 10
4 Example for trace data with the four gel lanes superposed. 11
5 Example for bad trace signal quality, start of trace 18
6 Example for good trace signal quality, middle of trace 18
7 Example for bad signal quality, end of a trace 19
8 Example for a gene architecture in eukaryotic genomes. 22
9 Example for gene splice variations in eukaryotic genomes. 23

10 Conventional assembly 27
11 Integrated assembly 28
12 Explainable assembly discrepancy 30
13 Unexplainable assembly discrepancy 30
14 Phases of a MIRA assembly cycle 32

15 Mode of operation for DNASAND 44
16 Transforming a nucleotide 8-tuple (octet) into a 16 bit hash. . . . 46
17 Transforming an octet with an undefined base into hashes 46
18 Computation of the combined hash-position table 47
19 Splitting a combined hash-position table into subtables. 48
20 Computing a octet distance histogram. 50
21 Compressing imprints into boolean arrays. 52
22 Transforming 9-tuples into hashes allowing for errors 53
23 Read scan matrices. 54
24 Smith-Waterman banding. 56
25 Smith-Waterman band prediction 57
26 Smith-Waterman (SW) band calculation predecessor rules 58
27 Modified Smith-Waterman algorithm: accepted match. 62
28 Modified Smith-Waterman algorithm: rejected match. 62
29 Assembly overlap graph. 63

30 Example for an assembly graph 66
31 Path building examples. 67
32 Assembly graph example. 69
33 Failure example when using simple read acceptance strategies in contigs. 70
34 Contig using additional sequencing information. 71
35 Example for using strict signal checking in contig standard repeat regions. 73
36 Calculating base group qualities 77
37 Discovery of previously unknown long term repeats. 82
38 Resolved long term repeats with PRMB tags. 83
39 Extension of confidence regions. 86
40 Coverage example in non-normalised EST project 89
41 Detection of SNP base in EST sequences 91
42 Merging mRNA transcripts 92
43 Categorising SNPs with sequences that have strain information . 93

44 Sample of an assembly HTML output 107

List of Tables

1 Simple weight matrix example 14

2 Used SW scoring weight matrix 59
3 Default gap length penalty scores 60
4 Scoring weight matrix for the expected score 61

5 Genome projects used for benchmarking 97
6 Comparison of MIRA/Edit, GAP4/cycle and PHRAP assemblies. . 98
7 Results from EST assembly 100
8 Runtime and memory consumption 103

1 Introduction and Motivation

> *"New problems demand new solutions. New solutions create new problems."* (Solomon Short)

Shotgun sequencing genomic sequences for subsequent reconstruction is comparable to assembling a jigsaw puzzle. These genomic puzzles, of course, are much more complex than the average jigsaw puzzle: they tend to be about 500 to 5 million pieces, printed on both sides, with many vital pieces possibly missing. Some of the pieces are dirty or unrecognisable, and several pieces from another puzzle might have been mixed in. Additionally, a few pieces themselves appear to have been cut and reassembled by a very impatient two-year-old with a pair of scissors and a bottle of glue.

The extensively studied reconstruction of the unknown, correct contiguous nucleic acid sequence by inferring it through the help of a number of representations[1] is called the assembly problem. The devil is in the details, however. If the collected sequences were 100% error free, then many problems would not occur. In reality, the extraction of data by electrophoresis is a physical process in which errors due to biochemical phenomena show up quite often. Ewing et al. (1998); Ewing and Green (1998) show that – together with errors occurring in the subsequent signal analysis – current laboratory technologies total an error rate that might be anywhere between 0.1% – for good parts in the middle of a sequence – and more than 10% in bad parts at the very beginning and at the end. This error rate, combined with the sometimes exacerbating fact that both DNA and RNA tend to contain highly repetitive stretches with only very few bases differing across different repeat locations, impedes the assembly process in a daunting way.

1.1 Genome and EST sequencing projects

Todays large scale genome sequencing efforts are a semi-industrial process (Dear et al. (1998)) and produce enormous quantities of data. They are nearly

[1] also called fragments (see Myers (1995)) or readings (reads)

all based on the chain-termination dideoxy method published by Sanger et al. (1977) in one way or another. But the gel or capillary electrophoresis used can determine only about a maximum of 1,000 to 1,500 bases in one run, the high quality stretch with low error probabilities for the called bases often being around the first 400 to 500 bases. Current sequencing strategies for a larger contiguous DNA sequence (contig) or for a whole genome – ranging anywhere between 20 kilobases (kb) and 3,000 megabases (mb) – therefore require an indirect approach. Basically the given DNA is fragmented in hundreds or thousands of overlapping subclones (Durbin and Dear (1998)), analysed by fluorescent-dye electrophoresis and subsequently the subclones are reassembled in-silico. This computer-based reconstruction of DNA (or RNA) from fragments is called "the assembly problem".

On the way to understand all genes of an organism, it is now clear that the genome sequence alone may not be enough, especially if the organism shows a high degree of complexity like, e.g. in mammals. Therefore, analysis of the genome must be supported by an understanding of its transcription – the transcriptome – occurring in cells. Projects that focus on sequencing mRNA transcripts are also called EST projects as they analyse *Expressed Sequence Tags*. This direct RNA sequencing remains – citing Camargo et al. (2001) – the "most definitive approach to the elucidation of transcripts". At the same time, Bonfield et al. (1998) conclude that "direct sequencing is required to define the precise location and nature of any [mutational] change", as this method ensures highest reliability and quality regarding the definition of single nucleotide polymorphisms (SNPs). EST projects constitute thus a perfect opportunity to both elucidate the transcriptome and analyse mutational polymorphisms contained therein, especially when doing cross-species EST analyses as was shown in Chevreux et al. (2004).

1.2 State of the art assembly

The sheer amount of data collected implies that the assembly itself must be done by computer-based methods. The above mentioned error rates and repetitive properties of DNA require the algorithms to tolerate faults and seek alternatives in a given solution space. Wang and Jiang (1994) showed that the assembly problem – even using error free representations of the true sequence – is NP complete. This means that the volume of data can only be assembled by approximating strategies, relying on heuristic algorithms that are well-behaved

in both time and space complexity.

The evolution of assembly strategies clearly moved away from simply using base sequences toward approaches using additional information like, e.g., coverage analysis, sequence orientation, quality and probability values and template identity. A common characteristic to all existing assemblers is that they rely on the quality values with which the bases were previously attributed. Within this process which is named "base-calling", an error probability is computed by the base caller program to express the confidence with which the called base is thought to be the true base. The positive aspect is the possibility for assemblers to decide in favour of the best, most probable bases when a discrepancy occurs. The negative aspects of the most widely used current base callers are 1) the fact that base probability values sometimes cannot be computed accurately[2] enough and 2) their inability to write confidence values for optional, uncalled bases at the same place, which would help assembly algorithms to compute better alternative solutions.

The common method for assembling DNA and RNA therefore consists of using fault-tolerant algorithms that produce a basic sequence which then has to be reviewed and manually corrected by human experts, the so called "finishing". Incorrectly assembled sequences must be dismantled and reassembled at different places. For large scale sequencing projects, this slow and very inefficient method contradicts the principle of parsimony and represents the most important bottle-neck and cause of errors, even if powerful finishing tools are now available (see Lawrence et al. (1994); Staden et al. (1997); Gordon et al. (1998)). But errors still happen much too frequently and especially sequencing projects of higher organisms are affected by problems as their genomes consistently contain more and longer repetitive regions that create new levels of complexity in the assembly process. Recent studies from Cheung et al. (2003) confirmed this by an in-depth analysis of the human genome sequence assembly: as of June 2002, "1.28% of the sequences in the 3,043.1 megabases of the genome are likely to be involved in sequence misalignment errors".

1.3 Aim of the thesis

The essential criterion in the design of an assembler is the quality of the final result. Although different groups may have different – sometimes arbitrary – sets of acceptance criteria for quality aspects of assemblies, the target accuracy

[2] e.g. for new electrophoresis methods

of 1 error in 10,000 finished bases in DNA sequence set by the Human Genome Project can be seen as most demanding, especially when assembling sequences from higher eukaryotic organisms. An assembly system has to support this target by making cautious use of the available data and increase automation by reducing human involvement in correcting assembly errors. This has to happen through ensuring a firm base and good building blocks in the beginning of an assembly.

The aim of this thesis is therefore centred at reducing assembly errors caused by repetitive sequences as well as increasing the reliability of consensus sequences derived from automatically assembled projects.

To achieve this goal, the first key point of the approach developed in this thesis is the usage of all additionally gathered information like, e.g., repeat tags, template insert sizes, quality values, original signal traces, etc. as checking mechanisms. The information is used to confirm and optimise the basic assembly and to identify possibilities to discern between different occurrences of repetitive sequences. The second key point in the algorithms designed is the combination of the assembler with the capabilities of an automatic editor. Both the assembler and the automatic editor are separate programs and can run separately, but the task of assembly and finishing is so closely related that both parts can include routines from each other (see also Pfisterer and Wetter (1999); Chevreux et al. (2000)).

In this combined information analysis and integration process, the signal-analysis aided assembler promises two substantial advantages compared to a sequential-base-caller-and-assembler strategy:

1. The assembler gains the ability to perform signal analysis on partly assembled data. Analysing experimental signal data at precise points with a given hypothesis 'in mind' helps to discern possible base caller errors from errors due to misassemblies, especially in problematic regions like repeats where simple base probabilities alone could not help.

2. During the assembly process, sequences in preliminary builds can be automatically edited to increase their quality if the signal data supports the hypothesis of an error that occurred in the base calling step. The edited sequences in turn can be used to increase assembly quality in the ongoing assembly process.

As the underlying problem of string assembly has been shown to be NP-hard by Armen and Stein (1995), heuristic algorithms represent the only possibility

to solve the challenges induced by assembly projects. This thesis presents the developed strategy of using all available assembly data together with the major assets of signal analysis and automatic editing as a tremendously useful and versatile policy for improving existing heuristic algorithms.

In chapter 2, the present thesis first presents and formally defines the problems posed by sequencing and describes the reasons for the complexity of this task. Chapter 3 presents the theory developed and the algorithms implemented for a new type of assembler that combines – and substantially extends – the strengths of existing assembly approaches while mimicking and automating some assembly analysis strategies used by human experts. Methods and algorithms both used in bioinformatics and developed for this thesis are described in chapter 4, together with their advantages and disadvantages for different applications in sequence assembly. The focus is to show both theoretically and practically how relatively simple, yet well behaved and highly effective algorithms can be used to build an assembler that handles large and complex real-world shotgun projects. Chapter 5 presents the results obtained with the assembler developed in this thesis in comparison with the two most widely used genome assembly systems at the time of the study. Furthermore, results for assembly of EST projects are presented. Conclusions of this work are presented in the last chapter (6) together with some remarks on additional work that could further improve the assembly system. Appendix A contains some thoughts and insights on implementational issues gained through this research project. Appendix B shows the documentation for the 2.2.8 version as of October 2004 for the `mira`[3] and `miraEST` assembler which has an accurate description on usage modes. The last but nevertheless important appendix 6 contains acknowledgements to people who helped this work become a success.

In closing this introductionary chapter, it must be noted that the work presented within this thesis focuses on the assembler part of the assembler $\leftrightarrow$ automatic editor combination. Conditions an automatic editor system has to comply with when providing assistance to an assembler will also be explained although detailed descriptions of the automatic editor algorithms are not subject of this work.[4]

[3] which is an acronym for **M**imicking **I**ntelligent **R**ead **A**ssembly

[4] The automatic editor is subject of a thesis to be presented by Thomas Pfisterer

2 Fundamentals of sequencing and assembly - integrating biology and computer science

"Understanding the laws of nature does not mean that we are immune to their operations." (Solomon Short)

This chapter is meant to be a short introduction to the chemical, biological and computational aspects of sequencing and assembly. It presents and formally defines the problem of genome and EST sequencing while introducing the terminology as it is used throughout this thesis. This chapter cannot be an exhaustive reference to sequencing methods, DNA properties or biological background information at large, some facts have been simplified to clarify the context. See for example DOE (1992), Bruce et al. (1994) or Klug and Cummings (1996) for more advanced information.

2.1 Definitions and methods

A genome physically and chemically consists of tightly coiled strands of deoxyribonucleic acids (DNA) which is normally organised in a twisted double helix. The actual information contained in DNA is encoded by four different nitrogenous bases: Adenine (A), Cytosine (C), Guanine (G) and Thymine (T).

Definition 1 (Alphabet) *An Alphabet $\mathcal{A}$ includes the characters needed for textual representation of DNA.*

In a DNA helix, each base will always pair with its complementary base.

Definition 2 (Complement base) *The complement base of Adenine is Thymine, the complement base of Cytosine is Guanine. If b defines a base, $\overline{b}$ represents its complement.*

The bases are arranged along a sugar-phosphate backbone. The information needed to create any organism is specified by the sequence of bases – and the molecular structure defined by it – within the DNA. Refer to Bruce et al. (1994) for more information on working mechanisms of nucleic acids in general.

Definition 3 (Sequence) *A sequence $\mathcal{S}$ is an ordered succession of characters from $\mathcal{A}$.*

As will be explained later on, extracting DNA sequence information from cells is a tedious process involving many different steps. Each of these steps is prone to errors. Due to these errors, uncertainties remain once a sequence has been determined.

Definition 4 ($\mathcal{A}^x$) *The alphabet $\mathcal{A}$ exists through different representations $\mathcal{A}^x$ to meet the needs of real world data sets used in assembly processes.*

$\mathcal{A}^b$ contains only those characters that are needed to represent the discrete DNA sequence as it is found in cells. $\mathcal{A}^b$ = {A, C, G, T}

$\mathcal{A}^n$ extends $\mathcal{A}^b$ with the character N (aNy) which shall stand for bases that could not be determined more specifically in the analysis process. $\mathcal{A}^n$= {A, C, G, T, N}

$\mathcal{A}^g$ extends $\mathcal{A}^n$ with the gap character "$*$" which implies missing bases within a sequence $\mathcal{S}$. "$*$" is also sometimes denoted as padding character. $\mathcal{A}^g$= {A, C, G, T, N, $*$}

Please note that $\mathcal{A}^b \subset \mathcal{A}^n \subset \mathcal{A}^g$

Beside these alphabets, a lot of other different alphabets have been created to meet the needs of specific employments, from which the IUPAC (International Union of Pure and Applied Chemistry) code is the most notable. The IUPAC code describes any uncertainty with one character.[1]

Definition 5 (Length of a sequence $\mathcal{S}$) *The length of a sequence is the number of characters $\in \mathcal{A}$ of a sequence $\mathcal{S}$. The length is denoted by $||$. Please note that $0 \leq | \mathcal{S} | < \infty$*

With the help of the definition for the length of a sequence, the raw definition of a sequence can be refined.

[1] e.g., the symbol 'W' for an uncertainty between A (Adenosin) and T (Thymin)

Definition 6 (Sequence $\mathcal{S}^x$) *A sequence $\mathcal{S}^x$ is an ordered succession of characters from $\mathcal{A}^x$.*

$$\mathcal{S}^x = \left(\begin{array}{cccc} s_1^x, & \ldots, & s_n^x \end{array} \right) \qquad with \qquad n = \mid \mathcal{S}^x \mid \quad and \quad s_i^x \in \mathcal{A}^x$$

Definition 7 (Reverse complement) *The reverse complement $\overline{\mathcal{S}}$ of a sequence $\mathcal{S}$ is constructed by writing the sequence $\mathcal{S}$ backwards and replacing the bases by their complement. The special characters "$*$" and N do not get replaced.*

$$\overline{\mathcal{S}^x} = \left(\begin{array}{cccc} \overline{s_n^x}, & \ldots, & \overline{s_1^x} \end{array} \right)$$

The length of DNA sequence of organisms present on earth ranges from a few hundred kilobases up to several gigabases. For example, the bacterium *Helicobacter pylori 26695* (GenBank accession number AE000511) has 1,667,867 bases (1.7 megabases) in its form as completed on September 7, 2001. In contrast, the human genome consists of approximately 3 gigabases.

But the gel or capillary electrophoresis technologies used to analyse DNA sequences can determine only about a maximum of 1,000 to 1,500 consequent bases, the high quality stretch with low error probabilities for the called bases often being around the first 400 to 500 bases. The problem to be solved is to find a method that allows the sequencing of long genomes, with the limiting factor that no subsequence to be analysed may be longer than 1 to 1.5kb.

Definition 8 (Shotgun method) *The shotgun method consist of cloning and analysing short overlapping DNA fragments randomly generated from a genome (see also DOE (1992)) and subsequently reassembling back in silico the experimentally gained sequences through computational methods.*

A simplified overview on the principles of the shotgun methods is given in figure 1. The DNA to be sequenced is first purified and amplified through cloning and then randomly fragmented into small parts, for example through sonication or the use of restriction enzymes. The fragments, also called *inserts*, are then sorted by size into so-called *clone libraries* of different sizes. For example, a genome sequencing project might use four clone libraries, where the fragments of the first library are approximately 2 kilobases in size, the second has 5 kilobases, the third 10 kilobases and the last 50 kilobases.

With the sensitivity levels of current sequencing technology, any type of nucleic acid sequence that is to be sequenced must first be amplified prior to its determination. This is done by inserting the fragment (insert) into a so-called

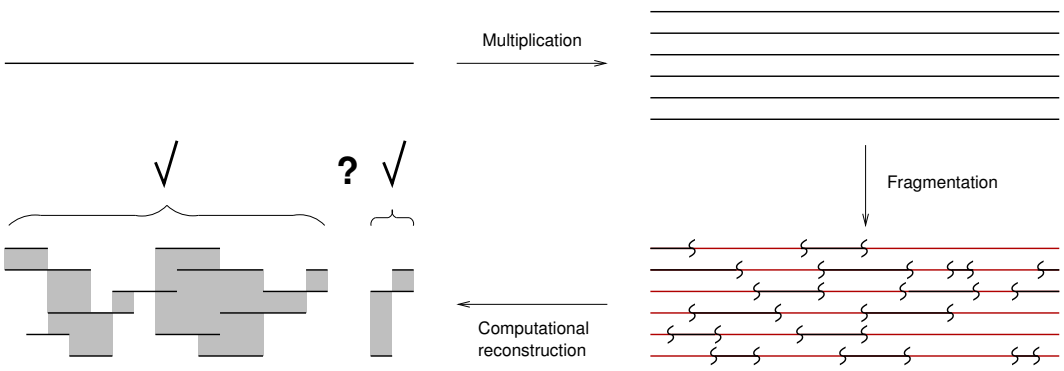

Figure 1: Simplified shotgun fragmentation sequence analysis. A DNA is cloned and fragmented. Fragments too long for electrophoresis are filtered, the remaining fragments are sequenced. The original DNA sequence must be reconstructed through computational methods by analysing overlapping regions between fragments. In some cases, no fragment covers certain parts of the original DNA sequence, which results in a gap.

cloning or sequencing vector and then inserting the resulting construct (a *plasmid*) into host cells – e.g. a bacterium like *E. coli* – for amplification as is shown in figure 2.

Definition 9 (Sequencing or cloning vectors) *Vectors are small fragments of DNA – often derived from viruses or bacterial phages – that have the intrinsic capability to replicate themselves and the sequence that is eventually attached to them. The payload sequence is also called "insert", its length "insert size".*

The construct plasmid is then introduced into a host cell , where it replicates together with the cell.

Once each vector/payload construct is amplified enough, a labelled polymerase replication process is started in the known sequencing vector part in forward and reverse direction as shown in figure 3. When using the chain termination dideoxy method published by Sanger et al. (1977), the replicates are terminated by dideoxynucleotides labelled with a fluorescent dye, a different dye for each terminating base. Each replicate will terminate at a different position of the fragment as the dideoxynucleotide prevents the polymerase from continuing the synthesis.

The combined replicates are then separated according to length in a gel or capillary electrophoresis process. The identity of a terminating base from a particular replicate can be detected by a two- or four-channel laser scanner as

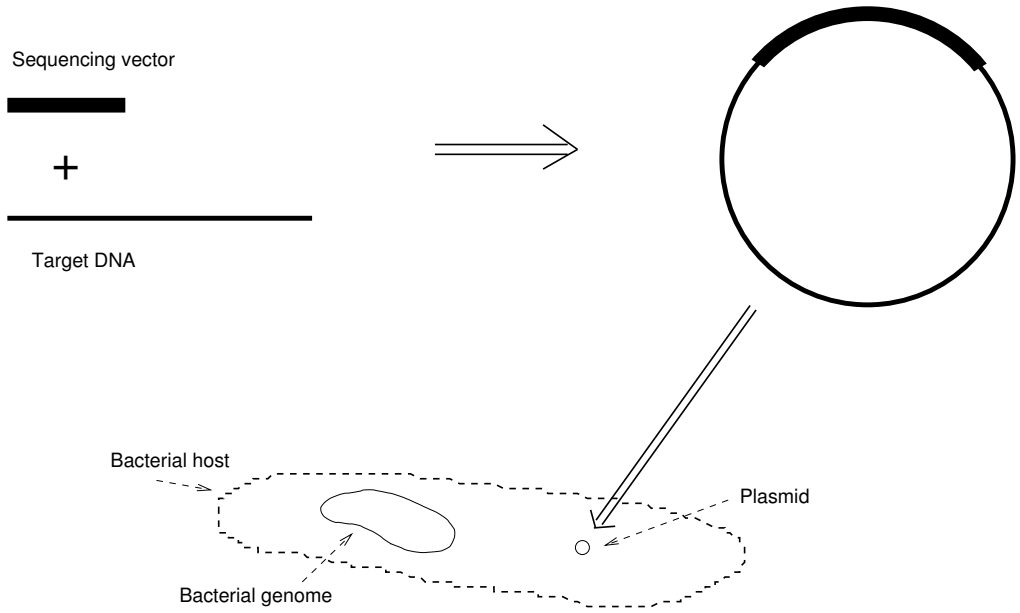

Figure 2: Amplification of DNA through vectors. Small DNA fragments can be easily amplified by attaching it to a vector sequence. The vector/payload construct is called plasmid and is inserted into a bacterial host, the vector gives it the ability to be replicated together with its host.

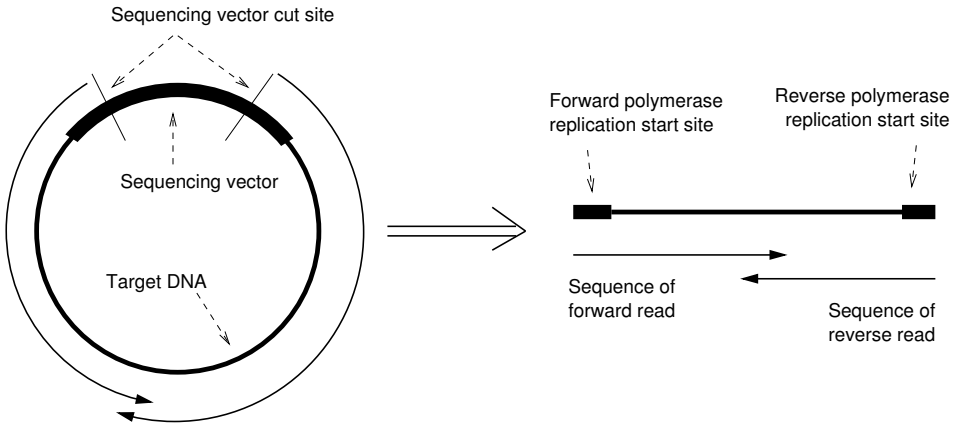

Figure 3: Simplified illustration of location of forward and reverse reads on plasmids. Amplified plasmids are extracted from the host cells and linearised. Then a chain-termination dideoxy polymerase reaction produces fluorescent labelled replicates of different sizes in forward and reverse direction that can be analysed by electrophoresis.

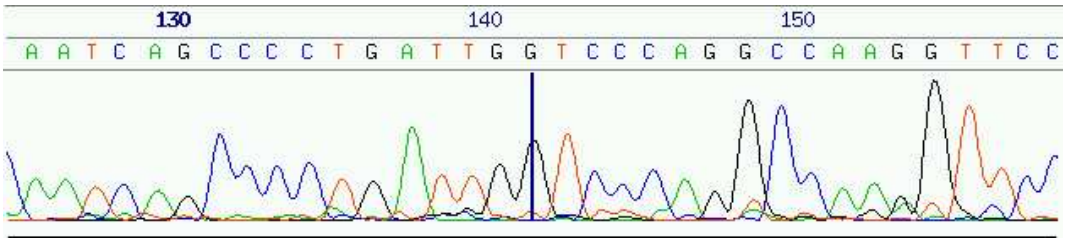

Figure 4: Example for trace data with the four gel lanes superposed.

peak a in the fluorescence signal in the base specific channel. As short replicates will travel faster through the electrophoresis medium, the order of the replicates that pass the scanning mechanism also determines the order of the bases in each insert and thus the order of bases in the DNA fragment sequence.

Definition 10 (Trace) *The DNA signal gained through electrophoresis is called a trace. Each base (A, C, G and T) has its own signal trace, but they are frequently shown superposed as shown in figure 4.*

Although trace signals are the most accurate representation of the sequences that have been analysed (see figure 4), they tend to be impractical as a basis for upcoming computational steps. Further on, genomes are a discrete ordered sequence of the four nucleotide bases. The trace signals must therefore be analysed to extract the original sequence.

Definition 11 (Base-calling) *Base-calling is the process by which a sequence $\mathcal{S}^b$ represented by a signal trace T is deduced from the signals. Usually, the alphabet $\mathcal{A}^n$ has to be used to generate a sequence $\mathcal{S}^n$ as the trace data is not always unequivocal.*

Please refer to section 2.2 for a more in-depth discussion on the origin and the problems caused by wrong base calls.

Definition 12 (Read) *A read is composed by the sequence $\mathcal{S}^n$ and the trace data from which it has been inferred. Additionally, data gained by preliminary analysis of the sequence – like sequencing vector pollution – or data from other sources – like chemistry and gel used – can augment the information present.*

The read sequences extracted from the forward polymerase replication (called *forward reads*) and the reverse polymerase replication (called *reverse* or *complement reads*) are all contained in the original DNA sequence of interest and are

supposed to cover it completely in an overlapping way multiple times. But this is a working hypothesis only, based on stochastic assumptions. Although the fragments are assumed to be distributed uniformly along the DNA sequence (Idury and Waterman (1995)), chemical properties of biological sequences occasionally introduce a bias. In real-world projects, the reads will probably leave some gaps in the DNA sequence to be analysed, sometimes due to chemical properties of the sequence at this place (like coiling of DNA), sometimes just due to 'bad luck'. To reconstruct the original DNA sequence by computational means, the relative order and orientation [2] of the fragments must be determined. This inference is done using alignment algorithms.

Definition 13 (Alignment) *An alignment is a* 2*-dimensional matrix formed by* k *sequences* $\mathcal{S}^x$*.*

$$\mathbf{L} = \begin{pmatrix} \mathcal{S}_1^x \\ \vdots \\ \mathcal{S}_k^x \end{pmatrix} \tag{2.1}$$

which can also be written as

$$\mathbf{L} = \begin{pmatrix} s_{11} & \dots & s_{1n} \\ \vdots & \ddots & \vdots \\ s_{k1} & \dots & s_{kn} \end{pmatrix} \tag{2.2}$$

It is clear that this definition of an alignment implies that all contained sequences $\mathcal{S}^x$ must have the same lengths. Theoretically, one could use the gap character ("$*$") to pad the sequences to the left and/or to the right and bring them to the desired lengths. But a "$*$" implies missing characters and is often seen as an error of some sort. This is clearly not the case for shotgun reads, as bases to the left or to the right of a sequence have simply not been determined in the lab.

Definition 14 (Endgaps) *Endgaps are special characters that may be present at the start or at the end of a sequence* $\mathcal{S}^x$ *(but never within), to increase formally the length of a sequence without changing the contextual information contained in the sequence.*

[2] the experimentally gained sequences have a 50% chance of being in reverse complement orientation

The alphabets $\mathcal{A}^x$ of the sequences $\mathcal{S}^x$ are therefore extended by the ∇character.

$$\mathcal{A}^B = \mathcal{A}^b \cup \{\nabla\} = \{A, C, G, T, \nabla\} \tag{2.3}$$
$$\mathcal{A}^N = \mathcal{A}^n \cup \{\nabla\} = \{A, C, G, T, N, \nabla\} \tag{2.4}$$
$$\mathcal{A}^G = \mathcal{A}^g \cup \{\nabla\} = \{A, C, G, T, N, *, \nabla\} \tag{2.5}$$

Please note that $\mathcal{A}^B \subset \mathcal{A}^N \subset \mathcal{A}^G$

Normally, in alignments the ∇character is often represented by "." (point) or " " (blank). As this may lead sometimes to confusion for printed text, the end-gap character will be represented by ∇in plain text and with a point or a blank in alignments. Please also note that

$$\bigvee_{s_i^X \in \mathcal{S}^X = \nabla} \left(\bigvee_{j < i} \left(s_j^X = \nabla\right) \right)$$

and

$$\bigvee_{s_i^X \in \mathcal{S}^X = \nabla} \left(\bigvee_{j > i} \left(s_j^X = \nabla\right) \right)$$

In plain words: a ∇ character may not be enclosed by characters from $\mathcal{A}^x$.

Theorem 1 *Without further proof, it will be assumed that every sequence $\mathcal{S}^x$ can also be written as $\nabla_k \mathcal{S}^x \nabla_l$ with ∇ being present k and l times at the start respectively end of a sequence.*

$$\nabla_k \mathcal{S}^x \nabla_l = \nabla_i \mathcal{S}^x \nabla_j \qquad \textit{for any } i, j, k, l \in \mathbb{N}_0$$

E.g.: ACGT = ACGT∇ = $\nabla\nabla\nabla$ACGT∇

Any sequence can therefore be brought to a desired length to fit into an alignment.

Definition 15 (Extended sequence length) *The extended sequence length is written $||\mathcal{S}^X||$ and calculated by counting every character $\mathcal{S}^X \in \mathcal{A}^X$.*

E.g.: $\mathcal{S}^X$ = $\nabla\nabla\nabla$ACGT∇

$|\mathcal{S}^X|$= 4

$||\mathcal{S}^X||$= 8

As can be seen in equation 2.2 on page 12, each column in an alignment forms a k-tuple over the alphabet $\mathcal{A}^X$. Two important numbers can be extracted from

Table 1: A simple weight matrix example. Each match of two bases $\in \mathcal{A}^b$ is scored with the value 1, mismatches with -1. Gaps within a sequence ($*$) penalise the alignment with -2.

	A	C	G	T	$*$
A	1	-1	-1	-1	-2
C	-1	1	-1	-1	-2
G	-1	-1	1	-1	-2
T	-1	-1	-1	1	-2
$*$	-2	-2	-2	-2	0

this k-tuple: coverage and column score.

Definition 16 (Coverage) *The number of characters $\in \mathcal{A}^x$ in a column of an alignment is called coverage. From the biological point of view, the coverage describes the number of times a given (sub-)sequence of DNA or RNA was sequenced. Please note that endgaps (∇) do not count as valid coverage characters: they are used as such only to pad shorter sequences into longer alignments. The characters N and $*$, however, imply a "problem" of some sort in the base-calling process, but otherwise sequence data is available and therefore to be counted as coverage.*

A scoring function is needed to asses the similarity of two or more aligned sequences, as the ultimate goal is to optimise the alignment.

Definition 17 (Weight or scoring matrix) *A weight matrix (also called scoring matrix) is a 2-dimensional matrix that maps pairs of characters to a numeric value. The indices can be from any alphabet $\mathcal{A}$ and the elements represent a comparative score.*

Definition 18 (Score of two bases) *Given a weight matrix W and two bases a and b, the score of these two bases is given by $score(a, b) = W_{ab}$. In case W is symmetrical, then obviously $W_{ab} = W_{ba}$.*

An example for a simple weight matrix W is given in table 1. Using this weight matrix, the score of an entire column can be calculated.

Definition 19 (Column score) *The column score is calculated by the score of the permutation of all elements in a column. The score of a column having k elements is*

$$\sum_{j=1}^{k} \sum_{l=j}^{k} score(s_j, s_l)$$

As an alignment consists of several columns, a method is needed to asses the quality of the total alignment using the column scores as reference.

Definition 20 (Alignment score) *An alignment score is the sum of all column scores in the alignment. An alignment having $||\mathcal{S}_k^X||$ columns calculates its alignment score as follows:*

$$\sum_{i=1}^{||\mathcal{S}_k^X||} \sum_{j=1}^{k} \sum_{l=j}^{k} score(s_{ji}, s_{li})$$

Definition 21 (SCS alignment) *The alignment represented by a matrix of k given sequences for which the alignment score is maximal is called "Shortest Common Superstring" (SCS) alignment.*

The shortest common superstring problem with strings containing no errors is NP-hard (Armen and Stein (1995)). It is a reasonable assumption that the same problem with strings (reads) containing errors cannot be simpler although this has not been proven formally.

Definition 22 (Global alignment) *Global alignments optimise the score of an alignment over the full length of k sequences, the use of the ∇character is prohibited, the length of the alignment is $\overset{max}{k}(\mid \mathcal{S}\, k \mid)$. Global alignments – also known as Needleman-Wunsch alignments – are thus most appropriate when the sequences are known to be similar over their entire length.*

E.g.: The two sequences `TACGTCAATTAGATCTACT` and `CTACTGTA` could be globally aligned like this:

```
TACGTCAATTAGATCTACT
**C*T*A*CT*G*T**A**
```

although this does not seem to make much sense as these two sequences are obviously not very similar when considering the entire length. For cases like this, local alignment methods are much more appropriate.

Definition 23 (Local alignment) *Local alignment methods give the possibility to align k sequences over common subsequences. The ∇ character is needed to bring the sequences to the same length. Local alignment – also known as Smith-Waterman alignment – is typically used when when nothing is known in advance about the similarity of the sequences being aligned. It is also used when it is suspected that the sequences may overlap only partly.*

E.g.: The two sequences from the example above (TACGTCAATTAGATCTACT and CTACTGTA) can be locally aligned like this,

```
TACGTCAATTAGATCTACT...
..............CTACTGTA
```

Now that alignments have been defined, the problem is still to calculate the alignments for a set of k given sequences. This problem has been solved in the 1970s and 80s by Needleman and Wunsch for global alignments by applying dynamic programming algorithms. Dynamic programming computes a sequence comparison and sequence alignment by comparing shorter subsequences first, so that their score can be made available in a table for the next longer subsequence comparison. Smith and Waterman extended this technique to local alignments.

Dynamic programming algorithms that locate optimal alignments of two sequences are central techniques for the comparison of biological sequences and have been extensively studied for the case of $k = 2$ sequences.

Definition 24 (Optimal alignment by dynamic programming) *The best score for an alignment between k sequences is determined by calculating the k-dimensional alignment matrix $H_{|\mathcal{S}^2|-1,\ldots,|\mathcal{S}^k|-1}$. Starting with $H_{0,\ldots,0}$ and working forwards through the matrix in some topological order (line by line or row by row), the value of each cell $H_{i_1,\ldots,i_k}$ is calculated by scoring the predecessor cells with the value of the sequence to be compared and the "$*$" character by using the weight matrix W and take the maximum value from the computation.*

Equation 2.6 shows an example for a general 2-dimensional matrix H, see also figure 26 on page 58 for the banding specialisation of the algorithm.

$$H_{i_1,i_2} = max \left\{ \begin{array}{c} H_{i_1-1,i_2-1} + W_{\mathcal{S}^1_{i_1},\mathcal{S}^2_{i_2}} \\ H_{i_1,i_2-1} + W_{\mathcal{S}^1_{i_1},*} \\ H_{i_1-1,i_2} + W_{*,\mathcal{S}^2_{i_2}} \\ 0 \end{array} \right\} \tag{2.6}$$

To calculate the value of H_{i_1,i_2}, it is only necessary to know the contents of the three predecessor cells $(H_{i_1-1,i_2-1}, H_{i_1,i_2-1}, H_{i_1-1,i_2})$.

While the multiple alignment problem is NP complete in the number of sequences, there is a well-known $O(n^k)$ dynamic programming algorithm for computing alignments among k sequences of average length n (Smith and Waterman (1981)). The problem can be solved by simply extending the dynamic pro-

gramming recurrence for the basic problem: computing the values of matrix H in some topological ordering requires a total relative time of

$$O(\prod_{i=1}^{k} n^i) = O(n^k)$$

steps, where

$$n = max\{n^i : i \in [1, k]\}$$

While the extension to multiple sequences is conceptually quite straightforward, the obvious algorithm to compute the exact solution takes an amount of time exponential in k. It is therefore generally impractical in real world problems for $k > 3$ (Myers (1991)) sequences and – even when restricting the computation space by some means – certainly so for $k > 4$ (Gotoh (1993)).

2.2 Error types and rates in DNA sequencing

2.2.1 Errors of the data acquisition process

The DNA sequence gathered through experimental process is gained through an examination of the fluorescent-dye intensity signal that is output by automatic sequencing machines. Even with the newest generation of sequencers, raw sequence data obtained from them is – by all means – everything but trustworthy in its entirety. Inevitable artifacts degrade the quality of the sequences obtained and are caused by experimental as well as systematic factors. Chromatography is a chemical process and thus subject to stochastic and non-stochastic oscillations, which can cause sub-optimal signal quality. Errors in a determined DNA sequence can be caused by flaws in the translation operations of the electrophoresis signal or quirks that arose during the experiment itself. This becomes visible in the wide diversity of data that is obtained even when using a single chemistry type, let alone different ones: under- and over-oscillations of the signals, unseparated curves (compression artefacts), and signal peaks or dropouts are frequent. Incorrect signal analysis raises errors in the base calling process of the signals and constitutes a limiting factor in the automation of assembly processes.

Depending on a multiple factors – ranging from clone preprocessing and different dye-labelled terminators (or primers) to the type and length of gel used during electrophoresis (see also Lario et al. (1997); Rosenblum et al. (1997)) – the quality of the data gained along a single sequence substantially varies.

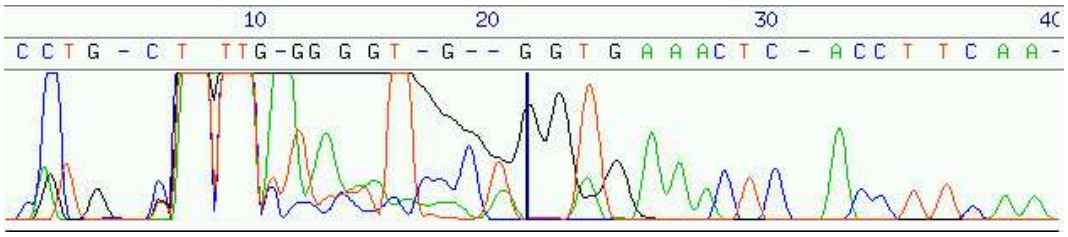

Figure 5: Example for bad quality data at the start of an electrophoresis gel or microcapillary trace. The clutter present at the very start of the trace is the result of instrument calibration.

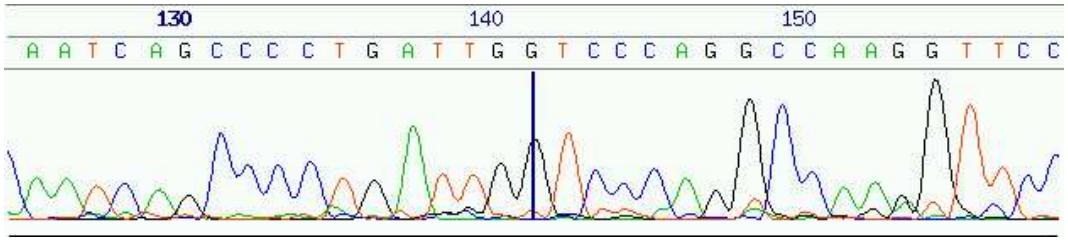

Figure 6: Good signal quality amidst a trace. The data has generally less than one error in 100 or even 1000 bases, although ambiguities do arise sometimes.

Current laboratory techniques can examine nucleotide sequence fragments between 600 and 1300 bases long. In most cases there is a typical curve of error rates to be observed (see Ewing et al. (1998); Richterich (1998); Lipshutz et al. (1994); Engle and Burks (1993, 1994)): it starts with a small stretch of low-quality bases (error rates between 3% and 8% for the first 50 to 70 bases, see figure 5) followed by a stretch of high quality data (error rates $\leq$1% to 2% for the following 600 to 800 bases in good traces[3], figure 6), although it is nevertheless possible for low-quality data to be present amidst a high quality stretch. As the signal-to-noise ratio degrades towards the end of of a trace, the base quality starts to deteriorate rapidly after a certain time with error rates ranging from 2% up to over 10% and to 20% in the tail of the sequence like shown in figure 7.

Basically, there are three types of errors introduced into the data by electrophoresis and subsequent base-calling: insertions, deletions and mismatches. Insertions are wrongly called bases at places were there are none, deletions are

[3]Mean length of useful sequences gathered on ABI 3730 machines at The Institute of Genomic Research (TIGR) in 2003, pers. communication from Bill Niermann (Investigator at TIGR) in April 2004

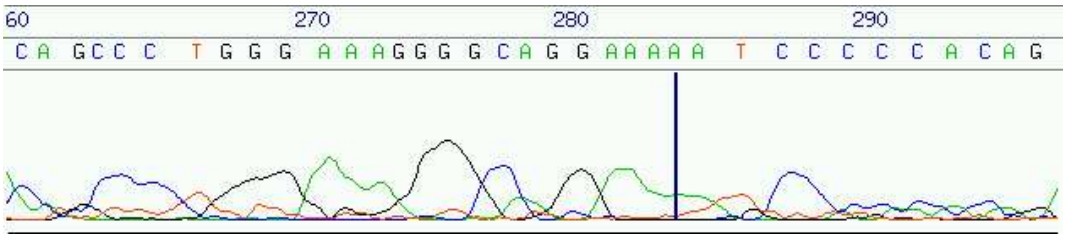

Figure 7: Example for bad signal quality towards the end of a trace. A low signal-to-noise ratio and unseparated curves cause high error rates.

bases that were not called in a sequence and mismatches represent wrongly called bases.[4] These types of errors can be reduced by using improved chemistry (Lario et al. (1997); Rosenblum et al. (1997), by applying image processing algorithms (Sanders et al. (1991)) or by using different base calling algorithms (Berno (1996)).

Having a viable numerical estimate of the base quality has been a major advance achieved by Ewing et al. (1998) and Ewing and Green (1998) who presented an improved base caller that also gives probability values for the called bases expressed as confidence estimates.

Definition 25 (Base error probability) *The value p with $0 \leq p \leq 1$ attached to a base call describes the probability with which the base caller has produced a wrong base call, where a value of 1 represents a certain wrong call.*

It must be noted that to give a correct estimate of the base probability, algorithm for computing the value of p often analyse a whole range of trace characteristics like shape, peak distances and other parameters gained through statistical analysis of a several million base calls.

The PHRED program was the first to transfer base error probabilities into a log-transformed value – also known as quality – to each called base.

Definition 26 (Base quality) *The quality of a base is assigned to be $q = -10 * \log_{10}(p)$ where q is the quality and p the error probability.*

Thus a quality of 40 would resort to the error probability of approximately 1 error in 10,000 bases.

[4] insertions and deletions are commonly referred to as *indels*

2.2.2 Errors due to biology

While errors due to the data acquisition process itself are problematic enough, the processes that precede it involve multiple steps of biological handling and add an additional level of complexity to the task.

One of the larger inconveniences is due to the method used to amplify small DNA clones which consist of adding an amplification vector and inserting the resulting construct into host cells (see also section 2.1). This vector/payload construct leads to an unpleasant consequence: any DNA sequence determined is likely to contain some part of the sequencing vector itself at the start – and sometimes the end – of the determined sequence. These stretches must of course be electronically removed as they do not belong to the target DNA that is to be sequenced. Unfortunately, the vector sequences are at the very front and rear of the sequence, which are the most error prone parts. Due to these errors, simple pattern matching algorithms often fail to recognise the sequencing vector completely.

The self-replication of the host-cells itself induces two further kind of errors: 1) errors in the base replication itself, which leads most of the time to small point mutations (SNPs, Single Nucleotide Polymorphisms) or 2) errors on a larger scale where the vector can "loose" its sequence payload, recombine with other plasmids or even recombine with some sequence parts of the host cell.

Definition 27 (Single nucleotide polymorphism) *A SNP (spoken: "snip") is a sequence variation in DNA or RNA where exactly one difference exists between otherwise two identical sequences. This difference can be either 1) a base-change, which is an exchange of a base* $\in \mathcal{A}^b$ *with another base* $\in \mathcal{A}^b$*, or 2) an "indel", which is an insertion or deletion of a single base in one of the sequences.*

Please note that SNPs can observed both because of errors in the base-calling process and because of real sequence differences.

While infrequent errors on the SNP level do not pose a particularly difficult problem, a non-recognised recombination of the vector with any type of sequence from the replication host (a contamination) leads to completely wrong results in the downstream sequence analysis. Although the awareness to the problem of contamination has increased in the last years in the scientific community, a quick search in public databases for example still reveals an uncanny number of *E. coli* or known vector fragment stretches in sequence clones that were taken from human or rat chromosomes.

Another common type of biological problem due to random recombination of the vectors with other sequences is called *chimera*.

Definition 28 (Chimera) *Chimeras are clones that contain adjacent DNA stretches that are normally located at two very different sites within a genome that is to be sequenced.*

Chimeras are formed due to spontaneous recombination during the self-replication of clones, the product of this recombination then hosts adjacent DNA subsequences that do not reflect reality of the original sequences. If chimeras are not recognised, this also can lead to wrong interpretation of the sequenced organisms.

2.3 Peculiarities of sequencing expressed sequence tags (ESTs)

Genomes are not the only nucleotide sequences in a cell that can be subject of sequence analysis. The second type of sequencing and subsequent assembly projects is called EST sequencing. This section gives a short overview on why and how it differs substantially in some critical points from genome assembly projects.

2.3.1 Biological background

The overall gene architecture in eukaryotic genomes and its recognition by computational means is complicated by the existence of the so-called exon-intron structures. This is exemplarily shown in figure 8. Genes are not positioned continuously at one location on the genome, but single parts of the genes (*exons*) are interrupted by intergenic regions (*introns*) which have the sometimes respectable size of several kilobases.

The cell however does recognise the correct gene structures on its genome as in the production path leading from the information contained in the genome to the final product – the protein – the first step is always the translation of the DNA into a messenger-RNA (mRNA) that conveys the information from the genome to the protein production facilities (ribosomes). The mRNA sequence of a cell is therefore of particular interest to scientists as it reflects both the genes that are currently being used (expressed) by the cell as well as the expression level of the expressed genes.

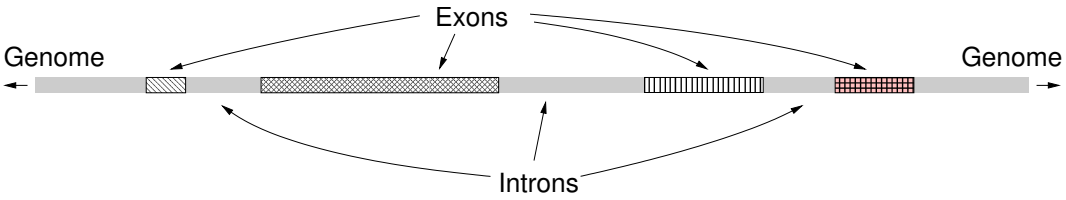

Figure 8: Simplified example for a gene architecture in eukaryotic genomes. A gene can be split in several parts (exons) that are located at different positions on the genomes. The intergenic regions (introns) can have several kilobases in length.

Definition 29 (Expressed Sequence Tag (EST)) *An Expressed Sequence Tag (EST) is a small portion of the DNA – usually a gene – that has been transcribed, i.e. expressed, into mRNA and then sequenced.*

The idea behind sequencing ESTs is the assumption that it is much easier to find genes in a whole genome when looking directly at the transcribed mRNA than to find them computationally in complex genome structures. Depending on the size of the gene and whether both sides of the genes are sequenced, the ESTs may or may not cover the entire gene. With current technology, genes up to 1200 to 1400 bases have a good chance to be completely covered when a two-sided sequencing strategy is used.

A further complication of eukaryotic genomes is given by the fact that the DNA is first transcribed into a pre-RNA that is composed of both exons and introns. In a subsequent step, the introns are then *spliced* (removed) away to form the mRNA. In this process which is not fully understood yet, different combinations of exons of a gene can also be removed. This leads to inherently different mRNA variants coming from one gene and subsequently also to different proteins. An example for this is shown in figure 9. Although this alternative gene splicing is now commonly seen as to be relatively frequent and not occurring haphazardly, the exact reasons and mechanisms for this are currently not completely elucidated. Citing Heber et al. (2002) “Recent studies indicate that alternative splicing is more frequent than previously thought and some genes may produce tens of thousands of different transcripts.”

2.3.2 Implications for assembly projects

Biology – and the phenomena encountered within – sets the boundaries for both for genome and EST sequencing projects. While most of the aspects addressed

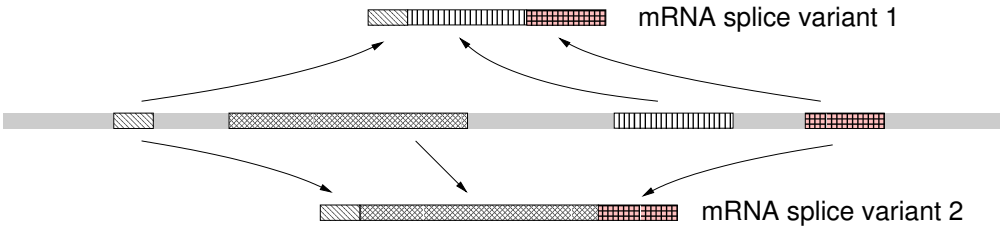

Figure 9: Simplified example for gene splice variations in eukaryotic genomes. During the splicing of the pre-mRNA into the final mRNA, introns and sometimes also some exons are removed from the pre-mRNA. The removal of exons leads to different gene transcript variations which are also called *splice variants* or *splices*.

earlier in this chapter like, e.g., data quality and coverage, are as important for genome assembly projects as for EST projects, two other characteristics influence the type of results or the quality of the computational EST assembly process in an important way: 1) the extremely wide range encountered in the abundance of mRNA transcripts of different genes, and 2) the additional complexity brought in by alternative splicing of genes.

Abundance of mRNA transcripts

Collecting mRNA samples for EST projects is one of the most critical tasks as the expression of genes varies over several orders of magnitude. In fact, genes are not evenly expressed in cells, neither through time nor tissue nor quantity. This differential expression is reflected in the abundance of specific mRNA transcripts in a cell.

> For example, *cytochromes* are a family of electron carrying proteins and constitute an important part of the respiratory chain in both prokaryotes (bacteria) and eukaryotes (higher organisms). Their central role in the metabolism makes them at the same time both ubiquitous in transcripts and closely related within gene families of cells. See Bruce et al. (1994) for more information.
>
> In contrast to that, the *spo0B* gene of *Bacillus subtilis* for example is required to initiate the so-called “stage 0” sporulation of the bacterium, but it needs to be expressed only at very low levels (Asayama et al. (1998)). Even for bacteria which initiate

the sporulation, only very few transcripts of this gene can be found.

Mostly due to cost constraints, not all of the tens of thousands of transcripts present at any time in a cell can be taken to a sequencing process. This in turn induces the reasoning that a naive sampling process – for example with the Monte-Carlo method – of the mRNA-transcripts present in a cell would therefore almost certainly sample many identical transcripts of highly expressed genes. Many transcripts with low abundance, however, could fall through the raster scan and not be represented at all in the samples.

In consequence, a biological "normalisation" process can be implemented to pre-select clones containing a representative subset of mRNA-transcripts. Although further adding to laboratory cost, this process is both needed to increase the discovery yield on genes (see also Schuler (1997)) and to alleviate the computational complexity for the assembly process (see also section 4.9.1 of this thesis). On the downside, normalised EST projects do not allow anymore quantitative studies of gene transcriptions.

Please refer to Klug and Cummings (1996) for more information on the selection of clones of the transcript normalisation process.

Splice variants

Splice variants add a further level of complexity to an assembly process. Variants that are present only in single copy numbers within clone libraries are computationally indistinguishable from chimeras. In fact, in this case they are completely indistinguishable even for human researchers when prior knowledge – like for example the underlying genome sequence – is not available. However, chimeras are created completely randomly while splice variants are not. A splice variant can thus be seen as validated once it is observed more than in a single sequence copy.

Hence, the additional complexity that splice variants bring into an assembly is mostly not related to computational methods, but to the interpretation of results that include one-time observations of certain splice variants.

3 Assembly strategies

"If it were easy, it would have been done already." (Solomon Short)

Referring to Dear et al. (1998), a "sequence assembly is essentially a set of contigs, each contig being a multiple alignment of reads".[1] Most unfortunately, the underlying problem of string assembly as a variant of the shortest common superstring problem has been shown to be NP-hard by Armen and Stein (1995) so that heuristic algorithms represent the only possibility approach this problem.

According to Chen and Skiena (2000), many assemblers have similar fundamental designs but differ substantially on important engineering issues. This chapter shortly summarises the most important strategies existing as of this writing. It then outlines the global concept developed in this thesis to address weak points in current strategies, that is, to tackle the goal of producing better consensus alignments by correctly discerning repeats set for this thesis.

3.1 Short overview of existing strategies

From the very beginning (Peltola et al. (1984); Staden (1984)) up to recent publications (Jaffe et al. (2003)), a number of different strategies have been proposed to tackle the problem. They range from simple greedy pairwise alignments – sometimes using additional information (Dardel (1985); Johnston et al. (1986)), sometimes using a whole set of refinements (Huang (1996); Huang and Madan (1999)) – to weak AI methods like genetic algorithms (Parsons et al. (1993); Notredame and Higgins (1996); Zhang and Wong (1997); Notredame et al. (1998)).

There are nowadays mainly three different existing approaches for the 'simple' assembly of sequences: (i) the iterative, (ii) the quality based one-step approach and (iii) Myers scaffolding approach. The first type of assembly is essentially derived from the fact that the data analysis and reconstruction approximation algorithms can be parametrised differently, ranging from very strict

[1] The term *contig* is derived from "contiguous sequence"

assembly of only the highest quality parts to very 'bold' assembly of even lowest quality stretches. An assembly starts with strictest parameters, having the output edited manually (by highly trained personnel) or by software and then the process is reiterated with less strict parameters until the assembly is finished or the parameters become too lax (see figure 10). The second approach has been made popular by the PHRAP assembler presented by Phil Green[2]. This assembler uses low and high quality sequence data from the start and stitches together a consensus by using the highest quality parts of an assembly as reference, giving the result to a human editor for finishing (Gordon et al. (1998)). For the third approach, Myers (1999) presented first results of the bridge building strategy for whole genomes, where contigs are arranged in a process called *scaffolding*. This strategy relies first on a high coverage sequencing of a genome with approximately 12-fold coverage in average, but ideally not less than 7-fold coverage. The clones are prepared in a mixture of different insert sizes between 2 kilobases (kb), 5kb, 10kb, 50kb and 100kb and additionally each clone is sequenced from both ends (also called dual-ended or double-barreled sequencing). The probable relationship of two sequences of a clone (a mate-pair), together with an hardware supported all-out comparison of each sequence against each other, is used to build a read framework of the contig before the sequences are assembled together. Contigs are subsequently arranged in a scaffold, their order is determined by mate pairs of reads which bridge the sequence gaps between the contigs. One important aspect is the fact that repeats in the sequences are masked prior to the assembly in a process called *fuguization*[3] by Bailey et al. (2001).

A common characteristic to most existing assemblers from category (i) and (ii) is that they rely on the quality values with which the bases have been attributed by a base caller. Within base-calling process, an error probability is computed by the base caller to express the confidence with which the called base is thought to be the true base. The positive aspect of strategies relying on base qualities is the possibility for assemblers to decide in favour of the best, most probable bases when discrepancies between reads occur. The negative aspects of current base callers are 1) their inability to write confidence values for optional, uncalled bases at the same place and 2) the fact that base probability values sometimes cannot be computed accurately enough.[4] The scaffolding

[2]PHRAP as acronym for PHils Revised Assembly Program, see also http://www.phrap.org/

[3]this refers to the rather compact genome of the puffer fish (*Fugu rubripes*) which is largely devoid of large copy repeats.

[4]e.g. for new electrophoresis methods

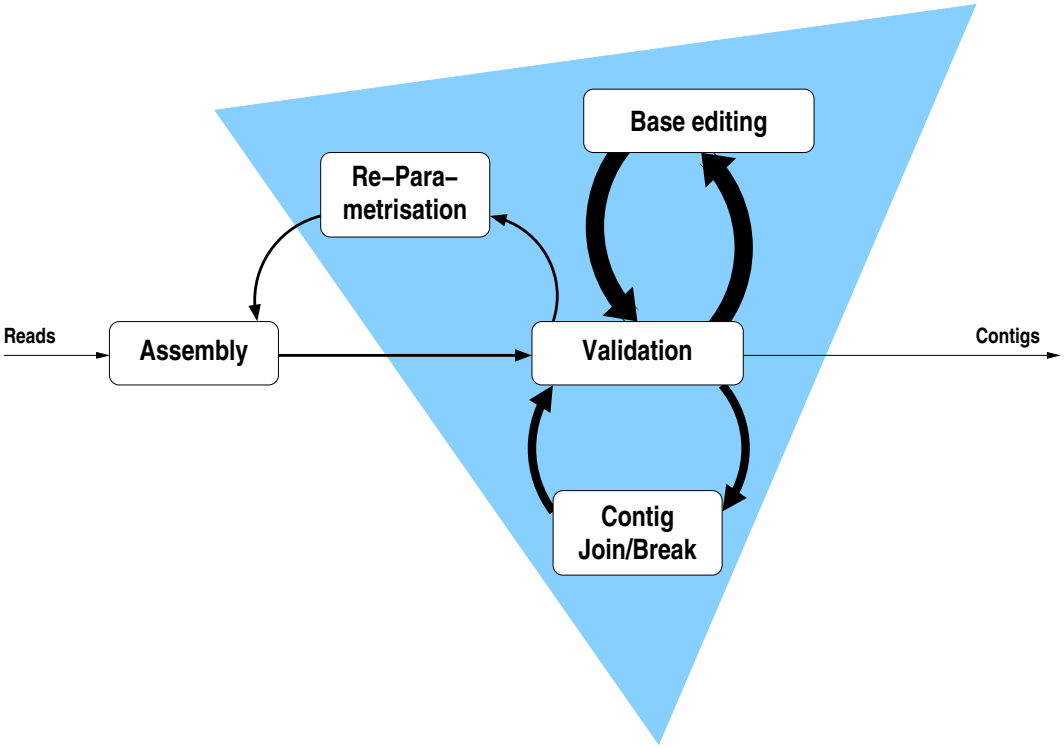

Figure 10: Conventional assembly has a high level of human interaction (area in the triangle). The thickness of the arrows represents the relative number of times a certain action has to be performed.

approach relies on hardware acceleration of certain operations[5] and the problem of repetitive sequences has empathically been taken out from the assembly algorithm, leaving only 'clear' and 'relatively simple' reads to assemble.

A special case of assemblers are constituted by "assemblers using assemblers": programs that try to address the shortcomings of different assemblers by combining them. Chen and Skiena (2000) presented a case study in genome-level fragment assembly in which they compared design, issues and results of 4 different assemblers. They found that the contigs formed by the different assemblers were sufficiently different to justify running multiple assemblers for comparison in the finishing stage of a project. Wang et al. (2002) demonstrated with `RePS` (REpeat-masked Phrap with Scaffolding) a package that constructed contig scaffolds using clone-pair information and assemblies made with the `phrap` program in the finishing phase. Other finishing approaches like the

[5]Specialised hardware for this type of operations starts approximately at EUR 500,000

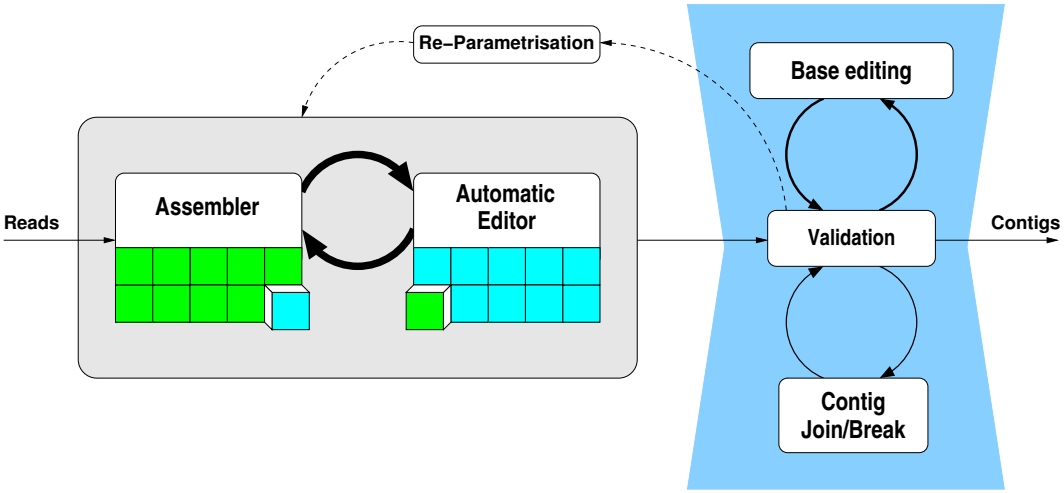

Figure 11: Using an integrated assembly and editing concept transfers a significant portion of the work to automated algorithms, leaving only non-standard problems to the human finisher (area in the hexagon).

prokaryotic genome assembly assistant system (PGAAS) developed by Yu et al. (2002) try to confirm the order of contigs and then fill remaining gaps through peptide links obtained by searching contig ends against protein databases with BLASTX.

3.2 Developing a new strategy

The development of a new type of assembler was started in 1997 at the DKFZ Heidelberg (German Cancer Research Center). The assembly strategy was conceived from the very beginning to combine and substantially extend the strengths of both traditional approaches that existed at this time ((i) and (ii), mentioned in the previous section) and reproducing assembly analysis strategies done by human experts (see figure 11). An important criterion in the design of this assembler is the quality aspect of the final result: the assembler starts only with the stretches of DNA sequences marked as 'high' or 'acceptable' quality, from which it selects the best to start an assembly. These high confidence regions (HCR) ensure a reliable base and good building blocks during the assembly process. Lower quality parts (low confidence regions, LCR) can be used later on if needed.

3.2.1 Tackling misassemblies

A multi-phased concept has been worked out to have the assembler perform the difficult task of optimal shotgun sequence alignments. Different authors have proposed different sets of acceptance criteria for the optimality of an alignment (an overview is given by Chan et al. (1992)). Traditionally (Myers (1995)), "the objective of this [assembly] problem has been to produce the shortest string that contains all the fragments as substrings, but in case of repetitive target sequences this objective produces answers that are overcompressed."

Overcompression can be compensated in part by using special cloning and sequencing techniques like taking different clone template insert sizes and using this information for a pre-assembly coverage analysis like it was done for the sequencing of the human genome by Celera company (Venter et al. (2001)). However, not all genomic or EST sequencing projects use this technique. Furthermore, although EST clone library projects can use dual end clone sequencing techniques, this information is of little help as mRNAs rarely surpass a few thousand bases in length. To make the matter worse, certain genes – or even complete gene families – in non-normalised EST clone libraries can be disproportionally expressed and sequenced more often than others (e.g. cytochromes). This results in more mRNA clones of these genes and thus more sequence fragments of these sequences.

Searching alignments by suffix trees and analysing the trees to deduce possible repeat resolving strategies was discussed several times in the past. But, as Ma et al. (2002) noted, suffix tree approaches suffer from two main problems: they are not efficient in handling mismatches and suffer from large space requirements. The later problem is gradually being addressed by more efficient algorithms like MUMmer2 and NUCmer (Delcher et al. (2002)) and – trivial but still important to note – technology advances that nowadays allow relatively inexpensive computers to have several gigabytes of memory.

Another method to handle repeats consists of reducing the fragment assembly problem to a classical variation of the Eulerian superpath problem (see Pevzner and Tang (2001)) which was extended in Pevzner et al. (2001) to use template insert size information[6] generated by clone-end sequencing. Still other methods like the one by Otu and Sayood (2003) used a very different strategy with divide-and-conquer approach by computing average mutual overlap information in their fragment assembly algorithm to simultaneously solve the overlap, layout and consensus phases of an assembly.

[6] which they call "double-barreled data"

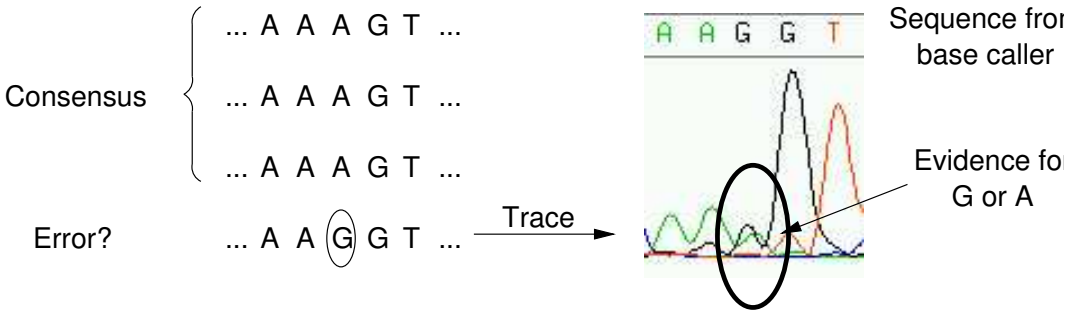

Figure 12: Simple example for an explainable assembly discrepancy under the prerequisite that the consensus is right. The mismatching G base in the offending read could be edited toward the consensus by looking at the trace evidence.

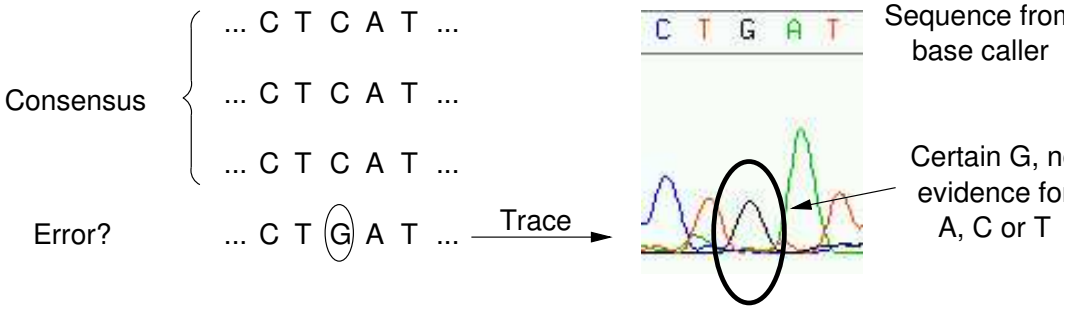

Figure 13: Simple example for an unexplainable assembly discrepancy under the prerequisite that the consensus is right. There is absolutely no evidence in the trace of the offending read to edit the mismatching base toward q consensus.

Lee et al. (2002) introduced heuristics using partial order graph to use pairwise dynamic programming for multiple sequence alignments to resolve problems posed by repetitive sequences and overcompression. One shortcoming of this approach was that it turned out to be not immune to order-dependency effects in the constructed alignments.

In the end, all this means that the overcompression criterion is only partly useful for tracking misassemblies in genomic sequencing and not useful at all for detecting misassemblies in EST sequencing projects. As a result to this, the strategy conceived was the one of the *least number of unexplainable errors* present in an assembly to be optimal.

3.2.2 Focussing on observable data

Unexplainable error are errors in an assembly where evidence in the trace data does not permit to correct wrongly called bases of the offending read toward a consensus. Figure 12 shows a simple case of an explainable error – which almost certainly will be edited away – while figure 13 pictures an unexplainable error in an assembly. Discrimination possibilities between base calling errors and real discrepancies is the main difference to using simple quality values as discerning criterion only: quality values do not offer alternatives to the called base. Regardless of the possible quality of a base, trace data always offers this possibility.

So, a novelty of the new assembly strategy is that the assembler has been combined with some capabilities of an automatic editor. Both the assembler and the automatic editor are separate programs and can be run separately, but the tasks of assembly and finishing can be viewed to be closely related enough for both parts to include routines from each other (see also Chevreux et al. (1999); Pfisterer and Wetter (1999)). In this process, the assembler gains the ability to perform signal analysis on partly assembled data which helps to reduce misassemblies especially in problematic regions like standard repeats – e.g. ALUs, LINEs, MERs, REPT etc., see Bruce et al. (1994) for more information on these – where simple base qualities alone could not help. Analysing trace data at precise positions with a given hypothesis 'in mind'[7] is a substantial advantage of a signal analysis aided assembler compared with a 'sequential base caller and assembler' strategy, especially while discriminating alternative solutions during the assembly process. In return, the automatic finisher gains the ability to use alignment routines provided by the assembler.

3.2.3 Pattern analysis

Another important improvement of the developed assembly strategy that differentiates it from simpler alignment algorithms is the possibility to perform post-assembly validation checks on the resulting contigs. These checks are run to find misassembled reads and remove them from the assembly. Most often, these misassemblies are due to repetitive sequences that might differ in only one or two percent on stretches as long as several hundred bases. This is something almost impossible for alignment algorithms to discern correctly as these

[7] e.g. 'could the base G at position 235 in read 4 be replaced by a A?' (because the overall consensus at this position of the other reads suggests this possibility)

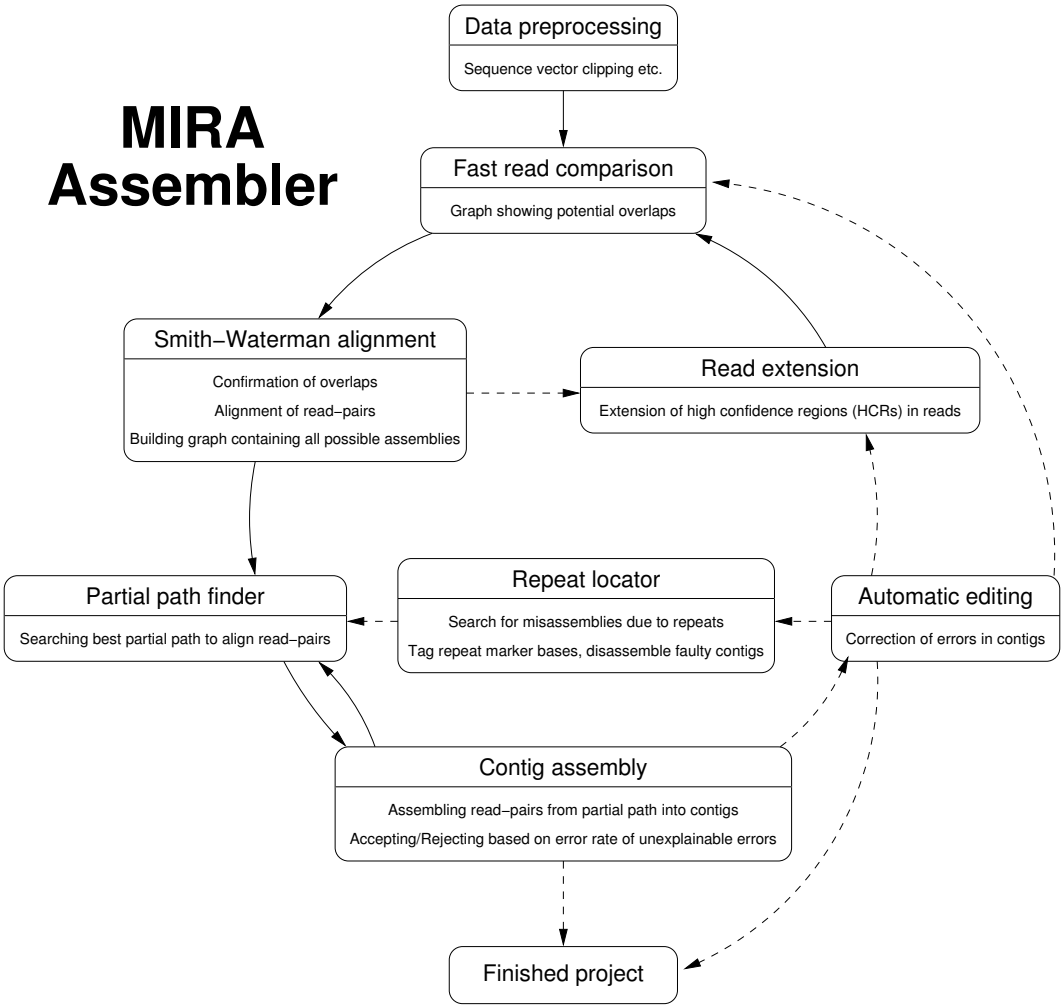

Figure 14: Phases of a MIRA assembly cycle. Plain arrows show imperative pathways, dashed arrows denote optional pathways.

algorithms are specifically designed to work with slightly erroneous data. So, repetitive sequences still might get wrongly assembled together a first time. Fortunately, these repeats produce error patterns in an assembly that can be searched for and recognised by pattern recognition algorithms. These algorithms are helped by the fact that the automatic editor will have edited away most of the trivial base calling errors. The remaining error patterns can be compared to typical repeat error patterns and the offending reads can be tagged as repetitive and marked for a subsequent assembly iteration.

Figure 14 depicts an approximate overview of the developed strategy as well

as the different phases involved while assembling a project:

1. The reads constituting an assembly project are preprocessed by external programs to perform different refinement steps to the original read data, e.g. sequencing vector clipping, standard repeat tagging, quality clipping etc. A multitude of programs is available, each of them being very specialised for a certain task.

2. The high confidence region (HCR) of each read is compared with a quick heuristic algorithm to the HCR of every other read to see if it could match and have overlapping parts (these are the 'DNASAND' and 'ZEBRA' filter). All the possible overlaps form one or several initial building graphs.

3. The reads in the initial building graphs which could have overlaps are being reviewed with an adapted Smith-Waterman alignment algorithm (banded version). Obvious mismatches are rejected and removed from the initial building graph, the accepted read-pairs are inserted into one or several alignment graphs. These alignment graphs define all the assemblies that are possible with the given reads.

4. Optional pre-assembly read extension step: the assembler can try to extend HCRs of reads by analysing the overlap pairs from the previous alignments. This can help to elongate reads which were cut back too much by conservative quality clipping mechanisms during pre-assembly preprocessing. The confirmation of a base sequence by two similar reads combines the advantage of single read quality clips and coverage security.

5. A contig is assembled by building a preliminary partial path through the alignment graph and then adding the most probable overlap candidate to a given contig. Contigs can reject reads if these introduce to many unexpected and high profile errors in the existing consensus. Errors in regions known as dangerous – for example tagged standard repeats like ALUS and REPT, or possible repeat marker bases (PRMB) found in previous iterations – get additional attention by performing simple signal analysis when alignment discrepancies occur.

6. Contigs can be be optionally analysed and corrected by an incorporated version of an automatic editor (EdIt). This editor analyses fault regions in contigs and corrects base call errors (and alignment errors resulting from these) by analysing the underlying trace signals of the reads and calculating probabilities with respect to the coverage of the contig.

7. Also optionally, long term repeats that were misassembled can be searched for. The assembler looks for typical misassembly patterns provoked by repeats (like mismatch columns). Bases that are dissimilar in the different repeats get tagged as possible repeat marker base (PRMB) to help the assembly algorithm in subsequent iterations and then the wrongly assembled contigs get dismantled for further reassembly.

8. The resulting project is written out to standard file formats for further post-assembly processing.

As can be seen, the overall structure of the assembly strategy chosen – i.e. the iterative nature of its design and the steps concentrating on high quality parts first – reflects the needs imposed to it by the aims set for this thesis: identifying repeats to avoid misassemblies and decrease the error rate of the final consensus sequence. Each subpart of the assembly strategy is embedded into the overall scheme, which in turn means that, sometimes, algorithms show unexpected – and mostly undesired – side effects when used within the whole system. The next chapter therefore goes into the details of those algorithms, explains the inner working mechanisms of each subpart and how they are connected to each other.

4 Methods and Algorithms

> *"Half of being smart is to know what you're dumb at."* (Solomon Short)

This chapter describes in detail the different steps of the assembly design that was outlined in the previous chapter. The chain of reasoning that led to certain algorithms is also exposed, together with examples coming from real world projects when appropriate.

4.1 Data preprocessing and input

Strictly speaking, data preprocessing does not belong the actual assembler as almost every laboratory has its own means to define 'good' quality within reads and already use existing programs to perform this task.[1] But as this preprocessing step directly influences the quality of the results obtained during the assembly, defining the scope of the expected data is desirable. Moreover it can explain strategies implemented to eventually handle incorrectly preprocessed data.

The most important part in the sequenced fragments (apart from the target sequence itself) is the sequencing vector data, which will invariably be found at the start of each read and sometimes, for short inserts, at the end. These parts of any cloned sequence must imperatively be marked or removed from an assembly as these would contaminate the "real" sequence that is to be determined. Programs like `LUCY` presented by Chou and Holmes (2001) go a great length to remove vector sequences, perform quality trimming and even compare the sequence produced by several different base-calling programs from the same chromatogram file to define what they call the "final clean range" (or high confidence region, HCR, in terms of this thesis). In analogy to the terms

[1] For example, quality clipping, sequencing vector and cosmid vector removal can be controlled by the PREGAP4 environment provided with the `GAP4` package (Bonfield et al. (1995b); Staden (1996); Bonfield and Staden (1996)) or the `LUCY` program, parts of these tasks can also be done with `cross_match` provided by the PHRAP package or other packages like, e.g., PFP from Paracel (Paracel (2002a)).

used in the GAP4 package, this thesis will refer to marked or removed parts as 'hidden' data (Staden et al. (1997)), other terms frequently used are 'masked out' or 'clipped' data.

Errors occurring during the base-calling step or simply quality problems with a clone can lead to more or less spurious errors occurring in the gained sequences. These in turn sometimes interfere with the ability of preprocessing programs to correctly recognise and clip the offending sequence parts. Therefore the `mira` and `miraEST` assemblers developed during this thesis incorporate a number of routines across all steps of the assembly that 'save' sequences that were incorrectly preprocessed. While this section gives a brief algorithmical overview over implemented methods within the scope of this section, please refer to the program documentation in appendix A for a full description of all available options. The routines that were implemented and that can be used by the assembler are:

1. Standard quality clipping routines:
   Clipping is done with a modified sliding window approach known from literature as in Chou and Holmes (2001); Staden et al. (1997), where a window of a defined length l is slided across the sequence until the average of the quality values attains a threshold t. Usual values for this procedure are $l = 30$ and $t = 20$ when using log-quality values as described in section 2.2.1. An additional backtracking step is implemented to search for the optimal cutoff-point within the window once the stop-criterion has been reached, discarding bases with quality values below the threshold. This is performed from both sides of the sequences.

2. Pooling masked areas at sequence tails:
   Parts of sequences that were masked (X'ed out) by other preprocessing programs sometimes contain small areas between 1 and 30 nucleotides of non-masked characters within the masked area due to, e.g., low quality data or the usage of slightly differing sequencing vectors. If requested, the assembler will merge such masked areas when the non-masked sections do not exceed a given length. E.g, the sequence `XXXXATXXXXXXXXXX...` becomes `XXXXXXXXXXXXXXXX...`

3. Clipping of sequencing vector relicts (while differentiating them from possible splice variants:
   This is done by generating hit/miss histograms of subsequence alignments between all the sequences. In an alignment of two sequences, it is nor-

mally to be expected that two neighbouring subsequences of one sequence should also be neighbouring to each other in the other sequence. If this is the case, then a "hit" is counted, if not, a "miss". The good quality middle parts will have a high ratio of consecutive subsequence alignment hits versus "unexpected" misses within a sequence histogram. Meanwhile, vector leftovers at the end of sequences will have a very low ratio of hits vs. misses. The beginning/end of such vector fractions is marked by a relatively sharp change in the ratio – a "cliff" – which can easily be detected. Unfortunately – in EST projects – different splice variants of eukaryotic genes present the same effects within histograms so that hit/miss ratio changes are searched for only within a given window at the start and end of the 'good' sequence parts (usually between 1 and 20 bases) to only catch such vector relicts present there.

4. Uncovering and tagging of poly-A and poly-T bases at sequence ends in EST projects:
   Unlike other specialised transcript assemblers like `pta` (Paracel (2002c)), the algorithms of the assembler developed in this thesis differentiate between different splice variants present in an assembly. They therefore include poly-A / poly-T bases when aligning EST sequences. The assembler will recover those areas by comparing masked sequences with the original counterpart and uncover exactly the poly-A/T stretches present at the end of the sequences by a simple but fault-tolerant base-by-base comparison algorithm. These stretches will furthermore be tagged with assembly-internal meta information to help the algorithms in the splice detection task.

A *high confidence region* (HCR) of bases within every read is selected through quality clipping as an anchor point for the next phases. Existing base callers (ABI, PHRED, TraceTuner and others) detect bases and rate their quality quite accurately and keep increasing in their performance, but bases in a called sequence always remain afflicted by increasing uncertainty towards the ends of a read. This additional information, potentially worthful, can nevertheless constitute an impeding moment in the early phases of an assembly process, bringing in too much noise. It is therefore marked as *low confidence region* (LCR) for cautious use in the assembly process.

The following list shows the type of data the assembler will work with, any of which can be left out (except sequence and vector clippings) but will reduce the efficiency of the assembler:

1. the initial trace data, representing the gel electrophoresis signal;
2. the called nucleic acid sequence;
3. position specific confidence values for the called bases of the nucleic acid sequence;
4. a stretch in each sequence marked as HCR;
5. general properties like direction of the clone read and name of the sequencing template etc.;
6. special sequence properties in different regions of a read (like sequencing vector, known standard repeat sequence and known SNP sites etc.) that have been tagged or marked.

4.2 Read scanning

A common start for any assembly is to compare every read with every other read (and its reversed complement) using a fast and fault-tolerant algorithm to screen all reads and detect potential overlaps.

As Anderson and Brass (1998) denoted, "the effectiveness of database search methods can also be illustrated by the distance the query sequences can be evolved before the search method no longer finds the homologous sequence in the database."

In this section, the problem of finding similar subsequences including errors is formalised and existing algorithms are presented. Then two fast scanning algorithms – named DNA-SAND and ZEBRA – and their theoretical effectiveness are presented.

4.2.1 Formalising the problem

Let $\mathcal{A}$ be the alphabet of an (unknown) true sequence T – which has passed through the electrophoresis and base-calling process – and of which we now have a representation $\mathcal{S}^x$ in the alphabet $\mathcal{A}^x$.

In the notation of regular expressions: let a true sequence T of length n_t be denoted by $T = t_1 t_2 ... t_n$ (with $t_i \in \mathcal{A}$) and let the representation S be denoted by $S = s_1 s_2 ... s_m$ (with $s_i \in \mathcal{A}^x$). The three types of errors in $\mathcal{S}^x$ of length m can be modeled as follows (assuming that $a \in \mathcal{A}^x$):

– insertions into S with the operator * meaning zero or more occurrences of the preceeding character $a \in \mathcal{A}^x$

$$S = a^*t_1a^*t_2...a^*t_na^* \tag{4.1}$$

– deletions from S with the operator ? meaning zero or one occurrence of the preceeding character

$$S = t_1?t_2?...t_n? \tag{4.2}$$

– mismatches in S with $(t_i|a^{a\neq t_i})$ meaning a choice between t_i and a character from $\mathcal{A}^x$ which is unequal to t_i

$$S = (t_1|a^{a\neq t_1})(t_2|a^{a\neq t_2})...(t_n|a^{a\neq t_n}) \tag{4.3}$$

Typically, S does not contain only one type of errors but all three. The formally correct regular expression for this is

$$S = a^*(t_1|a^{a\neq t_1})?a^*(t_2|a^{a\neq t_2})?...a^*(t_n|a^{a\neq t_n})?a^* \tag{4.4}$$

which can safely be shortened to

$$S = a^*t_1?a^*t_2?...a^*t_n?a^* \tag{4.5}$$

during a string comparison process as

$$(t_i|a^{a\neq t_i}) = a \tag{4.6}$$

so that in the case of a mismatch, the preceeding a^* expression in front of $t_i?$ would already catch any error.

In plain words: each base of the true genomic sequence T one wants to know may or may not be present in its representation S; additionally, it may or may not be surrounded by one or more artefact bases in S.

Finally, the orientation of S compared to T is unknown. S can be in the same direction as T, but it also can be in the reverse complemented direction. In this case, the bases in S must be reversed and complemented before a search can take place. Defining $\bar{t} \in \mathcal{A}^x$ as the complementary base of $t \in \mathcal{A}^x$, it is very well possible that

$$S = a^*t_1?a^*t_2?...a^*t_n?a^* \tag{4.7}$$

but also that

$$S = a^*\bar{t}_n?a^*\bar{t}_{n-1}?...a^*\bar{t}_1?a^* \tag{4.8}$$

The problem is now to design fast algorithms that can recognise patterns in sequences containing all errors described above.

4.2.2 Present algorithms

Several algorithms developed for string comparisons in computer science have been adapted to the needs of bioinformatics during the past decades, as searching large protein or nucleic acid databases has always been a critical application in biosciences[2]. There is one main differentiation to be made: depending on the type of search, searching in protein databases sometimes requires algorithms completely different from those for searching in nucleic acid sequences. Errors occurring in the nucleic acid sequence will cause – in the case of indels – a frame-shift on the protein level, subsequently changing all the following amino acids. At the nucleic acid level, an indel just inserts or deletes a base without changing the following bases. The following text focuses on the special problem of searching patterns in nucleic acid sequences.

In many cases, algorithms that work well for normal texts or in signal theory are not well suited for DNA sequences.

Multiple sequence alignments techniques that use the fast Fourier transformation (FFT) have been much less studied than other string and character based algorithms, although FFTs are a widely used method in signal data computation. A reason for this could be that Felsenstein et al. (1982a) first published results of using simple variants of this method that cannot allow for insertions and deletions during matching and were not really satisfactory, but given the drastic increase in computation power in the last 20 years, Rajasekaran et al. (2002) and Katoh et al. (2002) were able to refine the approach and achieve good results.

The 'optimal mismatch' algorithm presented by Sunday (1990) relies on alphabet size and character frequencies in a text to achieve a searching speed which is in most cases well below $O(n)$. But in most nucleic acid sequences gained from sequencing, the size of the alphabet is only 4 characters (plus one wildcard) and the frequencies of the bases are almost equally distributed. This noticeably reduces the effectiveness of the algorithm. Although it is relatively easy to implement single character wildcards like "N" (Gronek (1995b)) or the

[2] see also Myers (1991)

IUPAC code, it still cannot allow for indels.

Algorithms based on adapted Boyer-Moore string searching methods (Boyer and Moore (1977)) have also been implemented and used in the bioinformatics field (Prunella et al. (1993)), but these require a comparatively slow sequence preprocessing step. Giladi et al. (2002) presented an algorithm based on tree-structured index of k-tuple windows in vector space, with tree-structured vector quantisation which worked well only in balanced trees.

The dynamic programming algorithms for global alignments introduced by Needleman and Wunsch (1970) and later refined for local alignments by Smith and Waterman (1981) are the most suited methods for detecting – even partial – overlaps between sequences. They allow insertions, deletions and mismatches in both sequences and some algorithms, e.g. Guan and Uberbacher (1996), allow frameshift errors in protein sequences. Unfortunately, the run-time complexity of unbanded versions of those algorithms – this means, without prior knowledge or assumptions about the alignment of the sequences – is $O(n * m)$, with n and m being the length of the two sequences. This makes the run-times unacceptable for most screening procedures when special hardware acceleration is not available.

Obviously, the problem of indels occurring in a sequence is one of the hardest ones to be solved. This is the reason why almost all programs available today for heuristic string searching are based on the word-based method introduced by Wilbur and Lipman (1983). The most known representatives for this search class are the FASTA family of programs (Pearson (1998)) and the BLAST family (Altschul et al. (1997)). They use (sometimes inverted) indices and the fact that matches between a search pattern and a sequence have at least one word of several error-free bases in common (Huang (1996); Schuler (1998)). Methods for allowing mismatches and indels (Grillo et al. (1996)) or basing on probabilistic interpretation of alignment scoring systems (Bucher and Hofmann (1996)) have also been devised, but are generally impracticable for more than one or two errors. The newer SSAHA algorithm by Ning et al. (2001) used a new way to combine position specific hashing algorithms using non-overlapping k-tuples and hit sorting algorithms. Recently, Kent (2002) developed the BLAT tool which performs "stitching and filling in" of multiple exact k-tuple matches to produce a fast aligner, but notes that the alignment strategy becomes less effective for sequence identity below 90%.

There is a major drawback of these methods: for finding matches in regions with high error rates, the specificity of the search has to be decreased to a point where true-positive matches are (by far) outnumbered by spurious, false-

positive matches. This is often the case at the ends of shotgun-sequenced data.

4.2.3 DNA-Shift-AND algorithm

An interesting algorithm for searching patterns allowing errors is the Shift-AND algorithm presented in Wu and Manber (1992b) and Wu and Manber (1992a), based on previous work of Baeza-Yates and Gonnet (1992). The alphabet Σ in which to search can be freely chosen, character frequencies of single letters in the text do not impede the speed of the algorithm. Searching for patterns in a text is abstracted to the problem of iterating bit arrays according to a predetermined schedule (Gronek (1995a)) stored in transition tables. These arrays represent different states of search results of a pattern against a text and can thus be held in machine registers if the length of the pattern does not exceed 32 or 64 characters, depending on the processor type.

The Shift-AND algorithm searches a pattern of configurable length in a sequence allowing for a configurable number of errors. Since most BLAST search actions in nucleotide sequences are performed with a word length well below 32 characters, the length of patterns has been limited to 32 bases. This allows to write the algorithms in a way that fits to the architecture of virtually all presently available machines. The algorithm has been implemented in a straight-forward way based on the publications of Wu and Manber (1992b) and Gronek (1995a).

Due to limitations of the pattern length discussed above, the complexity of the DNA-Shift-AND (DNASAND) algorithm as it has been realised is $O(cn)$; where c is the number of errors allowed in a pattern and n is the length of the text to be searched. The algorithm is nonetheless able to recognise regular expressions as defined in equations 4.7 and 4.8, as long as the number of correct bases and the number of errors does not exceed the length of a pattern to be searched. For example, searching for a DNA pattern of length 20 and allowing 3 errors will find any occurrence of the pattern in a sequence with 17 correct bases and 3 errors, 18 bases and 2 errors, 19 bases and 1 error or the complete pattern without errors. Each error might be either an insertion, a deletion or a mismatch at any location. This metric to measure the similarity between two strings is called the Levenshtein distance when each insertion, deletion or mismatch is defined to count as one error.

Implementation of the DNASAND sequence filter

The function of a DNA sequence overlap filter is to find as many potential overlaps between sequences as possible. A key point in this requirement is the ability to find – besides long overlaps with low error rates expected to be recognised by any algorithm – even weak overlaps between any two sequences. Weak overlaps are characterised either by only a small number of bases from each sequence overlapping each other or by overlapping bases having a high error rate; or both. Of all the string comparing and string alignment algorithms known up to now, only the Smith-Waterman dynamic programming algorithm has the possibility to find even weak overlaps. Unfortunately it has too high processor requirements to be used as a fast filter. The DNASAND algorithm is a good substitute for this task.

A reasonable assumption for determining overlaps would be that two sequences S_1 and S_2 might overlap if they both have a succession of k bases in common (a *common subsequence*). As the probability of the sequences having errors is high enough, it is wise to extend the assumption: S_1 and S_2 might overlap if they have a common subsequence k bases long with at most l errors in it. The problem is now to find the common subsequence.

As the problem is to find even weak overlaps, a first approach would be to take the first k bases of S_1 as pattern P_{1s} and search for it in S_2 (allowing l errors). The localisation of the two sequences relative to each other is unknown, it is therefore necessary to do the same the other way round: take the first k bases from S_2 as pattern P_{2s} and – allowing l errors – and search for it in S_1. Doing this, every overlap combination of two sequences having the same orientation (both forward or in reversed complement direction) and at least k bases in common with at most l errors will be found.

The next difficulty is the fact that the orientation of the sequences to each other is unknown: they both can have the same direction – for the sake of simplicity, it is indeed of no consequence to the filter whether the sequences are both in forward or both in reversed complement direction, as long as it is the same orientation – or they can have opposite directions where one of the sequences overlaps with the reverse complement of the other sequence[3]. To find overlapping sequences that do not have the same direction, the pattern P_s taken from a sequence S_i must be searched in S_j and the reversed complement pattern $\overline{P}_s$ from S_i must be searched in S_j.

[3]It is again of no consequence which sequence is in reverse complement direction to the other, as both will be searched with the reverse complement pattern of the other one.

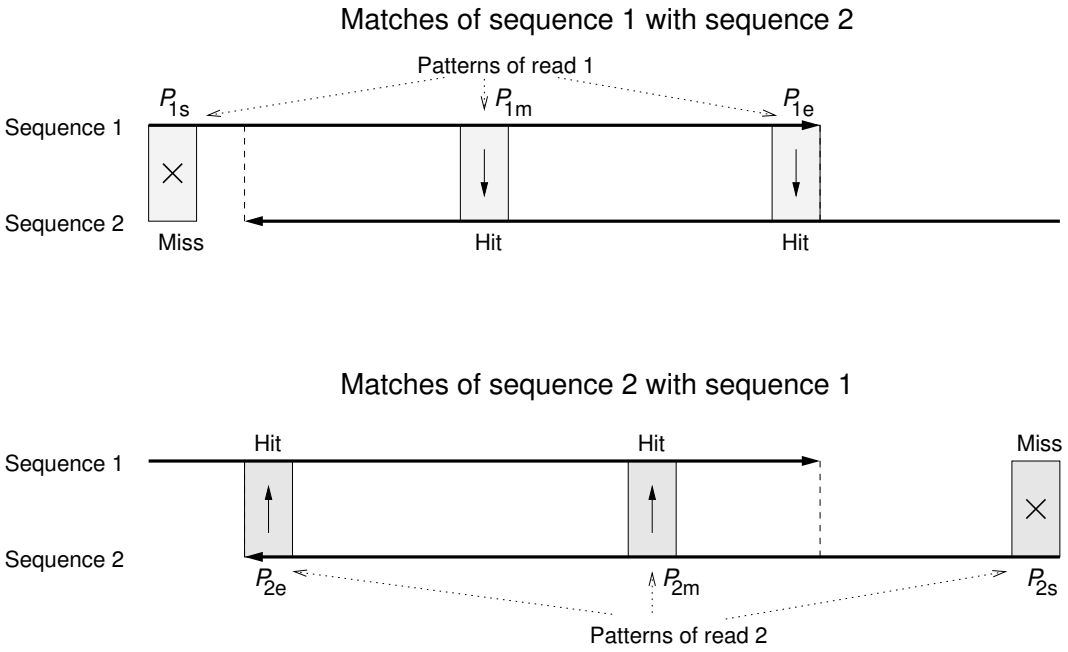

Figure 15: Mode of operation for DNASAND. When searching for overlaps for two sequences, patterns from the start, mid and end of each sequence are taken and slided across the other sequence with the Shift-AND algorithm. The same is done for the reverse of the sequences (not shown).

Doing this, there is still one further possibility for two sequences to overlap that would not be found using this method: when two sequences having opposite directions overlap at the end.The solution to this problem is to take not only the start of a sequence to search as pattern P_s or reversed complemented pattern $\overline{P}_s$ in any other sequence, but also to take the end of a sequence as pattern P_e or as a reversed complemented pattern $\overline{P}_e$ and search for it in each other sequence.

Summarising the above considerations: to determine if a sequence S_i might overlap with any other sequence S_j, two patterns – P_{is} at the start and P_{ie} at the end – and their reversed complements – $\overline{P}_{is}$ and $\overline{P}_{ie}$ – are taken from S_i. These patterns are compared with the modified Shift-AND algorithm to the whole sequence S_j and will show if the patterns match anywhere within S_j. If this is the case, the filter can assume that there is a potential overlap between S_i and S_j (or between $\overline{S}_i$ and S_j).

However, there is still one problem to consider: the filter relies on the potentially poorest data – present at the end of sequences – to find even the weakest

overlaps. It is always possible that the ends of the sequences contain too many errors to act as reasonable patterns, which, in turn, can cause the filter not to recognise potential overlaps. Some prerequisites have to be met for a situation like this to happen: the ends of the sequences are polluted with foreign DNA like non-removed vector sequence, or they contain an unusually high level of errors which has not been recognised by programs performing quality-clipping.

It is, consequently, wise to take preventive measures for cases like these. A third pattern and its reversed complement to be searched for are hence introduced in each sequence. These patterns P_{im} and $\overline{P}_{im}$ are taken from the mid of each sequence S_i; thus in a region where the error rate is presumably very low. This third pattern acts as safety net: it will not find weak overlaps, but it will recover potential overlaps that have slipped through the patterns from the ends of the sequences because of a too high error rate. Figure 15 gives an overview on the resulting mode of operation for DNASAND.

4.2.4 The ZEBRA algorithm

Grillo et al. (1996) presented an algorithm to find and remove redundancies in genomic databases. This algorithm is based on an 'approximate string matching' procedure, which is able to determine the overall degree of similarity between each pair of sequences contained in a nucleotide sequence database. The basic idea of their computational model is the generation of a set of highly descriptive indices for each sequence position by hashing each position in a nucleotide string and using wildcards.

Although the original method is too slow to be directly applicable to the problem of finding similarities in shotgun data, a way has been found for this thesis to extend and combine it with algorithms coming from the field of theoretical signal analysis. The ZEBRA[4] algorithm that was developed is based on the insight that within normal shotgun data, the number of reads not matching with each other increases with the square of the number of reads present, while the number of reads matching increases only linearly. It is therefore important not to recognise matching reads quickly, but to skip non-matching reads as fast as possible.

The strategy used is not a divide and conquer but rather a transcribe, divide, reorganise, concentrate and conquer strategy. As bonus, ZEBRA calculates the offset of the overlap on the fly.

[4]ZEBRA is not an acronym, but the algorithm was named because it produces 'bands' in memory which resemble the patterns of the african Zebra

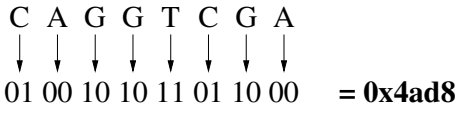

Figure 16: Transforming a nucleotide 8-tuple (octet) into a 16 bit hash.

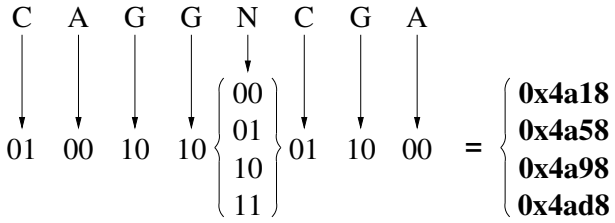

Figure 17: Transforming a nucleotide octet containing one undefined base into four 16 bit hashes.

Implementation of a ZEBRA sequence filter

As shown in figure 16, the alphabet of a nucleotide sequence can be translated into hash values of any desired length by segmenting it into tuples. The cardinality of the nucleotide alphabet is 4 (2 bit). Assume that one can code an A to 00, C to 01, G to 10 and T to 11. This enables to hash 8 nucleotides (nucleotide 8-tuple: an octet) into a 16 bit integer value (the reason for using 16 bits – apart from the fact that this is standard word size for computer systems – will be explained later on).

Doing this for every position of a nucleotide sequence of size n will result in $n-7$ hashes, but – similarly to the strategy used in the DNA-SAND algorithm – an N in the nucleotide sequence is treated as correctly recognised but otherwise unspecified base. Therefore, not one hash is computed for a nucleotide octet containing one N, but four (see figure 17): by sequentially substituting the 'N' for one of the bases A, C, G and T. Although more than one N within an octet could be computed using the same logic, this would increase dramatically the number[5] of hashes to be computed. Fortunately, more than one N within a short number of nucleotides can be seen as hint that the base caller had severe problems to find the right bases because of bad signal quality. As a consequence, nucleotide octets containing more than one N are not computed.

Figure 18 demonstrates how hashes are stored in a table (the combined hash-position table) together with the index position at which they occur. Once the sequence has been transformed into the table, it is sorted using the hashes as

[5] which is 4^k for k Ns.

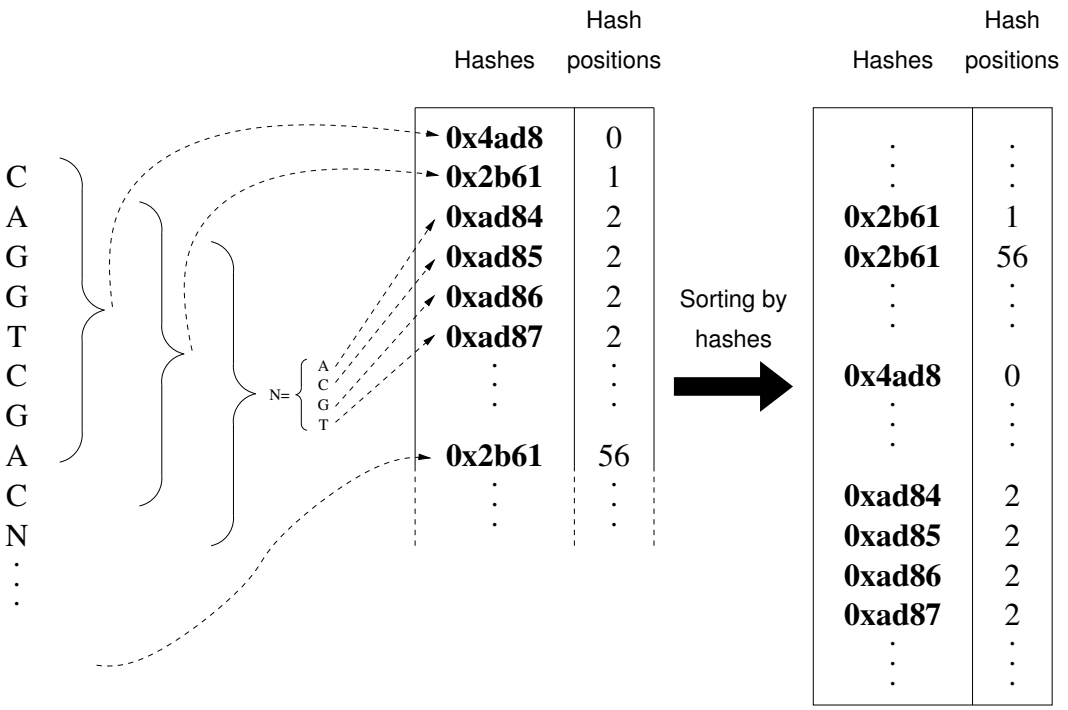

Figure 18: Computation of the combined hash-position table by successively computing nucleotide octet hashes for each position of a sequences and storing the hash values together with the position at which they occurred in a table. Then sort the table using the hashes as key.

key.

Figure 19 shows how the resulting sorted table is split up into two subtables: a hash index table (which will subsequently be called imprint) and a hash position table. The imprint contains exactly $k = 2^n$ elements with n being the number of bits in a hash value. In the case of the ZEBRA algorithm, $n = 16$ so this results to $k = 65536$ elements per table. Each element in the imprint is either a NULL pointer – if the corresponding hash has not been computed – or a pointer to the first hash position element in the hash position table. The hash position table is composed by the hash positions sorted in ascending order with a special element (-1) as delimiter to mark the end of the corresponding hash.

Comparing two sequences is now reduced to the task of comparing two imprints element by element and logging the distance at which equal hash positions appear in two sequences. A NULL pointer in one of any two imprint elements shows that the corresponding nucleotide octet does not appear in the

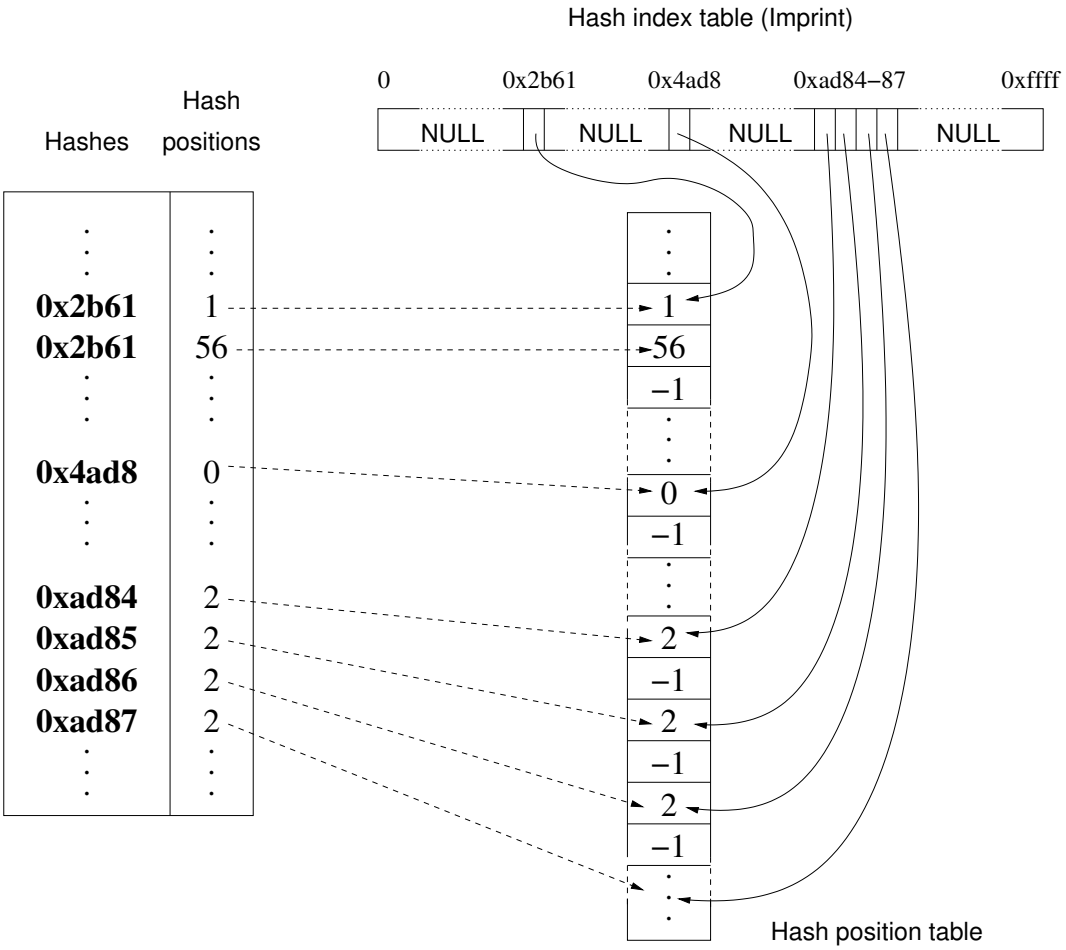

Figure 19: Splitting a combined hash-position table into a hash index table (an imprint) and a hash position table.

sequence belonging to the imprint, whereas a valid non-NULL pointer into a hash position table within both elements of an imprint shows that at least the octet belonging to this hash appears in both nucleotide sequences.

As the length of todays shotgun sequences varies between 300 and 1500 bases, imprints of size $k = 2^{16}$ will have only between $\frac{1}{64}$ and $\frac{1}{40}$ of their elements filled with non-NULL pointers. As result, sequences being completely dissimilar will generate only a few hits when comparing their imprints. Assuming a random distribution of hashes, the number of equal hashes in two

imprints is expected to be

$$\frac{(numhash_1) * (numhash_2)}{k} \qquad \text{with } k = 65536 \text{ in this case}$$

For example, one can expect between 1.7 (300 bases) and 34.3 (1500 bases) occurrences of the same octet in non-related sequences.[6]

In case a hash is present in both imprints, the relative distance between the two octets can be calculated by stepping through the corresponding positions in the hash position table and subtracting the position of the octet in the first sequence from the position of the same octet in the second sequence. This must be done for all appearances of this octet in both sequences. The relative distance at which two equal octets occur in a sequence is logged in a histogram (see figure 20).

Therefore, the algorithm does not explicitly track occurrences of longer subsequences, but looks at the similarity as whole between two sequences. This keeps data structures and code loops simple enough to let compilers optimise at their best.

Note that the problem of comparing two sequences has been reduced to a loop operation that fetches successive data from memory and compares it. Any of todays processors uses speculative data prefetch (sometimes called data burst), i.e, if data is fetched from address n in memory, the processor will speculatively prefetch a certain number of data beginning at $n + 1$ in its 'spare time' as many algorithms work on consecutive data in memory and the probability of needing this data within a short period of time is fairly high. The ZEBRA algorithm is designed in exactly this way to support that feature and this effectively leads to a speed increase of the algorithm: in dissimilar sequences, the algorithm spends most of the time by comparing the two imprints and only exceptionally looks up data in the hash position table, computes octet distances and stores them in the distance histogram.

The histogram itself shows the result of the comparison of two sequences in a very straightforward way. The three distinct types of histogram are:

1. Similar octets occurring in both sequences but at different relative distances from each other. The occurrences are scattered across the histogram, no distinct peak can be detected. Thus, the sequences do not resemble each other and are not possible overlapping candidates.

[6] see also the paper from Pearson (1998) for a review on empirical statistical estimates for sequence similarity searches

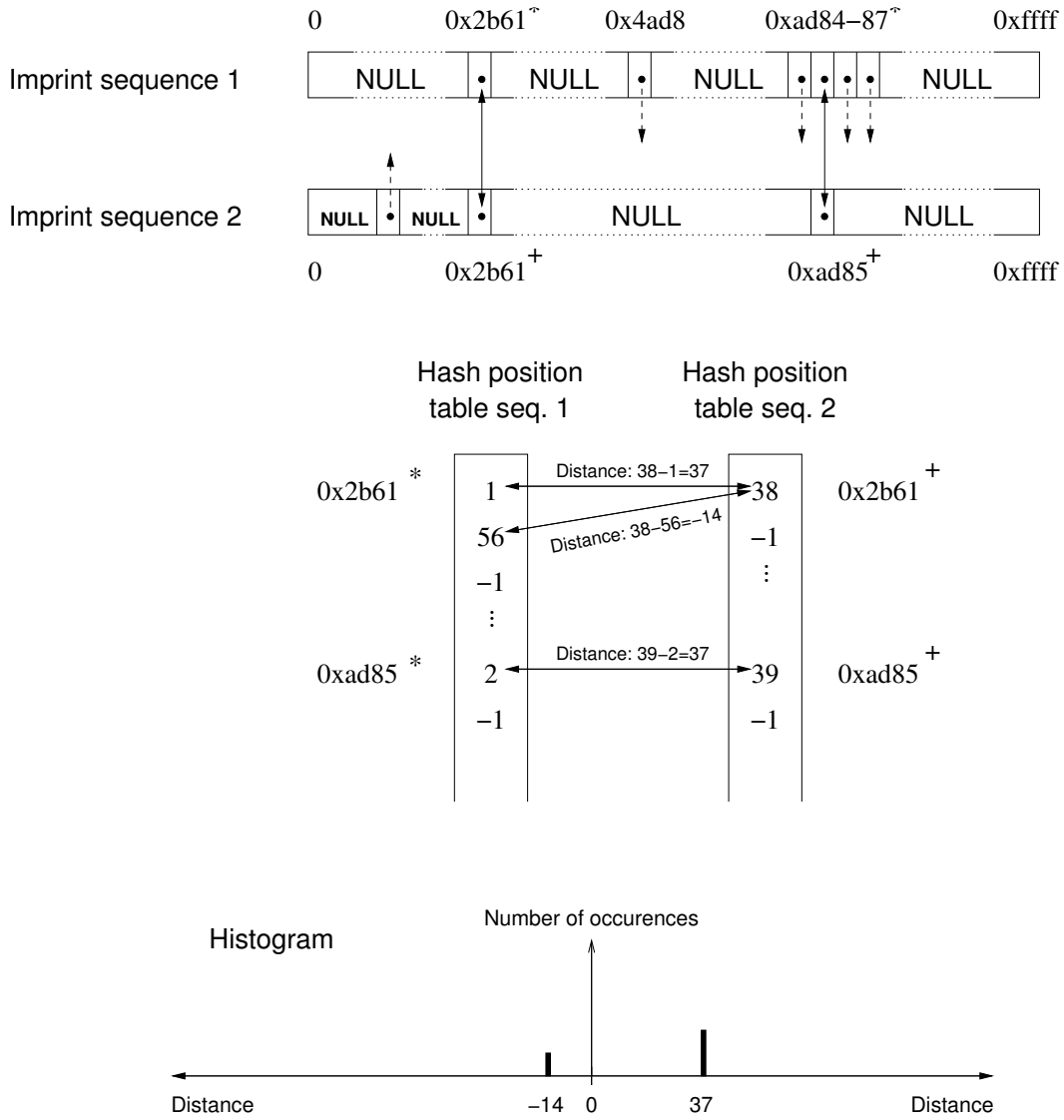

Figure 20: Computational scheme for the octet relative distance histogram of two sequences. Dissimilar sequences with dissimilar octets will show single occurrences scattered across the histogram. Similarities between two sequences will show up as peaks of variable height and width. The higher the peak, the more similar octets have the same relative distance. The wider the peak, the more insertion and deletion errors are in one or both of the two sequences.

2. A single peak. The higher the peak, the more nucleotide octets pairs have the same relative distance and the wider the peak the more insertion and deletion errors occur in one or both sequences.

3. A certain number of peaks, sometimes not clearly separated. This is typical for identical short term repeats between 1 and a few dozen nucleotides occurring in both sequences. The repeated occurrence of equal octets at different places in the sequences leads to a smearing effect.

Once a peak in the histogram exceeds a configurable height, the two sequences can be seen as potentially overlapping. The higher the peak, the more octets in both sequences have the same relative distance from each other. The wider the peak, the more insertion and deletion errors are in one or both of the two sequences. The offset of the peak and its height give – together with the length of both sequences – a valuable first guess of the overlap strength.

Improving the ZEBRA sequence filter: speeding up comparison

An important insight is the fact that comparing two imprints is implemented by iteratively making boolean decisions regarding the state of two array elements. In fact, an imprint element containing a NULL pointer can be expressed as 0 (or false), while an imprint element containing a valid pointer into the hash position table can be expressed as 1 (or true).

$$\text{compressed\_imprint}[k] = \begin{cases} \text{true if imprint}[k] \text{ != NULL} \\ \text{false else} \end{cases}$$

This leads immediately to the approach of compressing the imprints into bit vectors as performing AND operations on multiple bits (8, 16, 32 or even 64 bits). Logical operations like AND are one of the most common and thus fastest operation realised within hardware. Figure 21 shows the compressing process exemplarily.[7]

Instead of comparing 2 times 128kB, the same operation can be done by performing an AND operation on 2 times 8kB. This reduces greatly the number of loop iterations and the amount of data transported on the system bus between RAM and processor.

[7] The graphical display of the resulting bit vectors reminds vaguely to the african zebra, hence the algorithm's name.

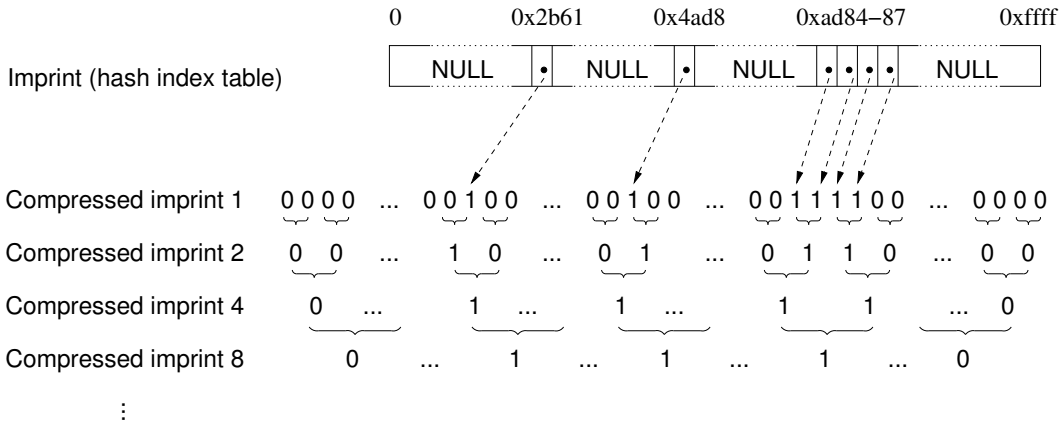

Figure 21: Compressing the imprint by transforming pointers into boolean arrays.

The number of potential overlaps grows linearly to the number n of sequences an assembly project contains, while the number of non-overlapping reads approximates to n^2. It is therefore preferable to speed up the comparison of two dissimilar sequences than the comparison of two similar sequences.

In fact, ZEBRA implicitly allows to take advantage of the data structures created to support this goal. As already proven, major parts of any sequence imprint are filled with NULL pointers (respectively 'false' in a compressed imprint) so that comparing dissimilar sequences will generate only sporadically false positive octet hits. The idea is now to compress the compressed imprint further to reduce the number of loop iterations and data transports between RAM and processor even more.

Consecutive bits in a compressed imprint can be bundled by the OR operation: bundling n consecutive bits leads to a compressed_imprint$_n$. Comparing two sequences is then done by comparing two compressed_imprints$_n$. If both have a bit set at the same position, then the corresponding imprint elements have to be examined more closely. This is – of course – affected with an overhead: data speculatively prefetched by the processor must be discarded and data stored in other tables must be fetched. If n is chosen too low, the full potential of compression and loop iteration decrease is not used. On the other hand, if n is chosen too high, the imprints will be compressed too much: this leads to an exaggerated number of false positive hits in the comparison of two compressed_imprint$_n$.

During experiments with projects between 500 and 5,000 reads, having an

```
A C A G G T C G A
↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓
   01 00 10 10 11 01 10 00    = 0x4ad8
00    00 10 10 11 01 10 00    = 0x0ad8
00 01    10 10 11 01 10 00    = 0x1ad8
00 01 00    10 11 01 10 00    = 0x12d8
00 01 00 10    11 01 10 00    = 0x12d8
00 01 00 10 10    01 10 00    = 0x1298
00 01 00 10 10 11    10 00    = 0x12d8
00 01 00 10 10 11 01    00    = 0x12d4
00 01 00 10 10 11 01 10       = 0x12d6
```

Figure 22: Transforming a nucleotide 9-tuple into 9 hashes describing the tuple allowing for 1 error.

average usable sequence length between 300 and 1,000 bases, nesting a compressed_imprint$_1$ loop within a compressed_imprint$_8$ loop gave the best overall time performance.

Improving the ZEBRA sequence filter: dealing with errors

Up to this point, the ZEBRA algorithm still cannot handle insertion or deletion errors very well. Still, it is known that the sequences that are to be compared are potentially error prone, especially toward the ends. This may include insertion and deletion errors.

This can be addressed quite easily by using a technique made popular by Grillo et al. (1996). The technique presented in that paper consists of computing not one hash per sequence position, but expecting the nucleotide tuple to contain errors and taking this into account by computing multiple hashes, each hash representing the possibility of a certain number of errors being contained in the tuple.

As already seen in section 2.2, sequences gained in laboratory tend to have less than 1% errors within the good middle part, but up to 15% toward the ends before the reads become completely unusable.

Following the conclusions of Grillo et al. (1996), computing n_1-tuples – this means, tuples containing n nucleotides but allowing 1 error – is a good compromise between sensitivity, specificity, computational overhead and execution speed. Porting this to the ZEBRA algorithm, this results into using 9-tuples allowing 1 error (see figure 22).

Doing this for any position in each sequence allows the ZEBRA algorithm

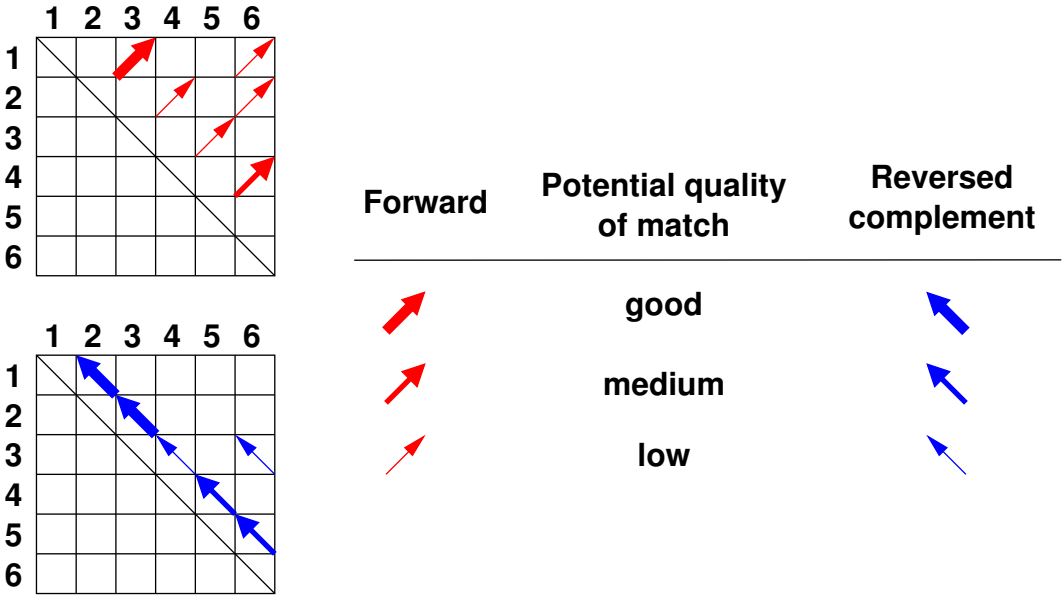

Figure 23: The "matrices" generated after the first fast scan of every read against every other in search for potential overlaps. Unlike this small example with 6 reads might suggest, the matrices in real world projects – with a large number of reads – are normally sparsely occupied. Therefore, using hash storing techniques is more appropriate as otherwise memory consumption would explode for larger number of reads.

to deal with insertion/deletion errors, but also increases the number of hashes present in an imprint by approximately a factor of 8 to 9.

4.3 Systematic match inspection

Although the DNASAND and the ZEBRA algorithm do not specify the overall type of global relationship of two sequences (total correspondence, containment or overlapping, see Huang (1994)), any type of relationship is recognised. As result of this first scan, a hash storing technique is used to represent sparse matrices containing information on potential overlaps of all the fragments and their orientation (forward–forward or forward–complement) is generated. Figure 23 shows a schematic drawing on how these data structures are interpreted internally.

In the next fundamental step, these potential overlaps found during the scanning phase must be examined more thoroughly. Several algorithms have bee

devised for alignments of multiple sequences: Allison (1993) devised a divide-and-conquer technique for aligning three strings, Stoye (1998) used a similar technique for 6 to 12 strings. But these multiple alignment algorithms are still too slow (see Gotoh (1993) for a good explanation) to be used in an assembly match inspection phase, even when parallelised multiple alignment algorithms run on multiple processors like Kleinjung et al. (2002) presented.

In the end, the strategy devised for the `mira` assembler works by examining potential matches two at a time with a modified Smith-Waterman algorithm for local alignment of overlaps. Similar strategies were extensively studied by Barton (1993) and have found their way into other current assemblers like, e.g., PGA (Paracel (2002b)).

4.3.1 Improving Smith-Waterman alignment by banding

Observe that two sequences that align well will generate a path through a Smith-Waterman alignment matrix that will almost run diagonally from the entry point to the exit point. Only indels will cause a horizontal or vertical shift of the alignment graph. Assuming that one knew the approximate whereabouts of either the entry or the exit point and that the alignment has 'acceptable' quality, i.e. not too many indels that are spread unilaterally in one of the sequences. It is then a valid assumption that it should be feasible to reconstruct the alignment by calculating only the narrow corridor (a band) through the matrix where the alignment graph is expected to run through, as is shown in figure 24.

A bonus information not to be underestimated is the fact that both DNASAND and ZEBRA not only provide the information whether two sequences are similar enough to try a sequence alignment, but they also provide the approximate offset for the sequence alignment. This information is valuable because it locates the approximate entry points of such an alignment in the alignment matrix and therefore allows to reduce the computational complexity – and with it the time needed – for a SW alignment with two sequences drastically.

Let n_1 and n_2 be the length of the sequences, m the approximate length of the overlap and k a safety margin of indels that might arise in the alignment of the overlap. It is clear that

$$m \leq min(n_1, n_2)$$

is always true for every possible configuration of m, n_1 and n_2. Furthermore

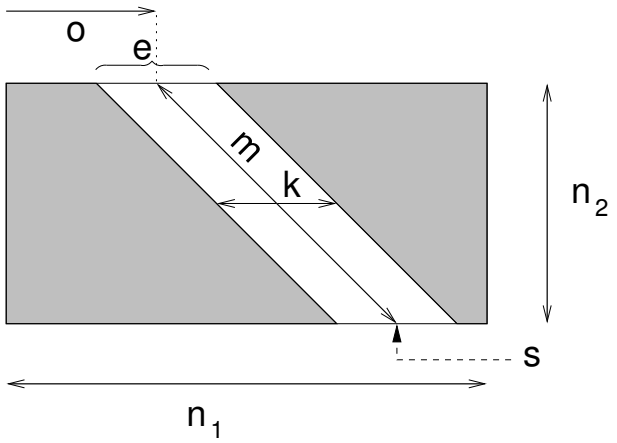

Figure 24: Smith-Waterman banding. n_1 and n_2 are the length of the sequences. Instead of computing the whole matrix ($n_1 * n_2$ cells), only a small band within the approximated area of the overlap must be computed: o is the approximated offset from sequence 1 to sequence 2 in the alignment, m the approximated length of the overlap and k the band width to be calculated. s is the predicted starting point for the recursive path drawing algorithm and e the predicted exit area.

one can assign k to be

$$k = \frac{1}{10} * m \qquad \text{where typically} \qquad 30 \leq k \leq 70$$

The higher k, the more tolerant the algorithm will be regarding slightly false approximations of alignment path entry-points or the more non-compensated indels can occur in one of the sequences. Using the values mentioned above ensures a sufficiently wide and adaptable security margin for the length of sequences that have to be aligned in a shotgun assembly while keeping k linear within bounds.

While the normal Smith-Waterman alignment will need $O(n_1 * n_2)$ time to compute the alignment matrix, the banded version will need only

$$O(k * m) \lessapprox O(\frac{1}{10}m^2)$$

to compute the band through the matrix where the alignment is supposed to be. For n_1, n_2 being typically between 300 and 1200 and normally

$$m \ll n_1 * n_2$$

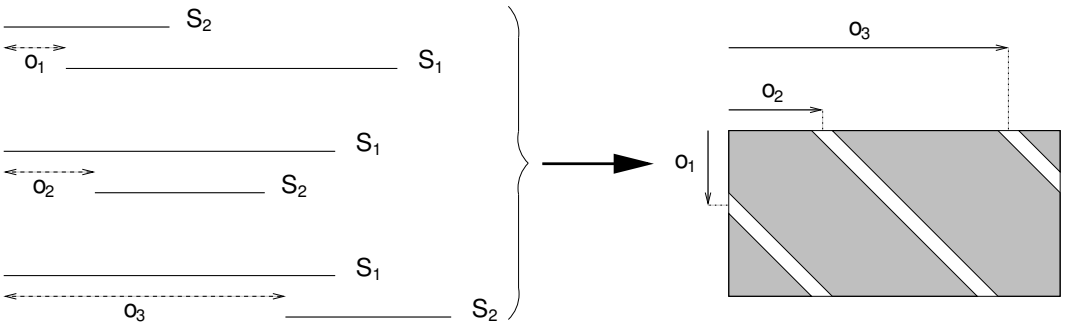

Figure 25: Smith-Waterman band prediction. According to the offset prediction from DNASAND and ZEBRA-BLOCKING (examples o_1, o_2 and o_3 in the figure), different bands can be computed, all differing in length. These examples show particularly well how the $O(n^2)$ Smith-Waterman calculation of a matrix can be effectively scaled down to less than $O(n)$.

so $O(k * m)$ can be seen to have mostly a less than linear complexity compared to the square complexity of $O(n_1 * n_2)$.

Note that although methods for linear-space alignment methods have been devised[8], these have not been implemented for the time being as typical shotgun fragment lengths are small enough to allow quadratic space algorithms.

The changes needed to adapt the SW algorithm are minimal. A small logical overhead is needed to segment the band and the matrix to facilitate computation. Additionally – and assuming the alignment matrix is calculated row by row – the left and right edge of each row must calculated with a slightly adapted algorithm to take care that the recursive alignment algorithm afterwards will not run out of the band.

Cells outside the calculated band are assigned to have a very high value, e.g. the `INTMAX` value, which is defined to be the highest integer value that can be represented within a given integer datatype. In practice, to avoid the $O(n^2)$ complexity of filing the whole matrix, only the matrix cells adjacent to the calculated band are initialised. This constitutes a 'fence' that the recursive path alignment algorithm will not be able to transgress later on and make sure that all alignments found later will stay within the calculated band.

The calculation of cells that are not on the edges (the 'a' cell in figure 26) is

[8]for example see Chao et al. (1994); Grice et al. (1997) for an overview and Chao et al. (1995) for an application

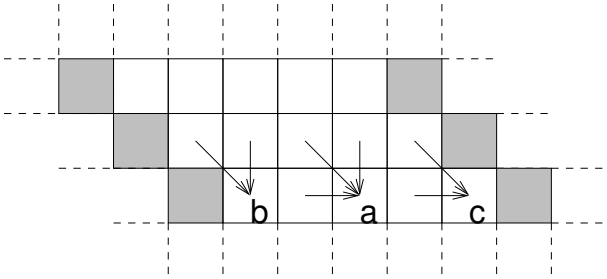

Figure 26: Smith-Waterman (SW) band calculation predecessor rules. The figure shows a section of a larger SW matrix. The grey cells represent the border of the band within the matrix, arrows show the predecessor cells needed to compute a cell. Most cells in a band will be of type (a), i.e., cells with normal predecessor rules like in a normal SW matrix. Cells on the border of the band (b) and (c) need special handling as they only have two (different) predecessor cells instead of three.

performed like in a normal SW algorithm:

$$H_{i_1,i_2} = max \left\{ \begin{array}{c} H_{i_1-1,i_2-1} + score(\mathcal{S}^1_{i_1}, \mathcal{S}^2_{i_2}) \\ H_{i_1,i_2-1} + score(\mathcal{S}^1_{i_1}, *) \\ H_{i_1-1,i_2} + score(*, \mathcal{S}^2_{i_2}) \\ 0 \end{array} \right\}$$

while each cell on the left edge in a row (the 'b' cell in figure 26) must not consider the fence value to its left and consequently is calculated using

$$H_{i_1,i_2} = max \left\{ \begin{array}{c} H_{i_1-1,i_2-1} + score(\mathcal{S}^1_{i_1}, \mathcal{S}^2_{i_2}) \\ H_{i_1-1,i_2} + score(*, \mathcal{S}^2_{i_2}) \\ 0 \end{array} \right\}$$

Similarly, each cell on the right edge in a row (the 'c' cell in figure 26) must not consider the fence value to its top, so that the calculation is done using

$$H_{i_1,i_2} = max \left\{ \begin{array}{c} H_{i_1-1,i_2-1} + score(\mathcal{S}^1_{i_1}, \mathcal{S}^2_{i_2}) \\ H_{i_1,i_2-1} + score(\mathcal{S}^1_{i_1}, *) \\ 0 \end{array} \right\}$$

Table 2: Extend version of the matrix shown in table 1. Here, comparisons of $\mathcal{A}^b$ with endgaps (∇) or with "N" are score neutral.

	A	C	G	T	N	$*$	∇
A	1	-1	-1	-1	0	-2	0
C	-1	1	-1	-1	0	-2	0
G	-1	-1	1	-1	0	-2	0
T	-1	-1	-1	1	0	-2	0
N	0	0	0	0	0	-2	0
$*$	-2	-2	-2	-2	-2	0	0
∇	0	0	0	0	0	0	0

4.3.2 Parametrising Smith-Waterman alignment

Several scoring schemes for SW alignments have been devised within the last two decades, ranging from the simple original one (Smith et al. (1981)) to methods accounting for gap lengths using affine gap calculation and even alignments of sequences against trace signals. Most publications that appeared analyse and discuss the complexity of dynamic programming and scoring functions. Even very formal but systematic and generally applicable methods to build early prototypes for these algorithms were presented (see also Giegerich (2000)). Althaus et al. (2002) proposed a multiple sequence alignment (MSA) with arbitrary gap costs which computes an optimal solution using polyhedral combinatorics, but having only two sequences at a time to score.

Arslan et al. (2001) presented a method that used iterated Smith-Waterman computations with fractional programming with a run time of $O(n^2 log(n))$ – which is already higher than the $O(n^2)$ of standard Smith-Waterman – to normalise the score of sequence alignments. Their experimental results suggested that the number of required iterations were small, but they could not establish a satisfactory theoretical lower- / average- / upper-bound for the growth in the number of iterations needed.

Good alignments base on an appropriately chosen scoring scheme given to an optimisation model. A slightly improved original Smith-Waterman scoring scheme has been opted for simplicity, speed and sufficient sensitivity and specificity of a first alignment approximation. The Smith-Waterman alignment algorithm uses the weight matrix W given in table 2 to calculate an alignment score.

As can be seen by the values of W, this scoring scheme reflects ideally the requirements that alignments of shotgun sequences with some degree of errors

Table 3: Default penalty scores inflicted to the score calculated with a weight matrix. Each gap of a given length reduces the original score by a specified penalty.

gap length in bases	penalty in %
1	0
2	5
3	10
4	20
5	40
6	80
7+	100

contained: one can assume a 'mismatch' of a base against a 'N' (symbol for aNy base) to have no penalty as in many case the base caller rightfully set 'N' for a real, existing base (and not an erroneous extra one) that could not be resolved further. But to show that this still constitutes a minor reason for being careful, a 'N' gets a neutral score instead of a match.

The scoring algorithm can be improved further by taking into account that – regarding the fact that todays base callers have a low error rate – the block-indel model (see Giegerich and Wheeler (1996)) does not apply to the alignment of shotgun sequencing data: long stretches of mismatches or gaps in the alignment are less probable than small, punctual errors. In fact, more than two gap symbols following each other in an alignment of shotgun sequences is a distinct signal that either the sequences should probably not be assembled together or that something went wrong in the laboratory or during signal processing (base calling). For this reason, a configurable penalty function has been added that scales down the score calculated through the scoring matrix W in relationship to the number and length of gaps within an alignment. The penalties used are shown in table 3. This is a post-processing normalisation algorithm which is not dissimilar to the methods proposed by Pearson (1995) and Shpaer et al. (1996).

In addition, a second score – the expected score – is calculated using the score matrix represented in table 4. It can be deduced by the matrix values that this second score represents the score expected if the alignment was perfect, i.e., without errors. Consequently

$$score \leq score_{expected}$$

Table 4: Scoring weight matrix for the expected score

	A	C	G	T	N	$*$	∇
A	1	1	1	1	0	1	0
C	1	1	1	1	0	1	0
G	1	1	1	1	0	1	0
T	1	1	1	1	0	1	0
N	0	0	0	0	0	1	0
$*$	1	1	1	1	1	0	0
∇	0	0	0	0	0	0	0

is always fulfilled. Note that it is not possible to take the length of the overlap as expected score, because 'N' bases are treated as neutral in score calculation. They therefore cannot get a better score in the calculation of the expected score as this would mean there is an error although it is still assumed that an 'N' is a correctly found base.

4.3.3 Attributing an alignment overlap

It is now trivial to make a "rough guess" of the alignment quality of the overlap by calculating the score ratio.

$$R_s = \frac{1}{score_{expected}} * score \qquad \text{with } R_s = 0 \text{ for } \begin{cases} score < 0 \\ score_{expected} = 0 \end{cases}$$

and therefore

$$0 \leq R_s \leq 1$$

A score ratio of 0 shows that the two sequences do not form a valid alignment while a ratio of 1 means a perfect alignment without gaps or base mismatches.[9]

A problem left open up to now is the question on how to subsume the different quality criteria i) length of the overlap and ii) score ratio into one pregnant figure (the overlap weight) so that overlaps can be ranked easily and comprehensibly.

The simplest method would be to multiply both values to get a weighted length of the overlap. Let len_o be the length of the overlap, so the desired weight

[9] but perhaps one or several aligns of a base against an 'N'

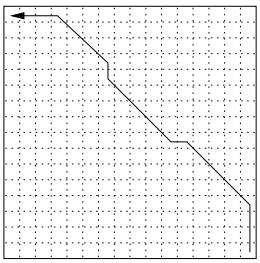

Figure 27: A modified Smith-Waterman algorithm for local alignment is used to confirm or reject potential overlaps found in the fast scanning phase. Accepted overlaps get a weight assigned depending on the length of the overlap and alignment quality.

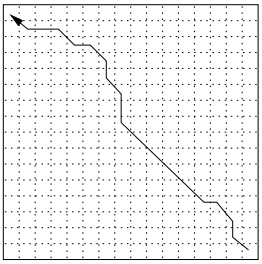

Figure 28: Although having a good partial score, the overall score ratio of this alignment is too low to be accepted. This read pair is eliminated from the list of possible overlaps.

w_o of this overlap could be computed by

$$w_o = len_o * R_s$$

But this approach attributes far too much weight to the length than it does to the score ratio, which is – after all – the predominant measure for the quality. For example, an overlap of length 650 bases with a score ration of only 0.65 (65% similarity) would get a higher weight (422) than an undeniably better overlap of length 400 bases and a score ratio of 0.9 (90% similarity, weight: 360). This problem can be easily circumvented by squaring the score ratio, giving it more importance in the calculation:

$$w_o = len_o * (R_s)^2$$

The first overlap would then get a weight of 274 and the second 324, which is exactly the desired outcome.

Every candidate alignment overlap pair whose score ratio is within a configurable threshold (normally upward of 70% to 80%) and where the length of the overlap is not too small is accepted as 'true' overlap, candidate pairs not matching these criteria – often due to spurious hits in the scanning phase – are identified and rejected from further assembly (see figures 27 and 28).

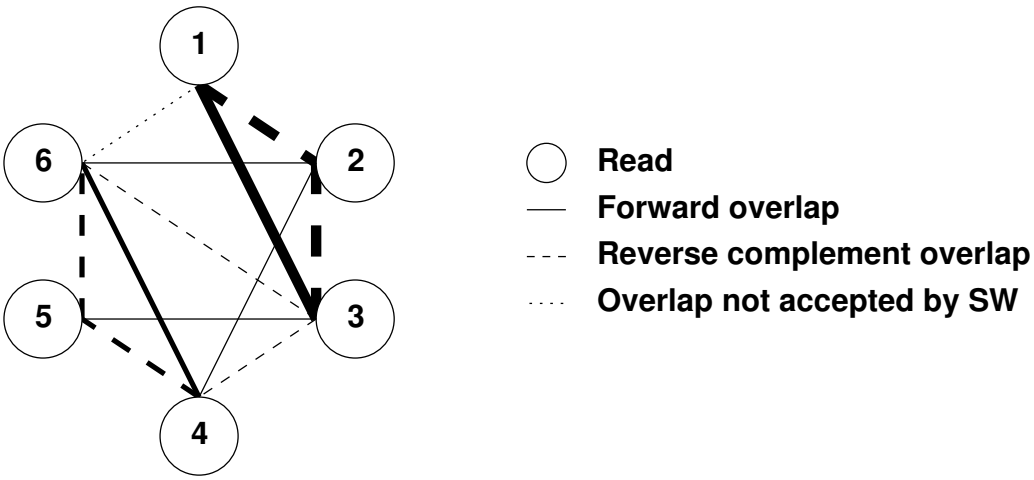

Figure 29: An overlap graph generated from the aligned overlaps that passed the Smith-Waterman test. The thickness of the edges represents the weight of an overlap. Although the (1,6) overlap is marked for demonstration purposes in this figure, rejected overlaps (due to spurious hits in the scanning phase, see figure 23) are not present in the graph.

4.3.4 Building a graph

The overlap alignment – along with complementary data (like orientation of the aligned reads, overlap region, score, score ratio etc.) – is called an aligned dual sequences (ADS). Every ADS that passed the Smith-Waterman test is kept in memory to facilitate and speed up the next phases. Good alternatives are also stored to enable alternative alignments to be found later on in the assembly.

All the ADS form one or several weighted graphs which represent the totality of all the assembly layout possibilities of a given set of shotgun sequencing data (see figure 29). The nodes of the graph are represented by the reads. An edge between two nodes indicates that these two reads are at least partially overlapping and can be aligned. Each graph sketches the alignment possibilities of reads for at least one contig. Hence, the number of non-connected graphs is equivalent to the minimum number of contigs the assembler will be able to build.

The edges themselves are attributed with the score weights computed for the overlaps and stored in the ADS.

4.4 Building contigs

The overlaps found and verified in the previous phases must then be assembled into contigs. This is the most fundamental and intricate part of the process, especially in projects containing many repetitive elements. Several basic approaches to the multiple alignment problem have been devised to tackle this problem. Although algorithms for aligning multiple sequences at once have been used with increasing success lately for up to 10 to 15 sequences (Stoye (1998)), the amount of time needed to perform this alignment is still too unpredictable – ranging from a few seconds to several hours – to be used in sequence assembly.

Reinert et al. (2000) conducted experiments on iterative methods for faster sum-of-pairs with a divide-and-conquer alignment approach (using a $\mathcal{A}^*$ algorithm) and quasi-natural gap costs for multiple sequence alignments. Schlosshauer and Ohlsson (2002) devised an algorithm based on fuzzy recast of dynamic programming algorithms in terms of field annealing to score the reliability of sequence alignments. These methods also show varying runtimes ranging from a few seconds to hours for a relatively low number of sequences, far lower that the realistic number of sequences encountered in real world assembly projects.

Due to these problems, it was decided to use iterative pairwise sequence alignment and devise new methods for searching overlap candidates. A key concept is empowering contigs to accept or reject reads presented to them during the contig building. The algorithm consists mainly of two objects which interact with each other: a pathfinder module and a contig building module.

4.4.1 Pathfinder and contig interaction

Because an iterative approach is used to the multiple alignment problem – this means one always successively aligns an existing consensus against the next read – the results of the alignment sensitively depends on the order of pairwise alignments (Morgenstern et al. (1996)). As a direct consequence it is therefore preferable to build contigs using overlaps with reliable and high quality first and then continue with lower quality.

Hence, one has to make sure to start at the position in the contig where there are as many reads as possible with almost no errors. The pathfinder will thus – in the beginning – search for a node in the weighted graph having a maximum number of highly weighted edges to neighbours. The idea behind this behaviour

is to take the read with the longest and qualitatively best overlaps with as many other reads as possible and use this one to ensure a good starting point: the 'anchor' for this contig.

A necessary constraint proved to be that the weight of the edges selected must not fall below a certain threshold as it is not infrequent to have a considerable number of relatively short reads with repetitive element contained in many projects. This introduces a sort of 'gravity well' in the assembly graph and misleads the algorithm to take a read with dangerous repetitive characteristics as anchor, which is generally not the intended behaviour.

The next step consists of determining the next read to add to the now existing contig by analysing nodes (reads) from the contig in the graph and determine which edge (overlap) with a node – not belonging to the contig – provides the most profitable overlap to augment the information contained within the contig.

There are theoretically two distinct approaches to perform this operation:

1. Backbone strategy: extend the existing contig in length with new, formerly not present data at the extremities of the contig

2. In-depth coverage strategy: improve the quality of the existing consensus by adding reads that increase coverage and thus increase base probabilities in the consensus

Both approaches have their specific advantages and disadvantages which will be discussed briefly.

Extending the contig length first implies the quality of the reads to be reliably high anytime and anywhere. This cannot be guaranteed for sequences extracted by electrophoresis and subsequent base calling as base probability values do not exist for every base calling method. The backbone strategy can also be easily affected by repetitive short and long subsequences in the target DNA if additional assembly support information is not available. On the other hand, once the backbone of the assembly has been constructed, adding more reads to reduce base calling errors step by step is trivial.

Increasing the coverage of a contig first – by adding reads containing mostly sequences already known – ensures that the base qualities of the consensus raise rapidly: more reads that cover a consensus position make consensus errors introduced by bad signal quality and subsequent base calling errors increasingly improbable. Reads added afterwards have thus an increasing probability to align against a right consensus sequence. The disadvantage of this method appears when parts of a contig are only covered by low to medium qual-

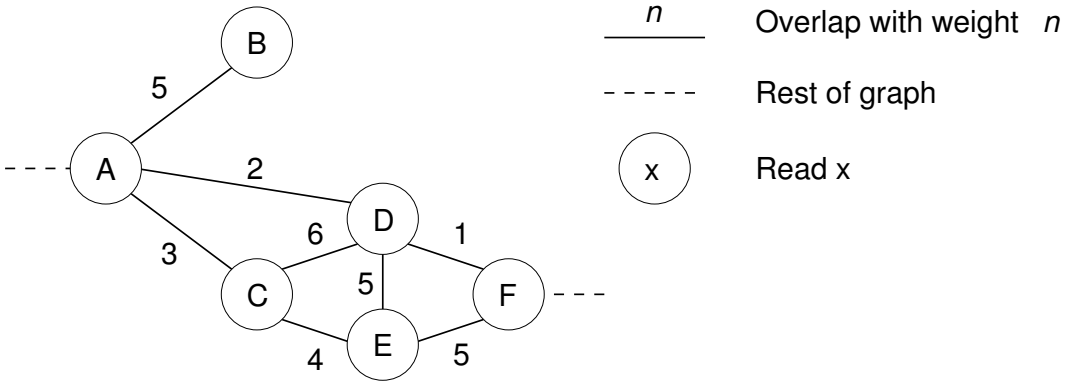

Figure 30: Example for an assembly graph (to be studied together with figure 31 on page 67 that causes problems when using a simple greedy algorithms, assuming that read B is a chimeric sequence.

ity reads. The increased coverage results in a growing number of discrepancies between read bases, leading to incorrect an consensus and sometimes stopping the assembler from adding reads to a contig where there is theoretically a possibility to continue.

4.4.2 Path traversal strategies

The assembly graphs build in the previous step have the unpleasant characteristic to be quite large even for small assemblies. In case of multiple – almost identical – repeats, the number of edges present in the graph explodes. Figure 30 shows a part of a much bigger graph used as example in this section.

Simple greedy algorithm

A simple greedy algorithm was first tried to find overlap candidates in the assembly graph, but on occasions – especially in highly repetitive parts of a genome or with chimeric reads[10] in low coverage sequencing – this algorithm fails. Figures 30 and 31 demonstrates how the simple greedy algorithm in conjunction with a chimeric read can provoke the termination of contig building while there exist possibilities to extend it further. This also corresponds to the results that have been reported previously by Myers (1994).

[10]Chimeric reads, as described in section 2.2.2, must be considered as garbage.

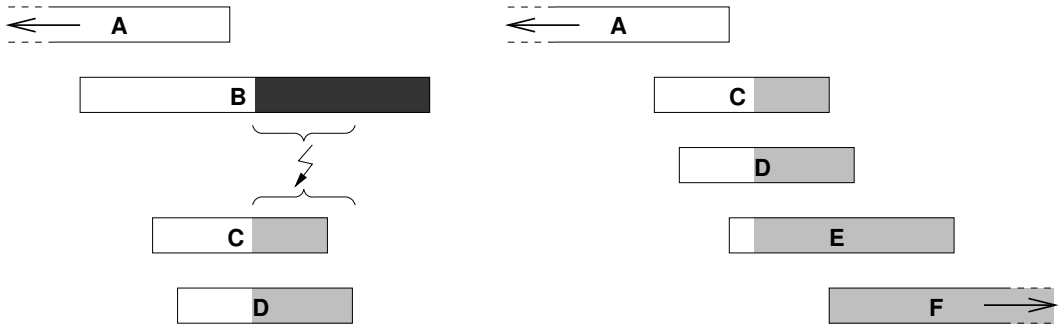

Figure 31: Example continued from figure 30 for a failure of the simple greedy algorithm (left side): read B is a chimeric read that has a strong overlap with read A, but would lead into a dead end of the assembly. A recursive look ahead strategy discovers contradictions between reads before they are assembled (right side): read B is automatically not used in the assembly.

n, m-step recursive look-ahead strategy

The current algorithm is an extension of the greedy algorithm. The pathfinder will search for the n best edges leading from nodes that are already inserted in the contig to nodes that have not been used yet. These n new nodes are inserted in n partial look-ahead paths as potential next alignment candidates. Each partial path is then descended recursively for a maximum of m recursions, repeating in each step the selection of the n next best edges. In the end the pathfinder designates the next read to add to an existing contig by taking the edge leading to the first node that is contained in the best partial path found so far, i.e., the partial path with the best summed score of its edges wins. It then presents the read which the edge led to – and its approximate position – to the contig object as potential candidate for inclusion into the existing consensus.

By performing an in-depth analysis – with a cutoff value of normally 4 or 5 recursions – of the weights to neighbouring reads contained in the assembly graph, the n, m recursive look-ahead strategy can substantially reduce the number of misalignments as chimeric reads get only a very small chance of getting aligned. At the same time, reads containing repetitive elements will not get aligned without 'approval' from reads already in the contig and reads to be aligned afterwards.

4.4.3 Contig

A contig is represented by a collection of reads that have been arranged in a certain order with given offsets to form an alignment that is optimal. The problem lies in the definition 'optimal' as different criteria like length, coverage, number of inconsistencies and others can be applied. The usual definition of an SCS (shortest common superstring) alignment given in literature leads in many cases to over-compressed alignments, especially when repeats are present. On the other hand, defining an alignment to be optimal when it has the least possible number of inconsistencies leads to alignments that are too long.

To elude these problems known in literature, an alignment has been defined to be optimal when the reads forming it have as few unexplained errors as possible but still form the shortest possible alignment.

For this purpose the contig object has been provided with functions that analyse the impact of every newly added read (at a given position) on the existing consensus. The assumption is now that – as the assembly presumably started with the best overlapping reads available – the bases in the consensus will be right at almost every position. Should the newly added read integrate nicely into the consensus, providing increased coverage for existing consensus bases and perhaps extending it a bit, then the contig object will accept this read as part of the consensus.

In some cases the assembly graph will contain chance overlaps between otherwise unrelated reads. Should such an overlap be good enough to be chosen by the pathfinder algorithm as next alignment candidate, the read will most probably differ in too many places from the actual consensus (thus differing in many aspects from reads that have been introduced to the consensus before). The contig will then reject the read from its consensus and tell the pathfinder object to search for alternative paths through the weighted overlap graph. The pathfinder object will eventually try to add the same read to the same contig but at a different position or – skipping it – try other reads.

Once the pathfinder has no possibilities left to add unused reads to the actual contig, it will again search for a new anchor point and use this as starting point for a new contig. This loop continues until all the reads have been put into contigs or – if some reads could not be assembled anywhere – form single-read contigs.[11]

[11] Of course, a single read itself cannot be called a contig. But putting it into the same data structure (a contig object) like the other, assembled reads is a convenient way to keep unassembled reads in a database.

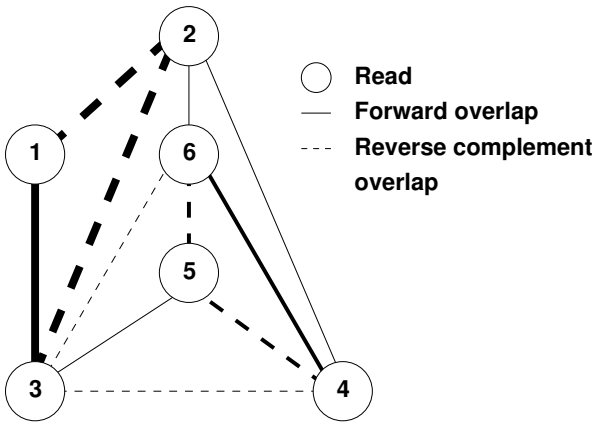

Figure 32: Assembly graph from figure 29 used as example in section 4.4.4.

4.4.4 Contig approval methods

Myers et al. (1996) proposed using constraints in progressive multiple alignment. The idea has been extended toward the object acceptance method described in this section. As mentioned briefly above, each contig object has the possibility to reject a read being introduced into its consensus. This is one of the most crucial points in the development of the assembler: allowing presumably good building blocks, i.e. reads with high quality, to start an assembly is a decisive step in the ongoing assembly process, but the contig must actively decide upon all available data if the reads proposed to it are good enough to be assembled. This allows to use explicit knowledge available in reads that are known to be good and the implicit knowledge present in the assembly.

The simplest behaviour of a contig could be to simply accept every new read that is being presented as part of the consensus without further checks. In this case the assembler would thus rely only on the weighted graph, the method with which the weighted edges were calculated and the algorithm which traverses this graph. Figure 33 shows this exemplarily for the reads from the example set up above. Although there are four columns having discrepancies, the assembly looks quite reasonable.

Acceptance upon actual consensus constraints

Every read added to a contig must match the consensus build by previously inserted reads. As the overall working mechanism of the assembly algorithm will insert reads with good quality and strong overlaps into a contig first, there

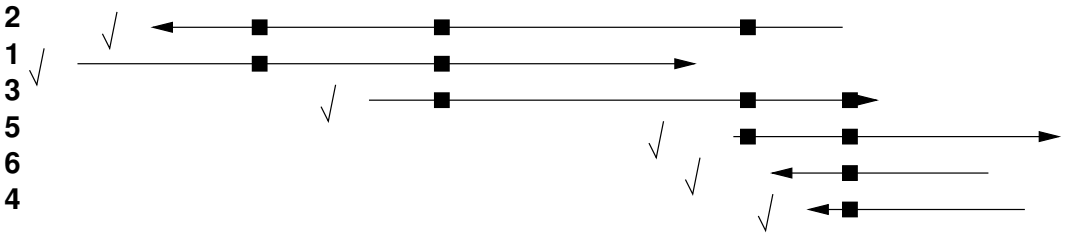

Figure 33: Example of simplest possible assembly which does not take advantage of additional knowledge. Reads are assembled as the algorithm walks through the path, the checkmarks showing the reads have been accepted as matching to the contig. Black squares mark discrepancy columns not having the same bases. Continued in figure 34.

is a reasonable chance to believe that the actual consensus formed by these reads is quite correct up to a certain threshold.

If a newly inserted read shows too many discrepancies to the actual consensus sequence, the contig will invariably reject this read and the edge representing the corresponding overlap will be removed from the assembly graph. The main reasons for overlaps of this type being present in the assembly graph include spurious matches that slipped through Smith-Waterman, repetitive sequence and chimeric reads.

Repeat marker tags

However, using additional information that is available at the time of assembly proves useful. For example, known standard repetitive elements – e.g. the so called ALU repeat type in human genome – can easily be tagged in the data preprocessing step of the assembly. It is therefore possible to apply much stricter control mechanisms in those stretches of a sequence known to be repetitive and therefore dangerous to the assembly process. In many cases, the copies present across different regions of a genome are up to 98% to 99% similar, hence the necessity to be able to decide upon a few bases that might really differ.

In case of repetitive sequences, each discrepancy between a newly inserted read and the actual consensus is treated as single-base error region, i.e. the explanation of the possible base-call error to be analysed has to be searched in the immediate vicinity of the base. The question the simple signal analysis (see the following section) has to answer is if whether or not it is possible – that is, if there is enough evidence in the trace data – to substitute the offending base of the new read against the base calculated in the consensus. If signal

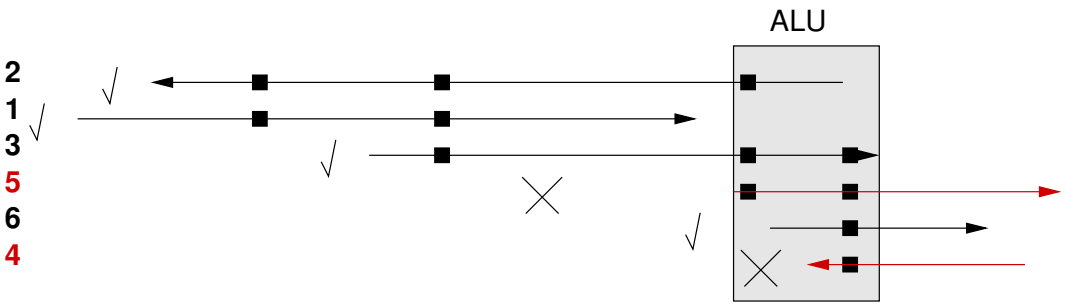

Figure 34: Using additional information on standard repetitive stretches, the contig object checks the repeat regions more thoroughly and rejects reads that have too many errors that cannot be explained by signal analysis. Example continued in figure 35 on page 73.

analysis reveals that it is possible to substitute bases, the error is treated as 'explainable' error and it is treated as 'unexplainable' if not.

If the percentage of 'unexplainable' errors in the repetitive region surpasses a certain threshold (default value is 1%), then the newly inserted read will be rejected from the consensus and its node put back into the assembly graph as shown in figure 34.

Simple signal analysis

The best method to adjudicate on conflicting bases of different reads is to perform signal analysis, going back to the original trace data produced by the sequencing machines. This labour intensive task is normally executed by highly trained personnel although in a significant number of cases it is fairly easy to decide on conflicting readings by analysing the trace data.

Two different computational methods have been established in the past years to adjudicate om conflicting bases in reads. The probably most common method is the usage of quality values first introduced by Staden (1989) and further developed by Bonfield et al. (1995a). The concept was then refined to probability values by Ewing and Green (1998). Base probability values are experimentally gained, position-specific probabilities of error for each base-call in a read as a function of certain parameters computed from trace data. Probability values are of high value for finding bases with weak quality, but these values are mostly only available for the called bases, not for uncalled ones. For an assembler, it is a big drawback not to have the capability to search for alternative base-calls that might have been possible at the same position in the trace.

Another suggestion on incorporating electrophoresis data into the assembly process promoted the idea of capturing intensity and characteristic trace shape information and provide these as additional data to the assembly algorithm (Allex et al. (1996, 1997)). Although methods for storing alternative base calls and other data have been suggested (Walther et al. (2001)), the approach from Ewing and Green (1998) produces results that are simple to handle and apparently 'good enough' so that it has imposed itself as standard over the years.

Complex shape approaches were discarded early in development of the algorithms for the assembler as essentially all the contig – and with it the consensus computing algorithm of a contig – needs to know is if the signal analysis reveals enough evidence for resolving a conflict between reads by changing the bases incriminated. This question is a simple working hypothesis which is a subset of the hypotheses generated by the automatic editor devised by Pfisterer and Wetter (1999).

It is possible to call the automatic editor in analysis mode to have this specific questions analysed. The *conditio sine qua non* for the assembly algorithm is that the signal analysis does not try to find support for a change of the possibly faulty base with "brute force". This would lead into the doctrine to use high quality sequence to adjust presumably low quality sequence, which is a complete contradiction of the working principle of using high quality data to explain errors in low quality sequences and correct them only if the low quality data supports the new theory.

In fact, the analysis is executed with the uttermost cautiousness and the signal analyser will support a base change only if hard evidence for this hypothesis can be found in the trace signals. Otherwise the assembler treats signal analysis as a black box decision formed by an expert system, only called during the assembly algorithm when conflicts arise. But this expert system provides nevertheless additional and more reliable information than the information contained in quality values extracted from the signal of one read only (compare to Durbin and Dear (1998)): there is a reasonable suspicion deduced from other aligned reads on the place and the type of error where the base caller could have made a mistake by analysing only a single read.

Figure 34 shows this information – along with a new assembly that took place – using the knowledge that repetitive elements must be checked much more strictly. In this new assembly, read 5 and read 4 were rejected from the contig as the pathfinder tried to enter them, because they induced errors into the already existing contig formed by the reads 2-1-3 (when adding read 5) and 2-1-3-6 (when adding read 4).

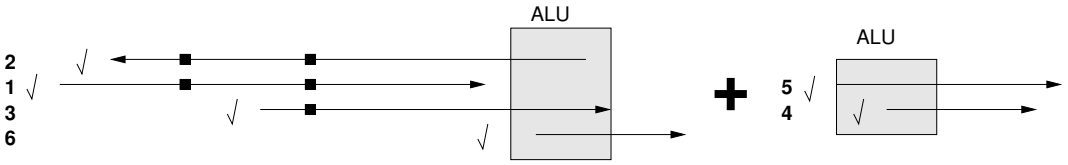

Figure 35: Example continued from figure 34 on page 71. Using strict signal checking in the ALU repeat region, unexplained errors in a first iteration of the assembler lead to the formation of two contigs in the second iteration. In this second iteration, no unexplained errors remain in the ALU region which is known as dangerous. Example continued in figure 39 on page 86.

The contig object failed to find a valid explanation to this problem when calling the signal analysis function – provided by the automatic editor – that tried to resolve discrepancies by investigating probable alternatives at the fault site. Figure 35 shows the result of the assembly that began in figure 34. There is not enough evidence found in the conflicting reads to allow a base change to resolve the discrepancies arising. There is also the additional knowledge that these non-resolvable errors occur in an area tagged as 'ALU-repeat', which leads to a highly sensitive assessment of the errors. Consequently, the reads 4 and 5 – containing repetitive sequence which the first contig object could not make match to its consensus – form a second contig as shown in the example.

Template size constraints

Large scale sequencing projects nowadays mostly use dual ended sequencing techniques as they provide valuable supplementary layout information for little cost overhead. With this technique both ends of a subclone (template) are sequenced so that normally two reads are generated for each template. Using short subclones of approximately 2 kilobases there is even a good probability that the reads will overlap in their ends.

The first constraint that can be deduced from this is the fact that – whenever the two reads belonging to a template can be inserted into the same contig – both reads have to be on opposite strands. As the approximate template size is also known, the second – equally important constraint – specifies a size range for both reads belonging to a template to be placed in the contig.

Large projects[12] are often sequenced using subclone libraries of different sizes. This can be done purposely to help building assembly scaffolds solely

[12] cosmid, BAC or even whole genome size

from overlap information and size constraints (Gene Myers' keynote on the German Conference on Bioinformatics (GCB) 99: "A Whole Genome Assembler for Drosophila"). It can also be a direct implication of the re-sequencing of certain parts of a project.[13] The insert size of a template is therefore a template specific information that must be read from experiment files as additional assembly information and cannot be given as general parameter.

If the read is the first of a template to be inserted, the contig will accept it without checking template constraints. In case the newly added read is the second, the contig will check the constraints discussed above and reject the read if either one is not fulfilled.

4.4.5 Consensus and consensus quality computation

Calculating the consensus and its quality is not trivial and many groups have proposed different strategies, ranging from the simplest sum-of-scores to algorithms using simulated annealing (Keith et al. (2002)), none of which is really optimal when base error probability values are available. The method developed for this thesis has a basic idea which is quite simple: go through each column of an alignment, calculate a *group quality* for each base and take the base with the highest group quality. The quality values of bases encode – if present – error probabilities, which in turn are used to give an assessment of the quality of the consensus base they are part of. Using the definition of alignments in chapter 2 where an alignment $\mathbf{L}$ is

$$\mathbf{L} = \begin{pmatrix} s_{11} & \dots & s_{1n} \\ \vdots & \ddots & \vdots \\ s_{k1} & \dots & s_{kn} \end{pmatrix}$$

with each element $s_{ij} \in \mathcal{A}^G$. A single column at position i within that alignment is defined as

$$\mathbf{C_i} = \begin{pmatrix} s_{1i} \\ \vdots \\ s_{ki} \end{pmatrix}$$

[13] for example to close gaps

which can also be seen as a simple vector. For the sake of simplicity, the vector only contains bases from $\mathcal{A}^g$ instead of $\mathcal{A}^G$ as endgaps (∇) do not really count as valid character.

Two more vectors can be defined by recording the error probability for each base and the direction of each sequence in the alignment:

$$\mathbf{P_i} = \begin{pmatrix} p_{1i} \\ \vdots \\ p_{ki} \end{pmatrix}$$

and

$$\mathbf{D} = \begin{pmatrix} s_1 \\ \vdots \\ s_k \end{pmatrix}$$

Note that the **D** vector does not need to be made for each column separately as – understandably – sequences in an alignment have exactly one direction which does not change for each base.

Should discrepancies in the called bases occur in this column, the probability for an error will be computed for each base character group separately. Two main points must be considered when setting up a formula to compute this probability:

- there must be a limit in the number of error probabilities used to prevent overrepresented bases – but with a high error probability – to take advantage over a low number of bases with low error probabilities. This is a problem especially encountered in non-normalised EST projects.
- probabilities coming from different sequencing directions should deserve a higher weigh than those coming from the same direction. This is a tribute to the sequencing methods currently used as there are sometimes sequence specific problems occurring.[14] Fortunately, when the other strand is sequenced, most – if not all – problems related to this will vanish.

As a first step, the probability values can be grouped together by base and

[14] e.g. the infamous AG-problem known with the ABI 373 and 377 machines where a G preceeded by an A is often unincisive

direction of the sequence in the alignment:

$$\bigvee_{c \in \mathcal{A}^b} P_i(c)^+ = \begin{pmatrix} p_{1i} \\ \vdots \\ p_{ki} \end{pmatrix} \qquad \text{with} \qquad s_{xi} = c \quad \text{and} \quad D(s_x) = +$$

and

$$\bigvee_{c \in \mathcal{A}^b} P_i(c)^- = \begin{pmatrix} p_{1i} \\ \vdots \\ p_{ki} \end{pmatrix} \qquad \text{with} \qquad s_{xi} = c \quad \text{and} \quad D(s_x) = -$$

E.g., all bases with an A from sequences in forward direction are taken together in the $P_i(A)^+$ group probability, bases with an A from sequences in reverse complement direction are taken in the $P_i(A)^-$ group probability.

Each of the $P_i(c)$ vectors is now sorted by value from low to high. The results are sorted vectors PS of error probabilities, for each base and each direction one.

$$PS_i(c) = sort(P_i(c))$$

For the next step, the vectors are pruned to contain a maximum of n lowest probability entries. The motivation for this step is loosely modeled after the entropy calculation theorem of the information theory. Grossly oversimplified, this theorem stipulates that the more information snippets sustaining one single fact are present, the less a single snippet of information itself is worth. Once the probability vector was sorted by value with the lowest values first, taking only the first n lowest values discards only those error probabilities that would not add much to the information itself.

In the next step, the inverse of the probability values are summed up, starting with the highest inverse and modifying the values with a decreasing scalar:

$$I_G(c)^{+/-} = \frac{n}{n}\frac{1}{PS_1(c)^{+/-}} + \frac{n-1}{n}\frac{1}{PS_2(c)^{+/-}} + ... + \frac{1}{n}\frac{1}{PS_n(c)^{+/-}}$$

The formula above ensures that the lowest error probability adds most of the information, with the following probabilities adding less and less until trailing off to zero. The probability for each directed base group is computed with the inverse of $I(c)$:

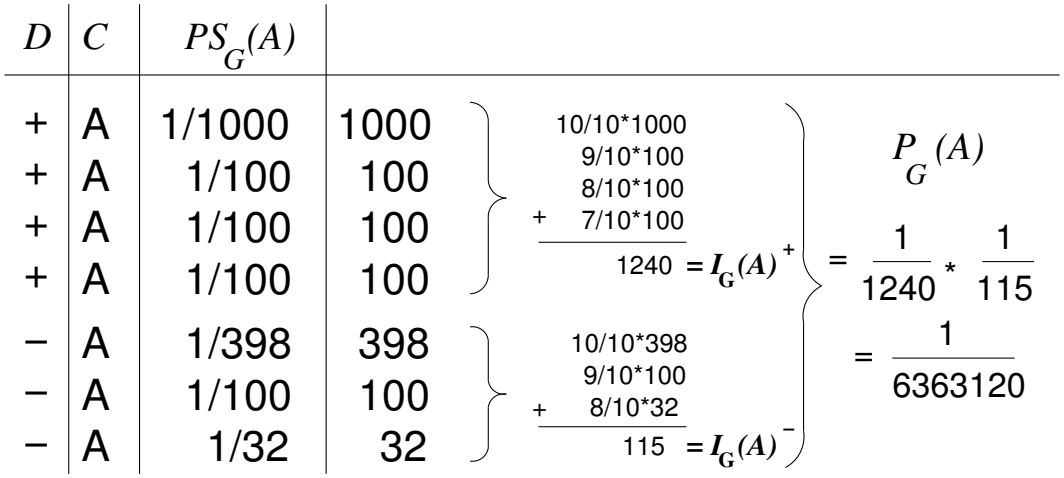

Figure 36: Simple example for calculating the group quality of base column. For the sake of simplicity, the vectors for the bases (C), their directions (D) and error probabilities ($PS_G(A)$) were already sorted by the probability values. The n factor has been set to 10 in this example, meaning that none of the vectors had to be pruned.

$$P_G(c)^{+/-} = \frac{1}{I_G(c)^{+/-}}$$

As last step, the total group error probability is computed by multiplying both directed group probabilities:

$$P_G(c) = P_G(c)^{+} * P_G(c)^{-}$$

with the special cases that $P_G(c)^{+/-} = 1$ for each of the non-existing directed quality groups and $P_G(c) = 0$ if both do not exist. See figure 36 for an example.

Once such a group quality has been computed for every base character of a column, the character with the highest group quality wins and is taken as consensus base for this column. When using IUPAC consensus characters, base characters with a certain minimum group quality – e.g. an error probability of less than 0.001 which equals a quality value of 30 and more – are taken to form the IUPAC consensus base.

4.5 Automatic editing

Up to this point, the `mira` assembler has put all reads into contigs, forming singlets for reads that could not be assembled anywhere. Assembling sequences with discrepancies in alignments induces the necessity to use other methods for dealing with possible base call errors that might be present in reads and introduce discrepancies or misassemblies in the assembly. This is entirely done by an incorporated version of the automatic editor developed by Pfisterer and Wetter (1999).

Previous solutions like the one presented by Xu et al. (1995) only used a dynamic programming algorithm and majority vote to adjudicate conflicting base positions. The main advantage of using an automatic editor is that decisions taken are based on the original trace signals and not on majority assumption. Different assumptions (hypotheses) on what could have gone wrong during the sequencing or base-calling process are established and inspected.

Although the exact methods and algorithms of the editor are not subject of this thesis, a short abstract on the strategy used is nevertheless included at this place to give an overview on the operations performed:

> The editor steps through the contigs column by column and searches for discrepancies between the bases of a column. Once a discrepancy is found, the editor will build an enlarged error region where it will test different base-calling error hypotheses in the reads present at this position. Enlarging the error region is necessary as clustered errors tend to obfuscate the true nature of the errors occurring in different reads because of an awkward multiple alignment. The most probable complex error hypotheses are then split into atomic fault hypotheses (AFH), each AFH describing an insert/delete/base change operation needed in one read to correct the whole error region. Each AFH is being tested by applying up to 30 different quality measures – depending on the type of atomic error hypothesis – directly to the underlying trace signal of a read.
>
> The signal quality measures analysed can be roughly classified into four categories: peak shapes, peak positions, peak distance and peak intensity. Relevant decision information is then extracted from the calculated measures by a neural network. The network decides whether the atomic fault hypothesis can be

confirmed by looking at the trace signals.

Although error hypotheses in regions involving complex base shuffling have a lower likeliness to be confirmed, they will be edited if they are the only possible solution that is supported by the trace signals. Using this methods ensures a maximum of safety for the assembler that editing decisions do not contradict the underlying trace signals. It can therefore – in contrast to simpler programs like ReAligner presented by Anson and Myers (1997) – also be seen as an improved method of realigning sequences to improve consensus quality.

4.6 Finding previously unknown repeats

Correct handling of repeats belongs to the most difficult problems an assembler has to perform. This section gives a short introduction to different types of repeats, the current methods to find and adjudicate them and present the method used in the assembler.

4.6.1 Repeat types

Repeats can be classified in the following categories:

1. Simple multiple bases where the repeat consists of a specific base present k time. Example: `AAAAAAAA...` Because of their usually short nature (the repeat will be far shorter than the average length of a read), simple multiple base repeats are not really problematic for an assembler unless they are very frequent. For longer stretches, the correct number of bases can be off by one or two. This is due to problems for any base-caller to correctly separate bases in trace signal stretches containing more than 5 to 6 equal bases when the trace quality is sub-optimal.

2. Micro-satellites where the repeat consists of a small number of bases present k times and where some copies might present point mutations. Example: multicopy `CG` in `CGCGCGAGCGCG...` or multicopy `CAG` in the sequence `CAGCAGCTGCAG...` Micro-satellites too are far shorter than an average read and therefore mostly unproblematic to assemble.

3. Short repeats where copies of a medium number of bases are separated by non-repetitive subsequences. The copies present an identity between 70% and 100%. In the human genome for example, ALU repeats are very

common. Although short term repeats are mostly shorter than the average read length, the sheer number of occurrences and the sometimes considerable identity can occasionally lead to misassemblies.

4. Long repeats are subsequences that contain up to several kilobases and where the repeat is present at least in two locations and the identity ranges from around 50% to 100%. Long repeats are the most difficult cases to assemble, especially if the identity exceeds more than 90% to 95% and the repeat itself is longer than the average read length.

The repeats of type 1 and 2 need no special handling routines as these are enclosed by (mostly) non-repetitive subsequences which ensure the correct placement of the read within an assembly. Repeats of type 3 (standard short term repeats) are sometimes harder to place as they are generally longer than repeats of type 1 and 2. But they have the considerable advantage that standard repeats are well known sequences, documented throughout literature and databases. Consequently, they can be searched for and tagged in the single reads before the assembly process takes place, giving the assembler the possibility to use the additional information gained during preprocessing.

From an assembler's point of view, the most annoying repeats are those of type 4. Segmental duplications – as an example for this type – are a special cases of extremely large repeats with sometimes several tens or even hundreds of kilobases. They play a fundamental role both in genomic diseases and gene evolution. Mutation and natural selection of duplicate copies of genes can diversify protein function, which explains why they are now seen as one of the primary forces in evolutionary change (Eichler (2001)). Bailey et al. (2001) note that they typically range in size between 1 and 200 kilobases and often contain special sequence features such as high-copy repeats and gene sequences with intron-exon structure. Another interesting – but from the viewpoint of an assembler rather annoying – recent discovery is the fact that, citing Delcher et al. (2002), "chromosome-scale inversions are a common evolutionary phenomenon in bacteria" and that some plants like the *Arabidopsis thaliana* contain large scale duplications on the chromosome level. On a similar level of annoyance is the fact that in grass genomes like rice, "most of the repeats are attributable to nested retrotransposons in the intergenic regions between the genes" (Wang et al. (2002)). Eichler (2001) observes that "exceptional duplicated regions underlie exceptional biology". For algorithms trying to resolve the assembly problem, they induce difficulties for in-silico computation and result in underrepresentation and misassembly of duplicated sequences in assembled genome.

4.6.2 Existing approaches

The most difficult task for an assembler consists in finding long term repeats in an assembly and preventing reads to be assembled at wrong locations within a contig. Suggestions to surmount this problem are sparse in literature and can be classified into two main approaches.

The first approach consists of relying on base probabilities only and prevent the alignment of reads that show too many discrepancies in high probability areas. This method is quick and its sensitivity can be easily adjusted. The advantages, however, are outweighed by the disadvantages this method inherently has: the assembler must rely solely on the ability of the analysing algorithms of the base-caller to correctly adjudicate each base upon the trace signal only. As good as current base-callers are nowadays, this cannot be guaranteed. Errors happen in the base-calling process and if the sensitivity of the assembler is set too high, the specificity of the repeat misassembly prevention mechanism decreases sharply: many non-repetitive reads will not align because their errors reach the repeat recognition threshold. Reads will thus not align although they might otherwise perfectly match, which in turn constitutes a handicap for the assembler when trying to build long contigs.

The second approach for repeat location assumes the shotgun process to produce uniformly distributed reads across the target genome. The solution to the long term repeat problem then consists in analysing read coverage in overlap graphs and rearrange read assembly in a way that the reads are distributed as uniformly as possible in the assembly (Kececioglu and Myers (1992)). The main problem of this method is the assumption of uniform read distribution of reads itself. A shotgun process is a stochastic method to gain reads from a genome. As in every stochastic process trying to reach a uniform distribution, the uniformity cannot be guaranteed throughout each segment of the genome. Additionally, chemical properties of the DNA itself sometimes inhibit the correct DNA duplication during the different cloning stages of the shotgun process, leading to skewed distributions of reads. In summary, assuming a uniform distribution is a working hypothesis that cannot be relied upon as only attribute.

4.6.3 Locating repeats through error pattern analysis

The assembler developed combines both methods described above together with information on template insert sizes and a fault pattern analysis algorithm. Contrary to the methods presented by Huang (1996) and Kececioglu and My-

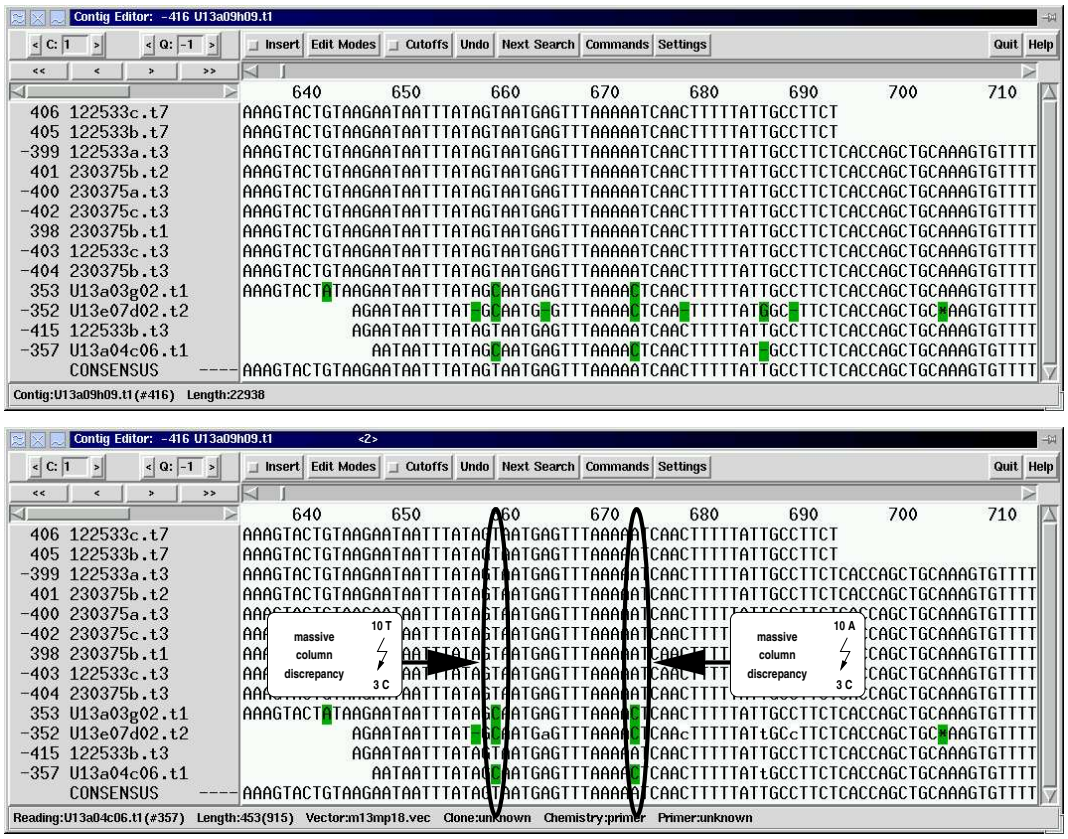

Figure 37: Example for a misassembly due to previously unknown long term repeats. Dark bases are discrepancies between reads and the actual consensus. The upper picture shows the initial alignment built by the assembler, the lower picture shows the same contig after the automatic editor made the corrections it could answer for. Observe the two heavy discrepancy columns automatic editor left untouched: the assembler will tag the bases of these columns as Possible Repeat Marker Bases (PRMB), dismantle the contig and reassemble the reads contained within.

ers (1992), the approach described within this section is able to handle complex repeat patterns which have more than two copies of extremely strong similarity. Here again, the use of an automatic editor during the assembly – which performs edits based on trace evidence only – is a major asset as it permits the duplication of the approach human finishers use when editing contigs.

A very important factor for any human finisher – when searching for misalignments due to repeats – is the observable circumstance that normally errors in reads which cause a drop in the alignment quality do not mass at specific col-

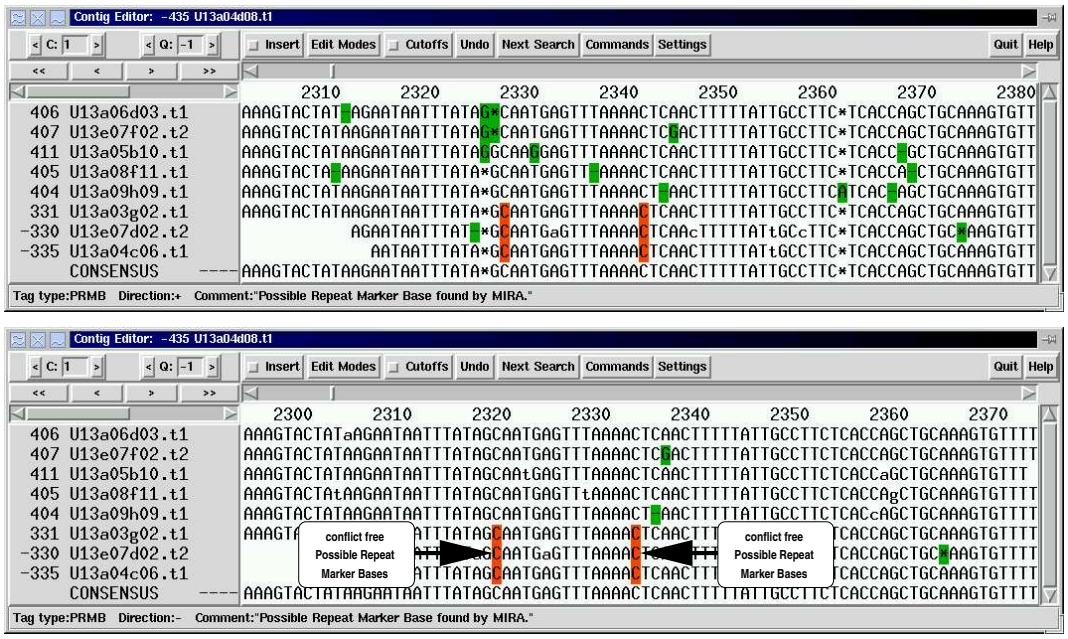

Figure 38: Resolved repeat problem: with the additional knowledge of possible repeat marker bases, the assembler is able to find the right solution when assembling repeats. The upper picture shows an alignment containing reads from figure 37 that were previously misassembled but are now at the right place. All the reads have been edited automatically at least once before they were reassembled. Observe that there are still a lot of discrepancies and base calling uncertainties contained in the reads. The lower picture shows the same alignment after automatic editing of the second assembly pass. This demonstrates that the correctly assembled repeats enabled the automatic editor to correct more errors and increase the quality of the assembly.

umn positions.[15] Repeats causing misalignments however will show up as massive column discrepancies between bases of different reads that simply cannot be edited away. The human finisher performs a search for patterns – like those shown in figure 37 on a symbolic level in an assembly to detect misassemblies.

The method developed is based on symbolic pattern recognition of column discrepancies in alignments to recognise long term repeats and non-marked short term repeats. For each column in an alignment, the method uses the same algorithms as for computing a consensus quality (presented earlier in section 4.4.5). But instead of computing a consensus, each column which contains contradict-

[15] although chemistry together with the sequencing direction of a read might play a minor role on the type of errors generated, but this has no real impact on the error distribution itself

ing bases with a group quality surpassing a predefined threshold (e.g. 30, which translates to an error probability of max. 0.001 for each base) is marked as potentially dangerous. By analysing the frequency of dangerous columns within a certain window length, the repeat detection algorithm can find and mark those columns that exceed an expected occurrence frequency.

Once most of the trivial base calling errors have been corrected by the automatic editor, even a single marked discrepancy column can be seen as a hint for a repeat misalignment if the coverage is high enough and the area has been built with reads sequenced from both strands of the DNA (see again figure 37). The bases allowing discrimination of reads belonging to different repeats are then tagged as Possible Repeat Marker Bases (PRMB) by the assembler. Contigs containing misassemblies are immediately dismantled and reassembled and during the subsequent reassembly, no discrepancy in alignments implicating these bases will be allowed and hence misassemblies will be prevented.

The reason for dismantling completely the contigs containing repeat induced errors in the assembly is the unpredictable effect the misaligned reads had on the alignment process. The most simple assumption could be that the misaligned reads could be inserted at another position of the assembly. However, in some cases the misaligned reads change the whole assembly layout and contig structure and lead to a totally different assembly. Misassemblies can be prevented best by the interaction of pathfinder and contig objects that were already described, the most sensible thing to do is to let these algorithms redo an assembly using the additional knowledge gained in this step. Figure 38 shows the example from figure 37 continued in which misassembled repetitive reads had single base columns marked as Possible Repeat Marker Base and subsequently reassembled at a totally different position, leading to a substantially different (and correct) assembly than the previous attempt.

4.7 Read extension

As the initial assembly used only high quality parts of the reads, further information can be extracted from the assembly by examining the end of the reads that were previously unused because the quality seemed too low. Although the signal-to-noise in read traces quickly degrades toward the end, the data is not generally useless. These 'hidden' parts of the reads can now be uncovered in two ways: (i) by uncovering parts of the reads that align to the already existing consensus and (ii) by uncovering hidden stretches of reads at the end of the

contigs that are not confirmed by a consensus.

Iterative enlargement procedures enable the assembler to redefine step by step the high confidence region (HCR) of each read by comparing it with supporting sequences from aligned reads. This usage of information in collateral reads is the assemblers major advantage over a simple base caller which has only the trace information of one read to call bases. It may also provide that extra linking leg needed to connect two previously disjunct contigs together.

4.7.1 Intra-contig and extra-contig read extension

Intra-contig extension is used to uncover reads and support areas of low coverage within a contig: the hidden sequence is aligned step by step to the existing consensus while allowing for a very low error in the alignment. This straightforward process is mainly used as a method to get more data confirmation than is available using only high quality parts. In most cases, the discrepancies found between the HCRs forming the existing consensus and the unaligned low confidence region (LCR) will be decided in favour of the HCR. But in some cases, especially in regions with very low coverage, one or more reads with LCR data can correct an error in the HCR stretch, e.g. when there is a local drop in the confidence values and signal quality of bases in the HCR stretch whereas signal quality and confidence values of the same bases in the LCR stretch seem better.

Extra-contig read extension uncovers LCRs at the ends of contigs and is used to extend the consensus to the left or to the right of a contig. LCR data present at the end of probably each read is not necessarily bad quality, but it is treated as hidden data: a region where the base caller calculated lower quality for the bases because it depended on the trace data of a single read. However, once reads have been aligned in their HCR, two or more stretches of lower quality can be used to uncover each other. The main purpose for this is to enable potential joins between contigs to be made in later steps. An example for this is given in figure 39.

4.7.2 Extension algorithms

Intra- and extra-contig extension is computed concurrently by analysing the overlap relationships characterised in the aligned dual sequences (ADS) computed in the earlier phase of the assembly. For every ADS which score ratio surpasses a defined threshold[16] and that aligns in same orientation, the exten-

[16] working with relatively high score ratios beginning with 80%

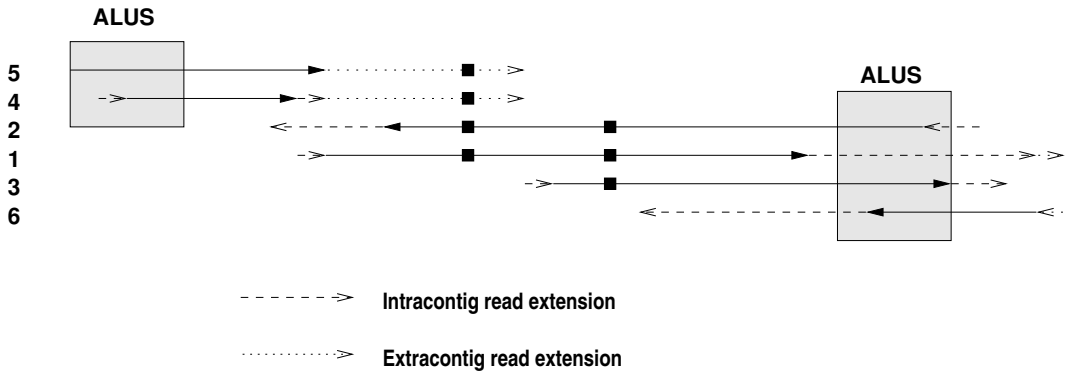

Figure 39: Example continued from figure 35 on page 73. Joining of contigs by extending the high confidence regions. Intra-contig read extension increases the coverage of contigs while extra-contig read extension allows existing contigs to be joined after a subsequent reassembly. The most striking difference to figure 34 on page 71 is that the two reads 4 and 5 are now assembled at a very different position.

sion algorithm tries to re-align the complete sequences including the previously unused low confidence region present at the end of each read.

Performing the extension operation at this stage of the assembly process incorporates the inestimable surplus value that the reads previously assembled into contigs will have been edited cautiously at least once by the automatic editor in their actual high confidence regions. The presumably few errors present in these parts of the read have thus been edited away where the trace signals and the alignment with other reads showed enough evidence to support the error hypothesis. Therefore less errors present in a sequence help the alignment algorithm to build more accurate alignments and thus will increase the score ratio of aligned dual sequences even with the LCR data included.

A window search is then performed across the new alignment – containing also the aligned LCR – to compute the optimal extension length of the HCR up to the point where the called sequence gets too bad to be correctly aligned. The chances for a long extension are increased because each read is present in many ADS objects, giving it many occasions to be extended.

There are two important advantages in extending reads using data from previously computed Smith-Waterman overlaps instead of aligning against the contig consensus:

1. short reads might be aligned at the wrong place in a contig, for example due to repeats. Should the LCR reach into non-repetitive sequence, the

read could not be extended. Using aligned dual sequence objects however will most probably ensure a correct overlap partner to be present.

2. reads that could not be inserted previously into contigs are given the chance to be extended and thus perhaps create an overlap with existing contigs.

The iterative enlargement procedure enables the assembler to redefine step by step the HCR of each read by comparing it with supporting sequences from aligned reads. This use of information in collateral reads is the major advantage of an assembler over a simple base caller, which has only the trace information of one read to call bases and estimate their probability.

4.8 Iterative cycling

Thompson et al. (1999b) showed in a large comparative analysis for multiple protein sequence alignment algorithms that iterative alignment algorithms offer improved alignment accuracy at the expense of computation time. As described in the previous sections, the assembler started the assembly process using sequence data with fairly high confidence and constructed – sometimes short – contigs of high quality. The quality of the contigs was then improved by automatic editing and eventual re-assembly in case of misassembled repeats due to formerly unknown repeats. The high confidence regions of the reads were then extended into the low confidence regions.

All these steps contribute to increase substantially the quantity and quality of usable sequence data that can be extracted from experimentally gained reads as they represent viable methods for removing inconsistencies during the assembly process. The new data can contain information crucial to the assembly, i.e. information that forces re-ordering of reads within contigs or even breaking up whole contigs to re-assemble the reads into new contigs. The single base-calling errors removed from the reads contribute to refine the pairwise alignments. This is a substantial advantage over simple iterative realignment approaches – like the round-robin algorithm from Anson and Myers (1997) or the method of Barton and Sternberg described in Chan et al. (1992) – that have to use sequences containing errors to build a correct alignment.

The operations necessary for reassembly and realignment are unpredictable and depend heavily on the type of genomic data that is to be assembled. To make the best possible use of the improved sequences, the assembler therefore restarts the whole assembly process from the beginning. This ensures an

optimal new assembly without risking errors introduced by unpredictable or wrongly predicted reordering operations.

The assembler will stop cycling should no major conflict be present in the contigs or should the newly gained information through automatic contig editing and read extension be minimal.

By cycling through the previous steps, the assembler iteratively corrects errors – like base-calling errors and misassembled repetitive repeats – that were made during previous steps and thus ensures the resulting contigs contain as few unexplainable errors as possible.

4.9 Modifications for EST assembly and SNP detection

For allowing the assembly of EST sequences and subsequent detection of SNP sites, the `mira` assembler was extended with some specialised algorithms: the `miraEST` modification of the standard assembler is specialised on using sequences gained by EST clone analysis to reliably detect and classify SNPs occurring in mRNA transcripts according to their SNP bases and strains (respectively cell types). Unlike other existing approaches like TRACE-DIFF (Bonfield et al. (1998)), polyphred (Nickerson et al. (2000)) or PTA (Paracel (2002c)), the method that was devised and implemented in the `miraEST` assembler does not first assemble all the sequences and then classify the SNPs. It rather uses information about potential SNP sites gained during the assembly to first cleanly assemble the transcript sequences by SNPs, strains, cell types and gene splice variants. Although it must be noted that not all SNPs or splice variants can be differentiated computationally: sequences with low transcript abundance and SNPs at sites having a low quality value cannot be differentiated. Also, splice variants that are a 100% subset of other variants will blend into their superset. Only in the last pass the resulting mRNA transcripts are assembled in a way that SNPs can be analysed, classified and reliably assigned to their corresponding mRNA transcriptome sequence.

The following sections describe both the problems and their solutions that were encountered respectively devised during the implementation and tests of the specialised EST assembly and SNP detection modules.

4.9.1 Coverage: meeting both extremes

Compared to assembling genome projects, EST projects can be much harder to assemble and evaluate. Genomes – or portions of it – are normally sequenced

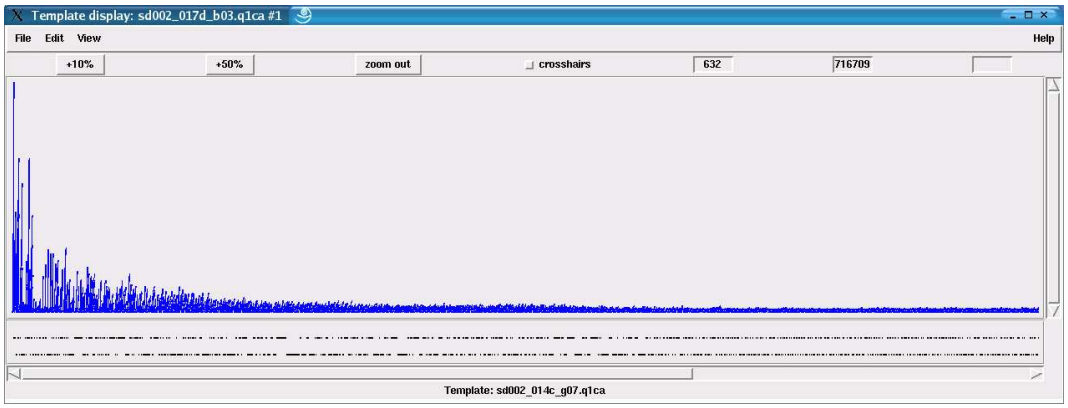

Figure 40: Coverage example in non-normalised EST project. Assembly after the first pass where all the sequences have been classified by true mRNA transcripts. Gene families and transcripts having SNP sites are in different contigs already.
The seemingly vertical lines on the left are in fact several hundreds of sequences one above the other. One can easily see that a few transcripts were sequenced extremely often while the majority of the mRNA transcripts has a low coverage.

a few times over and thus have a more or less comfortable coverage as well as clone insert sizes values for adjudication of problematic positions. EST projects on the other hand can have about any possible coverage: either very few reads or – on the contrary – a lot of reads per transcript. To make things even worse, some projects combine both: extremely high and extremely low coverage. The reason for this lies in the two possible ways a clone library is build: 1) in non-normalised libraries, all mRNA transcripts are collected regardlessly whether there are already duplicates in the library or not 2) normalised clone libraries, which use some special techniques to reduce duplicate transcripts.

Knowing that some genes or gene families are expressed more often than others within a cell, it is clear that working with non-normalised EST libraries poses the risk that there might be a few hundred or even thousand very similar transcript sequences of a gene / gene family within the sequenced project like, e.g., *cytochromes*. On the other hand, rarely expressed genes might be represented just once or even not at all.

The part of the `mira` assembler that showed to be the most susceptible to the increased coverage in EST projects was the pathfinder module with the "width-first-depth-last n, m-recursive look-ahead strategy" described in section

4.4.2. Understandably, a coverage of approximately 6 to 12 in genome projects with some repeat stretches scattered across the genome does by far not lead to graphs that are as dense as those build by, e.g., 1000 sequences that are very similar. The recursive nature of the path traversal algorithm led to disproportionate time and memory requirements. Different strategies were tried to tackle this problem. In the end, internal testing in the development phase showed that a time based cutoff-strategy combined with automatic search space reduction proved to be the most successful in terms of result quality, time and memory consumption. The time needed to traverse the connection graph is recorded for each recursion. If it surpasses a certain threshold, only the fraction of the possible targets corresponding to the ratio of time allowed versus time consumed is kept for the next recursion levels.

This strategy has, of course, a few drawbacks. First, its behaviour becomes that of a simple greedy algorithm for very large and dense graphs. Second, reproducibility is not guaranteed across different runs of the algorithm on different platforms or even on the same computer: the faster a computer is and the less other tasks are running, the more likely it will be that the search algorithm has enough time to analyse more solutions and not fall back to the quasi-greedy algorithm.

4.9.2 Detection of SNPs in genes and gene families

As is the case for detecting previously unknown repeats based on differences in single bases, the detection of SNPs is tightrope walk. Even in 2003, methods that propose redundancy based detection of SNPs were seen as state of the art (Barker et al. (2003)). Alas – as was shown in section 4.9.1 – EST projects tend to have extremely low "coverage" per mRNA transcript of rarely expressed genes, so this model does certainly not suffice. The algorithms conceived and implemented for this thesis take things a step further by combining the redundancy based approach with a symbolic pattern analyser and the usage of descriptive values that can be gained via the automatic editor from the trace signals (like, e.g., overall quality, possible miscall probability etc.).

The best solution for human experts when doing this work manually is to search for patterns on a symbolic level in an mRNA transcript assembly to detect potential SNPs. Here again, the important factor is the observable circumstance that – normally – discrepancies in reads which cause a drop in the alignment quality do not accumulate at specific column positions. Sequences from different mRNAs, however, may show column discrepancies between bases of

Figure 41: Snapshot of the EST sequence assembly after the first iteration (visual representation by the means of the gap4 program). All sequences were assembled together. After the assembly, `mira` searched for unresolved mismatches with good signal qualities, tagging entire columns as 'dangerous' potential SNP site for the next iteration (shown in bright red).
As there are unresolved problems in this assembly, `mira` will dismantle that contig and reassemble the sequences immediately, this time using the information gained about the potential SNP sites in the previous assembly to correctly discern between different mRNA transcripts having different SNP variants.
The black rectangle amidst the sequences depicts the three trace signal extracts that have been exemplarily shown below. One can clearly see that there will be at least three different mRNA transcripts to be build, based on the fact of the double-base mutation in the middle of the box, one reading `CC`, the next `CT` and the last `TC`.

different reads that simply cannot be edited away: these are the potential SNP sites. See figure 41 for such a typical real life example.

Adjudicating now whether discrepancies between similar EST sequences are significant – and thus a polymorphism – or not relies therefore much more on the underlying traces and their quality than for genome assembly.

Based on this, the symbolic pattern recognition methods developed for recognition of column discrepancies in repeat alignments in section 4.6.3 were expanded to allow detection of SNPs. The algorithms developed base on the same working principles, i.e., the most important criterion are the group qualities

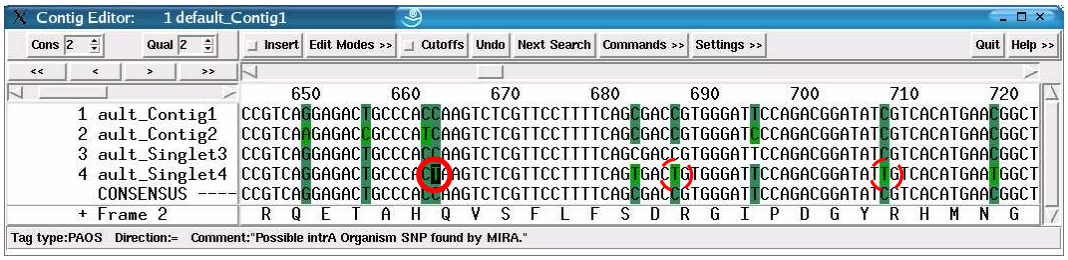

Figure 42: The last step of the EST assembly: merging the mRNA transcript sequences gained in the previous steps.
This example is continued from figure 41 and does not contain strain information yet. The sequences had been put into 4 different mRNA transcript sequences, two of them (named *default_Contig1* and *default_Contig2*) having multiple experimental sequences, the two others (*default_Singlet1* and *default_Singlet2*) consisting only of one sequenced probe.
Interestingly enough, most of the SNPs shown in this example will not cause a change in the amino acids of the resulting protein, with one notable exception: the SNP of *default_singlet4* at base position 662 (solid red circle) causes a `TAA` codon to be expressed, which is a stop signal. The SNPs of the same sequence at position 686 and 707 (dashed red circle) would cause mutations in the amino acid sequence, but are – because of the `TAA` mutation earlier – in the 3' UTR of the mRNA.

that can be calculated for different bases in a column. Under those circumstances, even a discrepancy caused by a single base in a single column of an alignment can be seen as a hint for a SNP site, i.e., if the base probability values of the bases in the immediate area are high and the signal traces do not allow an alternative sequencing error hypothesis. The bases allowing discrimination of reads belonging to different mRNA transcripts will be marked as possible SNP marker bases (PRMB) in an intermediate first pass by the assembler. Technically speaking, the bases tagged as PRMB in this pass are – of course – not marker for possible repeats but marker for possible SNPs and should be named possible SNP marker base (PSMB). However, their function is exactly the same and handled by exactly the same routines.[17]

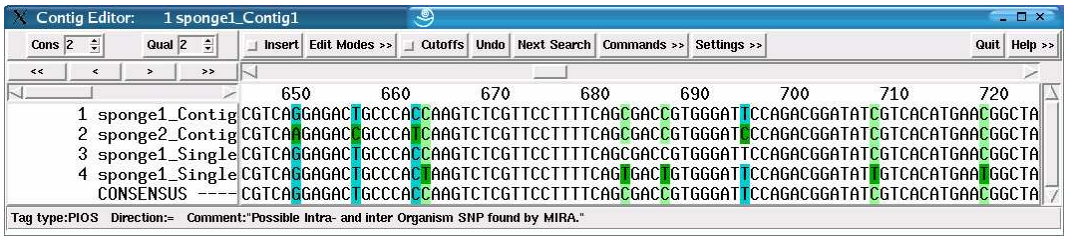

Figure 43: In contrast to figure 42, the input sequences were given strain information to show the effect when two different organism strains (named 'sponge1' and 'sponge2') are sequenced and analysed. In this example, `miraEST` classified the SNPs into two categories: PROS (shown in light blue) for SNPs that occur only between strains / organisms (e.g. column 661) and PIOS (shown in light green) for SNPs that occur both within a strain as between different strain (e.g. column 662).
One can now clearly see, that the mutation causing a stop codon to be expressed (position 662) is only present in one transcript from 'sponge2'.

4.9.3 Classification of SNPs

Basically, one can differentiate between three distinct types of single nucleotide polymorphisms (SNP) when analysing transcripts from one or several organisms, strains or cell types.

1. **PAOS** Polymorphisms that occur within a single organism or cell transcriptome are tagged as "**P**ossible intr**A** **O**rganism **S**np"

2. **PROS** Polymorphisms that occur between different organisms or cells are tagged "**P**ossible inte**R** **O**rganism **S**np"

3. **PIOS** Polymorphisms that occur both within and between organisms (respectively cell types) are tagged as "**P**ossible **I**ntra- and inter **O**rganism **S**np"

During an optional last assembly pass, the `miraEST` assembler will merge almost identical – strain and SNP separated – transcriptome sequences from the previous passes for a last alignment. Such an alignment shows SNP differences between the mRNA sequence transcripts. The transcript sequences used for this final assembly stage will be precisely classified and assembled at least by SNP types and – if the information was present – by organism / strain / cell

[17] This is one of the more prominent places where it shows that the EST assembler is a sibling of its genome pendant.

type in the previous passes. Consequently, it is reasonable to assume that the transcript sequences used at this stage are pristine, that is, they code existing proteins.

It is important to note that this step, like the whole process performed by `miraEST`, is still an assembly and not a clustering step, i.e., sequences composed by different exon structures or which contain large indels will not be assembled. The results obtained here are nevertheless important in a sense that they allow analysis and classification of the SNP types of nearly identical mRNA sequences which occur in one or several sequencing assembly projects.

Figure 43 illustrates the assembly of two strains. SNPs are classified into all three categories, the example figure showing two of them (PAOS and PIOS). Sequences without strain information will also have the bases tagged, but only as PAOS as they will be assigned to a *default* strain. Comparing figures 42 and 43 exemplarily shows the differences in classification with and without supplied strain data. Figure 42 shows the assembly of one strain, each SNP is therefore classified as as PAOS. Figure 43 shows the simulated assembly of two strains. SNPs can now be classified into all three categories, the example figure shows two of them (PAOS and PIOS).

Poly-base stretches of As or Ts marking the end of translation are left as help for visual analysis, but are not used computationally to decide whether a position contains a SNP or not.

5 Results and discussion

> *"Never trust a tall dwarf. He's lying about something."* (Solomon Short)

The principles presented within this thesis have been implemented in the `mira`[1] assembler. Development of the assembler started in 1997, the 1.2 version of `mira` containing the methods discussed in this thesis was used at the Institute of Molecular Biotechnology (IMB) Genome Sequencing Centre Jena (GSCJ) and several other public institutes or private companies after having passed intensive testing in fall 1999, during which the overall concept has been refined. Since November 1999, the assembler was subject to constant scrutiny, performance improvement algorithmic redesign, the current version 2.2.8 was released in May 2004.

As stated in chapter 1, the aim of this thesis was to reduce assembly errors caused by repetitive sequences as well as increase the reliability of consensus sequences derived from automatically assembled projects. During the course of this thesis, it became apparent that two type of assembly projects could be taken to evaluate whether the methods and algorithms developed for this thesis could meet the expectations: i) assembly of highly repetitive eukaryotic genome sequences and ii) assembly of non-normalised EST projects, which contain per se a high degree of very similar mRNA sequences.

This chapter presents the results in three sections: the first presents and discusses qualitative results of the `mira` assembler for genome assembly, the second section gives an overview on results achieved for EST assembly and SNP detection and the third section discusses the results obtained in general.

5.1 Genome assembly

While the data sets used for the evaluation of the genome assembly in 2000 seem quite small for today's standards (between 30 and 130 kilobases), they

[1] MIRA: Mimicking Intelligent Read Assembly

covered the whole range of problems that arose while performing a shotgun assembly of purportedly hard to assemble eukaryotic genome sequences.

In cooperation with the Genome Sequencing Center in Jena (IMB Jena), seven projects from human chromosome 21 were taken for this evaluation. The small GenBank sequence AF045449 was chosen because it had been a small though reportedly hard project to finish due to repetitive regions with additional multiple base inserts. The other projects were randomly chosen from standard medium sized projects that took longer than average to finish, contain a moderate to high number of repetitive regions and had been submitted to GenBank at the time of the study (2000).

As Chen and Skiena (2000) noted and Miller (2001); Bray et al. (2003) confirmed later, it proves to be a non-trivial problem to compare genomic DNA sequences as are results of different assembly programs on a given project. General tools that permit this task are focussed on multiple alignments of protein sequences[2], but are not well suited for comparison and characterisation of differences in long DNA sequences. First steps toward this goal were published by Delcher et al. (2002); Bray et al. (2003) but still need to be improved regarding the automation of difference analysis reporting.

For this reason, the comparison was done both automatically and manually. The submitted, human-edited and finished sequence was taken as golden standard and compared with different assemblies using the cross_match program from Phil Green. Areas in which the consensus sequences showed discrepancies were submitted to visual inspection of differences between the resulting assemblies using the GAP4 program from the Staden package. Relevant results for the evaluation assembly projects are shown in tables 5 and 6 to demonstrate the effectiveness of the methods presented.

The `mira` assembler was compared with two of the most widely used assemblers freely available at the time of the survey: the assembler integrated into the sequence assembly package GAP4 from the Staden group at the MRC LMB in Cambridge (UK) and the PHRAP assembler developed by Phil Green. Incidentally, these assemblers nowadays (2005) still belong to the most widely used in sequencing projects around the world.

The data was gained by gel electrophoresis on ABI 377 machines. Each project ran through an entire assembly cycle with the respective tools consisting of ABI-basecall → PREGAP4 → MIRA/EdIt for the MIRA test, ABI-basecall

[2] see Notredame (2002) for a review of state-of-the-art algorithms,Thompson et al. (1999a) and Lassmann and Sonnhammer (2002) for a evaluation of some of these tools and Morgenstern et al. (2003) for the description of a web-based solution

Table 5: Genome projects used for benchmarking. Only the most prevalent families of repeat types present in the project are given, most of the types also consist of several subtypes that were summed up, e.g. AF045450 contains in the Mlt family repeats of type Mlt1A2, Mlt1C, Mlt1E and Mlt1F while the Herv family is represented by Herv16, Herv17 and HervL.

GenBank accession number	Contig length	Number of repeats	Repeat fraction of sequence	Prevalent repeat families	Number of reads
AF045449	32,613	36	17.9%	Alu,L2,Mir,Mlt	832
AF045450	40,205	54	62.6%	Alu,Herv,Mlt	941
AF129076	42,051	39	32.2%	Alu,L1,Mer,Mlt	2,070
AF015722	47,162	71	15.6%	Alu,L1,Mer,The1b	850
AF222685	85,040	179	31.5%	Alu,L1,L2,Mir	2,408
AF165178	88,775	422	58.6%	Alu,L1,L2,Mlt	3,452
AF130248	137,074	157	35.7%	Alu,L1,L2,Mir	3,636

→ PREGAP4 → GAP4/cycle[3] for the GAP4/cycle test and PHRED-basecall → PREGAP4 → PHRAP for the PHRAP test.[4] Please refer to the respective user manuals of the software packages for a detailed description of the default parameters. Only ALU repeats were tagged during the PREGAP4 process, no read template information was available. GAP4/cycle and PHRAP assemblies were run using standard parameter sets from the IMB Jena sequencing centre. `mira` was started with default parameters as described in appendix B.

The assemblies were compared by building a standard GAP4 consensus for contigs longer than 1,500 bases in regions with a coverage $\geq$3. This ensures that very low coverage uncertainties and contigs too small to be useful in subsequent contig joining steps are clipped away in this study. The results are shown in table 6.

In six out of the seven projects, the consensus produced by `mira` has a lower error rate (errors per kilobase consensus sequence) than the GAP4/cycle assembly. The `mira` consensus of five out of the seven projects has a less errors per kilobase than the PHRAP assembly. It is interesting to note that no multiple adjacent base errors were found in the `mira` assemblies, while two PHRAP

[3]GAP4/cycle is a script performing several GAP4 assemblies with decreasing strictness

[4]PHRAP uses base qualities for the assembly, MIRA/EdIt can use them if present and GAP4/cycle does not

Table 6: Comparison of MIRA/Edit, GAP4/cycle and PHRAP assemblies.

Project	Assembler	Contigs ≥ 1500 bases	Bases coverage ≥3	GenBank entry covered	Number of edits	Single base errors	Multiple base errors	Accuracy	Errors per 10 kb
	MIRA/EdIt	3	33,340	95.3%	3,979	27	0	99.9132	8.68
AF045449	GAP4/cycle	8	27,496	71.5%	-	27	0	99.8842	11.58
	PHRAP	2	34,538	98.9%	-	32	3*15	99.7614	23.86
	MIRA/EdIt	2	40,114	96.9%	7,588	16	0	99.9589	4.10
AF045450	GAP4/cycle	11	37,458	79.4%	-	9	0	99.9718	2.82
	PHRAP	2	40,191	99.9%	-	54	1*4	99.8557	14.43
	MIRA/EdIt	4	43,681	94.1%	12,081	6	0	99.9848	1.52
AF129076	GAP4/cycle	15	44,085	69.7%	-	7	0	99.9761	2.39
	PHRAP	6	45,427	97.1%	-	7	0	99.8286	1.71
	MIRA/EdIt	6	44,516	89.5%	7,455	29	0	99.9290	6.87
AF015722	GAP4/cycle	15	36,562	59.4%	-	29	0	99.5394	10.35
	PHRAP	1	47,744	99.9%	-	34	0	99.9278	7.22
	MIRA/EdIt	7	84,230	89.9%	23,499	26	0	99.9660	3.40
AF222685	GAP4/cycle	28	58,896	52.2%	-	25	1*2	99.9436	5.86
	PHRAP	3	85,744	99.2%	-	5	0	99.9941	0.59
	MIRA/EdIt	10	89,971	90.8%	40,451	195	0	99.7581	24.19
AF165178	GAP4/cycle	26	58,874	48.9%	-	145	1*4+1*2	99.9965	34.79
	PHRAP	10	82,357	92.8%	-	104	0	99.8737	12.60
	MIRA/EdIt	9	130.246	94.5%	14,937	12	0	99.9908	0.86
AF130248	GAP4/cycle	28	108,102	68.4%	-	18	0	99.9808	1.92
	PHRAP	6	137,803	>99.9%	-	17	0	99.9876	1.24

projects (AF045449 and AF045450) and two GAP4/cycle project (AF222685 and AF165178) were affected from this. In the case of the PHRAP AF045449 it decreases dramatically the quality of the assembly. Visual inspection of the 15 base error stretches in AF045449 showed that different subtypes of repeats with 86% homology had been mixed and wrongly assembled with other similar repeats during assembly by PHRAP, an error that `mira` did not make at all.

In general, `mira` built less contigs than GAP4/cycle and slightly more contigs than PHRAP. `mira` projects have significantly more bases of the final project covered than GAP4/cycle, but a slightly less total coverage than PHRAP in the corresponding GenBank entries. On the other hand, the total number of errors present in the consensus is significantly lower most of the time with the `mira` assemblies than with the PHRAP assemblies. In fact, only two of the seven `mira` projects (AF222685 and AF165178) contain more single base errors in the coverage consensus than the corresponding PHRAP projects and `mira` never made multiple base errors – resulting from misassembled sequences in repeat stretches – whereas PHRAP and GAP4/cycle did.

These results are confirmed by several private communications with different sequencing groups showing that `mira` delivers correct assemblies with significantly less contigs and more coverage than the standard GAP4/cycle assembly. Compared to PHRAP, the number of contigs is higher and the GenBank entry coverage slightly lower. However, human finisher frequently report that routine use of PHRAP is often complicated by misassemblies, especially in the case of low redundancy sequencing (skimming, working draft), or high in case of high degrees of similarities ($\leq$ 1%) between different repeats. This shows that the strategy developed in this thesis to assemble high confidence regions first – and strict checking of repetitive DNA stretches known as problematic with routines for signal analysis – avoids mistakes in the assembly, while preparing good contigs for possible manual or automatic join operations that are to be performed later.

5.2 EST assembly

In comparison to an assembly of a genomic sequence, the assembly of an EST project has two notable differences: i) mRNA of genes is quite short, one kilobase is already considered long and two kilobases are rarely reached (and the contigs built will not exceed this length) and ii) the degree of similarity will be extremely high for some gene families like, e.g., cytochromes. The challenge

Table 7: Summary of results from EST assembly of sponge, dog and grapevine sequences.
Step 1: result sequences are transcripts separated by SNPs, but not by strain. The number of contigs, the classification numbers on maximum and minimum coverage (and the times they occurred) within the contigs as well as the number of singlets give a rough idea about the asymmetrical distributions of EST reads in the different contigs.
Step 3: 'assembly of pristine mRNA transcripts' to analyse SNP sites and types. The transcripts sequences gained there can be seen as a consensus of the (hopefully) pristine transcripts gained in the previous steps of the assembly. Classification of SNPs (see also the subsection of the same name in section Methods and Algorithms) is also performed in this step: 'Intra' means that SNPs occur only with a strain or cell type, SNPs of type 'Inter' occur only when comparing different strains or cell types, and the 'Intra and Inter' SNP type is a combination of the first two types.
Intermediary results from step 2 are not shown as sponge and dog do not use this step and the grapevine results are too extensive.

	Sponge	Dog	Grapevine
Input sequences	9,747	10,863	32,776
Strains / cell types	1	1	10
Step 1 : transcript SNP separation assembly			
Total transcripts	4,401	5,921	12,380
thereof singlets	3,151	4,204	7,904
thereof contigs	1,250	1,717	4,476
Max cov / occured	145 / 1	106 / 1	812 / 1
Min cov / occurred	2 / 637	2 / 885	2 / 2,143
Total transcript len.	3,342,596	3,941,124	7,082,719
Step 3: transcript SNP classification assembly			
Total unified transcr.	4,077	5,901	8,547
thereof singlets	3,780	5,811	6,131
thereof contigs	297	90	2,416
thereof with SNPs	285	81	2,103
Total transcript len.	3,120,847	3,897,635	4,872,333
Transcript SNP types			
Intra strain / cell	2,158	461	959
Inter strain / cell	–	–	1,505
Intra and Inter s. / c.	–	–	7,221
Total SNP sites	4,653	927	9,685

for an assembler is to correctly recognise splice variants of the same gene, but also to discern between the mRNA generated by different gene copies or by different allelic variations that sometimes have as only difference a single base polymorphism (SNP).

Three very different projects were taken to present results reached through an accurate assembly and subsequent SNP scanning of transcript sequences with the `miraEST` assembler. The non-normalised libraries contain ESTs sequenced from the plant *Vitis vinifera* Linnaeus (Plantae: Spermatophyta: Rosopsida / Dicotyledoneae), and two animal taxa, the sponge *Suberites domuncula* Olivi (Metazoa: Porifera: Demospongiae), and the vertebrate *Canis lupus familiaris* Linnaeus (Metazoa: Chordata: Vertebrata).

Although these three multicellular organisms are eukaryotes, they are only distantly related. In general, plants split off first from the common ancestor, approximately 1,000 million years ago (MYA). Later the Metazoa evolved, (700 MYA with Porifera as the oldest still extant phylum, and finally the Chordata appeared (500 MYA, reviewed in: Kumar and Rzhetsky (1996); Müller (2001)). Until recently, the Porifera were an enigmatic taxon, see Müller (2001). Only the analyses of the molecular sequences from sponges, both cDNA and genomic ones, gave strong evidence that all metazoan phyla originated from one ancestor. Therefore, ESTs from this taxon were included into this evaluation in order to obtain a first estimation about the abundance of particular genes in such a collection.

The assembled ESTs from the *S. domuncula* (sponge) were taken to allow a further elucidation of the evolutionary novelties that emerged during the transition from the fungi to the Metazoa. Likewise the data from the *V. vinifera* (grapevine) and the mammal *C. lupus familiaris* (dog) should provide an understanding of the change of gene pool in organisms under domestication. While the dog and sponge project had only ESTs sequenced from one strain (respectively cell type), the grapevine project had ESTs that were collected from a multitude of cell types, ranging from root cells to berry cells. Table 7 shows an overview of these projects together with some of the more interesting statistics of the assembly.

Depending on the projects, the sequences used were obtained by capillary electrophoresis on ABI 3100 or ABI 3700 machines with each project having specific sequencing vectors. For this study, all project sequences were preprocessed and cleaned using standard computational methods: TraceTuner 2.0.1[5]

[5]TraceTuner is from from Paracel Inc.

for extracting the bases. Datasets were cleaned by using PFP as described in Paracel (2002a): masking of known sequencing vectors, filtering against contaminant vectors present in the UniVec core database, filtering of possible *E. coli* and other bacterial contamination and masking of poly-A / poly-T tails in sequences. Repeats and known standard motifs were not masked as these are integral parts of the data and contain valuable information. Sequences that were shorter than 80 bases were removed from the projects. The remaining sequences used in the three projects total 53,386 sequences with 54,303,071 bases.

For each project, the `miraEST` assembler's integrated standard parameter set was used. This set is configured as a three pass assembly :

1. classification of the sequences by SNP type using all sequences from all strains / cell types etc. The motivation for performing a first pass that separates only by SNP and not also directly by strain / cell type is the simple observation that the assembler still can find useful SNP on rarely expressed genes when looking at the entirety of the available data within alignments. Interesting sequence features found in this first pass are valuable for the two subsequent passes in which the algorithms will benefit from them.

2. additional step if strain information is available: separation of the sequences by strain (resp. cell type) and SNP. This results in clean mRNA transcripts sequences that represent the actual state of the transcriptome of a strain / cell type as it is present in the clone library. Although the results of this step are interesting on their own, their major importance is the fact that they are used as pristine input for the following third pass.

3. production of a combined SNP-strain assembly. If strain information was available, this step uses results from step 2, else from step 1. The result of this assembly has the exact SNP positions and types tagged in the mRNA transcript sequences that form an alignment of the resulting consensus.

Each pass had a standard set of options activated to enhance the preprocessed reads by trimming for quality, unifying areas of masked bases at read-ends, clipping sequencing vector relicts and tagging remaining poly-A / poly-T stretches in sequences (see section 4.1 and appendix B for more details). Trace data was used in the assembly to edit base calling errors in sequences and assess bases and possible SNP sites when available. Table 8 shows computer requirements in conjunction with project complexity aspects.

Table 8: Runtime and memory consumption of the study projects using an Intel 2.4 GHz Xeon P4/HT PC with 512 K L2 cache and 2 G RDRAM. Comparison of the sponge and dog project, which have roughly the same number of sequences showing a clear relationship between the runtime and the number of detected contig reassemblies (which were triggered by newly detected SNP sites).
The reduced runtime from step 1 to step 2 is due to potentially problematic regions with SNP sites that were detected in the first step. These SNPs give additional information to the second step, which then prevents misassemblies that involve those sites. Hence the lower number of reassemblies reduced runtime.
In general, step 3 has less transcript sequences to assemble than step 1 and step 2, also leading to reduced runtimes.

	Sponge	Dog	Grapevine
Peak memory usage	250 M	280 M	1.7 G
Runtime in minutes			
Step 1	27	14	735
Step 2	20	10	101
Step 3	3	4	35
Total	137	69	871
Number of contig reassemblies			
Step 1	577	250	3,827
Step 2	51	18	1,927
Step 3	0	0	0
Total reassemblies	628	268	5,754

Comparing the projects led to some interesting insights both on the behaviour of `miraEST` and on the data itself. Although the sponge and the dog projects have about the same numbers of sequences used as input (9,747 versus 10,863), the assembly runtimes of the sponge project took about twice as long to complete than the dog project. When analysing log files and intermediary results from both projects, two main causes were found for this behaviour:

1. the more assembled transcript contigs contains SNPs, the more the assembler will have to break those up and reassemble them in step 1 and 2, leading to higher assembly times.

2. the more similar sequences from one or several gene families are present, the higher is the probability for an increased number of iterations needed to get the transcripts assembled cleanly.

Both these factors can be seen as predominant indicators for the complexity of a project. The sequences of the sponge project contain 285 mRNA transcript contigs (7.0% of the transcripts) with SNPs. These total 2,158 SNP sites, which is about 7.5 SNPs per mRNA transcript that contains SNPs. The sequences of the dog project however lead to only 81 mRNA transcript contigs (1.4% of the transcripts) with SNPs. These total only 461 SNP sites, which is about 5.7 SNPs per mRNA transcript that contains SNPs. The sequenced sponge EST sequences therefore not only contain more transcripts with polymorphisms than the dog sequences, they generally also contain more SNPs per transcript.

Comparing the grapevine project with the two other projects also yielded some interesting discoveries. First, the contig with the maximum coverage that occurred in step 1 contained 812 reads compared to 145 for the sponge and 106 for the dog. The grapevine data also contained several additional of these high-coverage contigs, which meant that the this project contained a number of genes or gene families that were, in absolute numbers, more expressed – and thus sequenced – than in the dog and sponge project. The second interesting discovery was the decrease in total transcripts from step 1 to step 3: the sponge project had a 7.4% reduction (from 4,401 clean transcripts to 4,077 unified transcript consensi) and the dog only 0.3% (from 5,921 to 5,901), but the grapevine project had a 31% reduction (from 12,380 down to 8,547) in the number of transcripts. This meant that many gene transcripts of the grapevine project differed only in a few SNP bases and were assembled together in step 3, forming transcript consensi which allowed the classification of SNPs whether they occur within a cell type, between different cell types or both. On the other hand, the 9,685 SNPs found were dispersed over 2,103 transcripts, which is about 4.6 SNPs per transcript containing SNPs and therefore less that the sponge or even the dog project.

The exact reason for these high transcript redundancy numbers in this project is currently under investigation, but preliminary results indicated that a significant number of almost identical common basic housekeeping genes are expressed and were sequenced in different cell types and that several of them contain SNPs. For example, a transcript family was found in 9 out of 10 cell types that was formed by 147 Metallothionein transcripts with no less than 98 positively identified SNP sites over a length of 650 bases. The SNPs are in the

coding region and the 3' UTR, with many of the SNPs leading to a mutation in the amino acid sequence of the protein.

5.3 Discussion

Very early in the development process it was discovered that using high quality sequence data first in the assembly process was a very viable way to proceed as it substantially reduced computing time. This permitted to reinvest this saved time into other algorithms that increased the actual quality of the final results: resolving detected misassembly conflicts, analysis and detection of previously unknown relevant sequence features (e.g. repeat marker bases, SNPs, etc.) and detection and elimination of conflicts caused by misassemblies. The ever increasing computing power permitted the design of exact iterative algorithms instead on relying on makeshift algorithms when assembly problems occurred. That is, it was a clear choice not to trade off quality for speed when the loss in quality was deemed to be substantial. Furthermore, integrating an automated trace editor into the assembly process was the correct choice as results showed that spurious base-calling errors are reliably detected and removed in an alignment and the assembler can also use the trace analysis routines to perform in-depth and multi-level analysis on problematic regions in alignments.

Presently no other assembly system – be it for genomic or transcript data – contains a comparable mix of algorithms that enables the assembler to dependably detect by itself and use the information about special base positions that differentiate between repetitive stretches within sequences as is the case for repeat marker bases (RMB) in genomic assemblies or single nucleotide polymorphisms (SNPs) bases in EST assemblies. Reiterating the stance regarding the importance of discovering such base positions during the assembly: they allow the assembler to perform a reliable separation of almost identical sequences – which may ultimately differ only in one single position within two single sequences – into their true original genomic sequence or transcriptome. This is significantly more sensitive and specific than other methods like relying on base qualities alone (PHRAP) or the one presented by Tammi et al. (2002), which needs at least two differences in reads to distinguish them from sequencing errors.

Additionally, corrections performed by the integrated automatic editor resolve errors in alignments produced by base-calling problems. This makes RMB or SNP detection much less vulnerable to sequence specific electrophore-

sis glitches and base-calling errors as is the case for, e.g., the AG-problem known with the ABI 373 and 377 machines where a G preceeded by an A is often unincisive or only weakly pronounced.

The `miraEST` assembler was developed concurrently to the `mira` version for genome sequences presented in Chevreux et al. (1999, 2000), which enabled to use basic algorithms for both branches of the assembly system. This also allowed to concentrate on developing and improving those algorithms that are specifically needed to tackle the slightly different assembly problems of genome and EST sequences once the basic facilities were in place.

In contrast to other assemblers or SNP detection programs – like `phrap`, `gap4`, `pga/pta`, the TGICL system, `polyphred` or `autoSNP` – the approach devised uses strict separation of sequences according to the real signals in the trace chromatograms. As the results presented in this chapter have shown, this is a reliable way to ensure that the consensus sequences produced as result correspond to the real genome or transcriptome sequence.

This method permits to use these results directly for the design of further investigative studies with high quality and precision requirements like, e.g. the design of oligo probes for specific SNP detection in clinical micro-array hybridisation screening experiments.

The possibility to export the assembled projects together with the analysis of RMB or SNP sites to a variety of standard formats, e.g. `gap4` directed assembly, `phrap` .ace, or even simple HTML pages as shown in figure 44 opens the door to visual inspection of the results as well as integrating the tool into more complex and semi-automated to automated laboratory workflows.

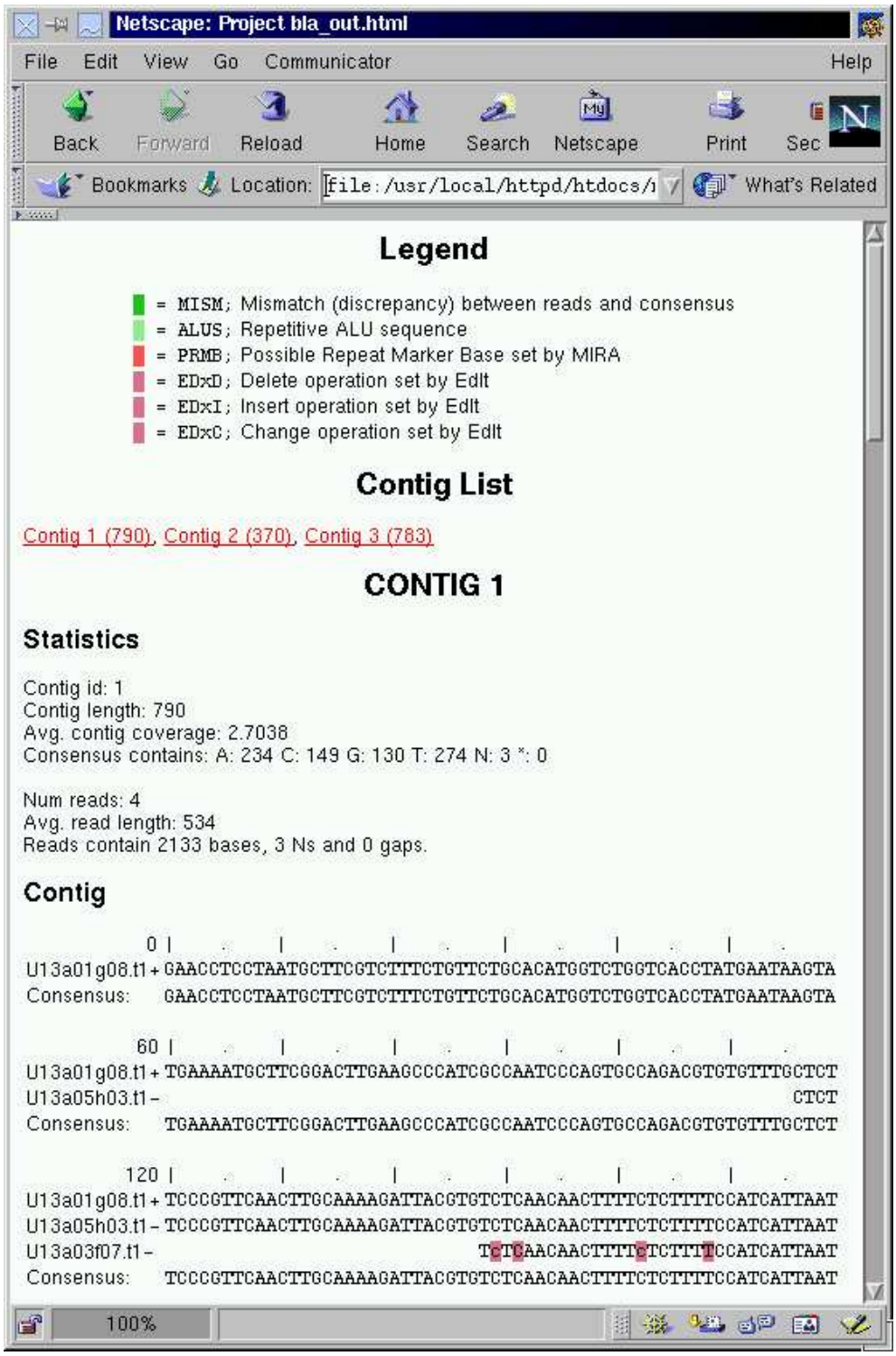

Figure 44: Sample of an assembly HTML output

6 Conclusion and outlook

> *"There is no such thing as absolute truth. That is absolutely true."*
> (Solomon Short)

A new strategy for assembling genomic shotgun and EST sequence data was developed and worked out in this thesis. It combines novel enhancements like repeat detection and on-the-fly automatic editing with strengths of existing assemblers. The strategy also provides the assembler with the ability to use and – more importantly – to acquire by itself additional knowledge present in the assembly data. Furthermore, the knowledge acquisition was combined with the ability to resolve potential conflicts – like long term repeats in genome sequencing projects or different mRNA transcripts in EST projects – during the assembly by falling back to trace signal analysis routines.

Especially the possibility to discriminate alternative solutions – due to previously unknown short and long term repeats – during the assembly process constitutes a systematic improvement in quality of assembly algorithms that produce sequences as accurate as possible.

The main aim set for this thesis was to reduce assembly errors caused by repetitive sequences as well as to increase the reliability of consensus sequences derived from automatically assembled projects. The results presented in chapter 5 demonstrate that the combination of the methods and algorithms devised for this thesis leads to a system that achieves this task. It reliably accomplishes the given task of reconstructing genomic or transcriptomic sequences from DNA or RNA fragments. This is done through the detection, analysis and classification of repetitive elements or single nucleotide polymorphisms which in turn prevents grave misassemblies that occur in other systems.

In most analysed assembly comparisons, the quality of the resulting consensus sequences was improved and the number of errors per kilobase consensus sequence was decreased. The improved strategy described here therefore permits to use resulting sequences almost directly for the design of further investigative studies with high quality and precision requirements like, e.g., the design of oligo probes in clinical micro-array hybridisation screening experiments.

Laboratories using the `mira` assembler routinely report that the most important benefit of using `mira` lies in the fact that – compared to other assemblers – the resulting assembly contains no or very few misassembled reads, which almost eliminates the tedious labour of examining contigs for this kind of error. Instead, simple template direction analysis at the end of contigs suffices to reorder contigs into the probable order as found on the original genome. The capability to recognise and tag previously unknown long term repeats for reassembly has proven to be a valuable asset in the assembly of projects with non-trivial repeats. The possibility to export the assembled genome and EST projects – together with the analysis of possible repeat or SNP sites – to a variety of standard formats, e.g. GAP4 directed assembly, flat text files, phrap ACE or even simple HTML format, opens the door to visual inspection of the results as well as the integration of the tool into more complex and (semi-) automated laboratory workflows.

No project is really perfect and this one is no exception. Usage of `mira` and `miraEST` assembler on a daily basis in production environments shows that some algorithms still need a form of fine tuning. In the future, the primary focus will shift to enable parallel execution in portions of the algorithms to take advantage of multiple processor architectures. Until now, the program uses only one processor on a given machine and this clearly represents a bottleneck when several hundreds of thousands or even millions of sequences are to be assembled. Fortunately, most of the methods presented can be parallelised using a divide-and-conquer strategy so that distributing the workload across different threads, processes and even machines is one of the targets currently pursued. Another point looked into is that usage of the C++ standard template library (STL) currently leads to unexpectedly high memory consumption in some parts of the algorithms. This was traced back to memory pooling strategies of the STL. First experiments with a combination of adapted algorithms together with better behaviour prediction (data not shown) led to a significant reduction of these side-effects.

A Development details

"Design flaws travel in herds." (Solomon Short)

In this section, some insights and numbers are given on how implementational choices were made and the reasons behind them.

A.1 Programming environment

Finding the right approach to resolve the potential conflict between design and programming is one of the hardest tasks to tackle. Several questions need to be addressed when developing algorithms for a sequence assembler:

- execution speed
- memory requirements
- reliability
- portability
- implementation ease

In fact, even nowadays no existing combination of language and compiler provides an optimum coverage of the points mentioned above, they all have positive and negative aspects. The final decision to choose C++ as implementation language was in taken in 1997 because it offers a reasonable mix of the stipulated requirements.

The common base of C and C++ compilers and its backing as a high profile language led very early to good optimising capabilities[1] as well as reliable code with very few errors of existing compilers. The possibility to fall back to plain C

[1] It is widely known that the speed of a program depends mainly on the quality of the algorithms it bases on. However, good compilers can squeeze a considerable amount of execution speed from optimal algorithms by optimising them on machine level, sometimes to a factor of 3 and more.

if needed was also positive aspect although – in the end – this feature was not required.

Portability proved to be somewhat harder to attain. The C++ specifications were finalised and adopted November 14, 1997 by the ISO (International Organisation for Standardisation) as well as several national standards organisations such as ANSI (The American National Standards Institute), BSI (The British Standards Institute), DIN (The German National Standards Organisation) and others. That is well 6 months after the beginning of the project when the first algorithms had to be implemented for feasibility studies. Fortunately, the SGI MIPS compiler and the open source EGCS[2] were stable enough to support the language decision for C++. The EGCS compiler had the undeniable advantage to be available on a multiplicity of different platforms so that portability could be ensured, but tests showed that the SGI compiler undeniably produced code at least twice as fast as the EGCS for the then primary development platform, a SGI Origin 2000 with R10000 processors. This resulted in some makeshifts within the code as the SGI `cc` C++ compiler did not, e.g., support standard C++ string classes even in 1999, but then again, neither did the EGCS. Fortunately, keeping the code compatible to both compilers was not too hard, but portability to other platforms still was not easy: the first compilation on HP machines in 1999 revealed errors in the HP STL implementation that were hard to come by.

Speaking of bugs ... one of the nastiest to encounter is a compiler bug that appears 1) only on specific platform architectures (e.g. SUN) and 2) only when the compiler produces optimised code and 3) is only triggered on rare cases where unexpected results are caught by internal error checking mechanisms which then throw an error ... and in the course of it trigger the bug. When, after quite some investigation, it turns out to be an optimising bug of the compiler used (and not the code one wrote), both relief and anger battle each other. In the end, relief won.

In the beginning the build environment consisted of some simple, hand-coded makefiles that grew larger and larger over time while snatching tricks from makefiles from other authors. But in the end, all these were replaced by the GNU `autoconf` and GNU `automake` systems for somewhat easier cross-platform portability. Which does not mean that the learning curve for these tools was not steep ... quite the contrary is true when it comes to get the build en-

[2] which later on was promoted as official GCC until the new GCC 3 compiler lineage appeared by mid of 2001

vironment running on different platforms, but it all paid off in the end as the package now builds out of the box on quite a number of different UNIX platforms.

Having virtually never worked with file versioning before and having gone through some obligatory code losses while typing silly things like "`rm * .o`"[3], using RCS almost from the very beginning felt like a big step forward in the right direction. But when Thomas Pfisterer joined me at the DKFZ to write his automated editor, we started also to develop some libraries needed by both the assembler and the automated editor. It took some time before the pain of dealing with file locks under RCS became so great that a switch to CVS was envisioned ... and finally made.

Last but not least, even the most careful programming approaches (see the next section) could not prevent some hard to find programming bugs to sneak into the code. As one day, after a week of endless debugging sessions by Thomas and myself, one particularly nasty specimen was identified to be a "simple" pointer arithmetic problem, we caved in and started to use "debugging tools for the sissies": `purify` and, later on, `valgrind`. Those tools have a runtime checking mechanisms for almost every problem that can occur in typical C/C++ programs, ranging from stack errors to buffer overflows to uninitialised memory access. We always thought that we had been careful enough while programming and were a bit shocked by the number of potential problems that were uncovered the first time we ran our program with this tool. Needless to say that since then, regularly using these tools belongs to standard operational procedure. This has – together with using ready made algorithm libraries like the STL, see next section – improved stability of the algorithms developed by an order of magnitude.

A.2 Programming approaches

Obviously, object oriented programming languages offer a wide variety of powerful techniques to support design and programming. The key is always to find designs that fit the problems and use the language constructs that best represent the designs in the code.

The `mira` assembler was implemented using a lot of simple data abstraction – i.e. classes without inheritance – as trying to force everything into a hierar-

[3] mind the blank between the asterisk and .o

chy[4] mostly results in some truly contorted program logic. Simplicity helped to keep the design (relatively) clean, modularise the development process and support the top-down bottom-up engineering model. Generic programming – templates and algorithms parameterised on types – was also used to get efficiency and type safety for containers: the Standard Template Library (STL) as nucleus of the C++ standard library provides facilities such as vectors, maps, and algorithms that work on these containers.

This might seem to contradict the statement on simplicity that was made above, but it really is not. The key is always to come up with a design that fits the problems and also using language constructs that best represent the designs in the code. The possibility to use pre-made algorithms and storage containers that have been validated for years on a wide number of platforms considerably sped up development cycles and improved the stability the algorithms. This also helped to focus design and programming on the immediate assembly problems to be solved, which was complicated enough the way it was, thank you.

A.3 Code statistics

`mira` and associated small helper programs currently have approximately 60 header and code files. The `mira` library, which contains about 95% of the logic, consists of 24 C++ classes with $>$ 25 thousand lines of code. This number is down from approximately $>$ 30 thousand lines of code that were reached during development, mainly due to increased usage of the STL while rewriting large parts of the libraries since mid of 2003. The automatic editor EdIt library from Thomas ended up to have 21 files, 18 classes and $>$ 7 thousand lines of code.

[4] especially into a single-rooted hierarchy

B Manual pages

> *"The manual only makes sense after you learn the program."* (Solomon Short)

The manual pages describe the options available for the `mira` assembler at the time of generation of this thesis and is directly taken from the automated documentation builder for man-pages of the program. The builder is better suited for man- and info pages than for LaTeX– the conversion routines have some minor blemishes (see B.1 Synopsis for example) – but the result is nevertheless readable.

B.1 Synopsis

mira [**-GENERAL:***arguments*] [**-ESTGENERAL:***arguments*] [**-ASSEMBLY:***arguments*] [**-DATAPROCESSING:***arguments*] [**-CLIPPING:***arguments*] [**-ALIGN:***arguments*] [**-CONTIG:***arguments*] [**-EDIT:***arguments*] [**-DIRECTORY:***arguments*] [**-FILE:***arguments*] [**-OUTPUT:***arguments*] [**-params=**<**filename**>] [**-project=**<**name**>] [**-fasta[=**<**filename**>**]**] [**-caf[=**<**filename**>**]**] [**-phd[=**<**filename**>**]**] [**-borg**]

B.2 Description

mira is a DNA sequence data assembly program (pr "fragment assembler") for whole genome and EST projects. *mira* assembles reads / sequences gained by gel or capillary electrophoresis experiments into contiguous sequences (contigs). Input can be in various formats like Staden experiment (EXP), Sanger CAF, FASTA or PHD file.

If present, base qualities in *phred*(1) style and SCF signal electrophoresis trace files are used to adjudicate between or even correct contradictory stretches of bases in reads by either the integrated automatic *EdIt* editor (written by Thomas Pfisterer) or the assembler itself.

mira was conceived especially with the problem of repeats in genomic data and SNPs in EST data in mind. Considerable effort was made to develop a number of strategies – ranging from standard clone-pair size restrictions to discovery and marking of base positions discriminating the different repeats / SNPs – to ensure that repetitive elements are correctly resolved and that misassemblies do not occur.

The resulting assembly can be written in different standard formats like CAF, Staden GAP4 directed assembly, ACE, HTML, FASTA or simple text, which can easily be imported into numerous finishing tools.

The aim of *mira* is to build the best possible assembly by

1. having a full overview on the whole project at any time of the assembly, i.e. knowledge of all possible read-pairs in a project,
2. using high confidence regions (HCRs) of several aligned read-pairs to start contig building at a good anchor point of a contig, extending clipped regions of reads on a 'can justify ' basis.
3. using all available data present at the time of assembly, i.e., instead of relying on sequence and base confidence values only, the assembler will profit from trace files containing electrophoresis signals, tags marking possible special attributes of DNA, information on specific insert sizes of read-pairs etc.
4. having intelligent contig objects accept or refuse reads based on the rate of unexplainable errors introduced into the consensus
5. discovering and analysing of possible repeats differentiated only by single nuceotide polymorphisms. The important bases for discriminating different repetitive elements are tagged.
6. using the possibility given by the integrated automatic editor to correct errors present in contigs (and subsequently) reads by generating and verifying complex error hypotheses through analysis of trace signals in several reads covering the same area of a consensus,
7. iteratively extending reads (and subsequently) contigs based on
   a) additional information gained by overlapping read pairs in contigs and
   b) corrections made by the automated editor.

mira – written by Bastien Chevreux – is part of a bigger project which also contains the automated editor / finisher *EdIt* package – written by Thomas Pfisterer. The strength of *mira* and *EdIt* is the automatic interaction of both

packages which produces assemblies with less work for human finishers to be done. This is the documentation belonging to the (public) release of the *mira* assembler.

MIRA2 has been rewritten in large parts. Compared to the previous versions, it got more accurate, has better support for standard cases, learned to assemble EST sequences and is faster ... much faster. I'd like to thank everybody who reported bugs to me, pointed out problems, sent ideas and suggestions they encountered while using the predecessors. Please continue to do so, the feedback made this second version possible.

B.3 Working modes

mira has two basic working modes: genome or EST. There is a different executable for each mode: *mira*(1) for assembly of genomic data and *miraEST*(1) for assembly of EST data and SNP detection within this assembly.

B.4 Options

Options can be given on the command line or via parameter files. While the format might look a little bit strange, it is borrowed from the SGI C compiler options and allows both compact command lines as well as readable and / or script generated parameter files.

The *mira*(1) command line options accept several arguments and allow a user to specify a setting for each argument. To specify multiple arguments, use colons to separate each argument on the command line. You may either use the long or the short form of each argument, the later being given in brackets.

A typical call of *mira*(1) with the command line could look like this (all in one line):

```
mira -ALIGN:min_relative_score=70
     -CONTIG:use_template_information=yes
            :insertsize_minimum=500
            :insertsize_maximum=2500
```

or in short form

```
mira -AL:mrs=70 -CO:uti=yes:ismin=500:ismax=2500
```

Please note that it is also perfectly legal to write

```
mira -CO:uti=yes -AL:mrs=70 -CO:ismin=500 -CO:ismax=2500
```

(just for those of you who want to use *mira* in scripted environments and / or write scripts for it).

There are example parameter file included in the distribution which shows how to format parameters in files.

B.4.1 -GENERAL (-GE)

General options control the type of assembly to be performed and other switches not belonging anywhere else.

project(pro)=*string* Default is mira. Defines the project name for this assembly. The project name automatically influences the name of input and output files / directories. E.g. in the default setting, the file names for the output of the assembly in FASTA format would be ”mira_out.fasta” and ”mira_out.fasta.qual”. Setting the project name to ”MyProject” would generate ”MyProject_out.fasta” and ”MyProject_out.fasta.qual”. See also **-FILE:** and **-DIRECTORY:** for a list of names that are influenced.

load_job(lj)=*fofnexp, fasta, caf, phd, fofnphd* Default is fofnexp. Defines whether to load and assemble EXP files from a file of filenames (”mira_in.fofn”), load and assemble FASTA sequences (”mira_in.fasta”) and their qualities (”mira_in.fasta.qual”), load and assemble sequences or qualities from a phd file (”mira_in.phd”) or to load a project from a CAF file (”mira_in.caf”) and assemble or eventually reassemble it. Note: fofnphd currently not available.

filecheck_only(fo)=*on|yes|1, off|no|0* Default is no. If set to yes, the project will not be assembled and no assembly output files will be produced. Instead, the project files will only be loaded. This switch is usefull for checking consistency of input files.

merge_xmltraceinfo(mxti)=*on|yes|1, off|no|0* Default is no. Some file formats above (FASTA, PHD or even CAF and EXP) possibly do not contain all the info necessary or useful for each read of an assembly. Should additional information – like clipping positions etc. – be available in a XML trace info file in NCBI format (see **Fileformats**), then set this option to yes and it will be merged to the data loaded. See also **-FILE:** for the name of the

xml file to load.
Please note: quality clippings given here will override quality clippings loaded earlier or performed by *mira*. Minimum clippings will still be made by the program, though.

readnaming_scheme(rns)=*sanger, tigr* Default is sanger. Defines the center naming scheme for read suffixes. Currently, only Sanger Institute and TIGR naming schemes are supported out of the box.
How to choose: please read the documentation available at the different centers or ask your sequence provider. In a nutshell, Sanger scheme is "somename.[pqsfrw][12][bckdeflmnpt][a|b|c|..." (e.g. U13a08f10.p1ca), TIGR scheme is "somenameTF*|TR*|TA*" (e.g. GCPBN02TF or GCPDL68TABRPT103A58B).

external_quality(eq)=*none, SCF* Default is SCF. Defines the source format for reading qualities from external sources. Normally takes effect **only** when these are not present in the format of the load_job project (EXP and FASTA can have them, CAF and PHD must have them).

external_quality_override(eqo)=*on|yes|1, off|no|0* Default is no, only takes effect when load_job is fofnexp. Defines whether or not the qualities from the external source override the possibly loaded qualities from the load_job project. This might be of use in case some postprocessing software fiddles around with the quality values of the input file but one wants to have the original ones.

discard_read_on_eq_error(droeqe)=*on|yes|1, off|no|0* Default is yes. Should there be a major mismatch between the external quality source and the sequence (e.g.: the base sequence read from a SCF file does not match the originally read base sequence), should the read be excluded from assembly or not. If not, it will use the qualities it had before trying to load the external qualities (either default qualities or the ones loaded from the original source).

print_date(ps)=*on|yes|1, off|no|0* Default is yes. Controls whether date and time are printed out during the assembly. Suppressing it is not useful in normal operation, only when debugging or benchmarking.

B.4.2 -ESTGENERAL (-EG)

General options to control EST assemblies (only useful for *miraEST*). switches not belonging anywhere else.

load_straindata(lsd)=*on*|*yes*|*1*, *off*|*no*|*0* Default is no. Straindata is a key value file, one read per line. It is used by the program to differentiate different types of SNPs appearing in organisms and classifying them.

est_startstep(ess)=*1* <= *integer* <= *4* Default is 1. Controls the starting step of the EST assembly.
EST assembly is a three step process, each with different settings to the assembly engine, with the result of each step being saved to disk. If results of previous steps are present in a directory, one can easily "play around" with different setting for subsequent steps by reusing the results of the previous steps and directly starting with step two or three.

B.4.3 -ASSEMBLY (-AS)

General options for controlling the assembly.

minimum_read_length(mrl)=*integer* >= *20* Default is 40. Minimum length that reads must have to be considered for the assembly. Shorter sequences will be filtered out at the beginning of the process and won't be present in the final project.

num_of_loops(nol)=*integer* > *0* Default is 3. Defines how many iterations of the whole assembly process are maximally done. Rule of thumb: quick and dirty assembly (corresponds to the assembly process of *mira* V1.2 and before): use 1. Assembly using read extensions and / or automatic contig editing (**-DP:***ure* and **-ED:***ace*, see below): at least 2. More than 3 will probably not change very much in the assembly. See also **-AS:***pbl* and **-AS:***mr* below for loop parameters that affect the assembly and disentanglement of possible repeats.

prmb_break_loopsmax(pbl)=*integer* > *0* Default is 3. Defines how many times a contig can be rebuilt during a main assembly loop (**-AS:***nol*) if misassemblies due to possible repeats are found.

spoiler_detection(sd)=*on*|*yes*|*1*, *off*|*no*|*0* Default is yes for *mira* and no *miraEST*. A spoiler can be either a chimeric read or it is a read with long parts of unclipped vector sequence still included (that was too long for the **-CL:***pvc* vector leftover clipping routines). A spoiler typically prevents contigs to be joined, MIRA will cut them back so that they represent no more harm to the assembly.
Recommended for assemblies of mid- to high-coverage genomic assemblies, not recommended for assemblies of ESTs as one might loose splice

variants with that,
A minimum number of two assembly loops (**-AS:***nol*) must be run for this option to take effect.

sd_last_loop_only(sdllo)=*on*|*yes*|*1*, *off*|*no*|*0* Default is yes. Defines whether the spoiler detection algorithms are run only for the last loop or for all loops (**-AS:***nol*).
Takes effect only if spoiler detection (**-AS:***sd*) is on.

use_emergency_search_stop(uess)=*on*|*yes*|*1*, *off*|*no*|*0* Default is yes. Another important switch if you plan to assemble non-normalised EST libraries, where some ESTs may reach coverages of several hundreds or thousands of reads. This switch lets MIRA save a lot of computational time when aligning those extremely high coverage areas (but only there), at the expense of some accuracy.

ess_partnerdepth(esspd)=*integer* > *0* Default is 500. Defines the number of potential partners a read must have for MIRA switching into emergency search stop mode for that read.

use_max_contig_buildtime(umcbt)=*on*|*yes*|*1*,*off*|*no*|*0* Default is no. Defines whether there is an upper limit of time to be used to build one contig. Set this to yes in EST assemblies where you think that extremely high coverages occur. Less useful for assembly of genomic sequences.

buildtime_in_seconds(bts)=*integer* > *0* Default is 10000. Depending on **-AS:***umcbt* above, this number defines the time in seconds alloted to building one contig.

B.4.4 -DATAPROCESSING (-DP)

Options for controlling some data processing during the assembly.

use_read_extension(ure)=*on*|*yes*|*1*, *off*|*no*|*0* Default is yes. *mira* expects the sequences it is given to be quality clipped. During the assembly though, it will try to extend reads into the clipped region by analysing alignments between reads that were found to be valid.

tag_polyat_at_ends(tpae)=*on*|*yes*|*1*, *off*|*no*|*0* Default is no. This option is useful in EST assembly. Poly-AT stretches at end of reads that were not correctly masked or clipped in preprocessing steps from external programs get tagged here. The assembler will not use these stretches for critical

operations. Additionally, the tags do provide a good visual anchor when looking at the assembly with different programs.

polybase_window_len(pbwl)=*integer* > *0* Default is 7. Defines the window length within which all bases (except the maximum number of errors allowed, see below) must be either A or T to be considered a polybase stretch.

polybase_window_maxerrors(pbwme)=*integer* > *0* Default is 2. Defines the maximum number of errors allowed in a given window length (see above) so that a stretch is considered to be a polybase stretch. The distribution of these errors is not important.

polybase_window_gracedistance(pbwgd)=*integer* > *0* Default is 9. Defines the number of bases from the end of a sequence (if masked: from the end of the masked area) within which a polybase stretch is looked for without finding one.

B.4.5 -CLIPPING (-CL)

Controls for clipping options: when and how sequences should be clipped.

possible_vector_clip(pvc)=*on*|*yes*|*1*, *off*|*no*|*0* Default is yes. *mira* will try to identify possible sequencing vector relicts present at the start of a sequence and clip them away. These relicts are usually a few bases long and were not correctly removed from the sequence in data preprocessing steps of external programs.

You might want to turn off this option if you know (or think) that your data contains a lot of repeats and the option below to fine tune the clipping behaviour does not give the expected results.

pvc_maxlenallowed(pvcmla)=*integer* >= *0* Default is 12. The clipping of possible vector relicts option works quite well. Unfortunately, especially the bounds of repeats or differences in EST splice variants sometimes show the same alignment behaviour than possible sequencing vector relicts and could therefore also be clipped.

To refrain the vector clipping from mistakenly clip repetitive regions or EST splice variants, this option puts an upper bound to the number of bases a potential clip is allowed to have. If the number of bases is below or equal to this threshold, the bases are clipped. If the number of bases exceeds the threshold, the clip is **NOT** performed.

Setting the value to 0 turns off the threshold, i.e., clips are then always performed if a potential vector was found.

quality_clip(qc)=*on*|*yes*|*1, off*|*no*|*0* Default is no, but is set automatically to yes when using the quickmode switches *-fasta* or *-phd* (can be turned off again by subsequent options afterwards). This will let *mira* perform its own quality clipping before sequences are entered into the assembly. The clip function performed is a sequence end window quality clip with backiteration to get a maximum number of bases as useful sequence. Note that the bases clipped away here can still be used afterwards if there is enough evidence supporting their correctness when the option **-AS:***ure* is turned on.

qc_minimum_quality(qcmq)=*integer* >= *15 and* <= *35* Default is 20. This is the minimum quality bases in a window require to be accepted. Please be cautious not to take too extreme values here, because then the clipping will be too lax or too harsh. Values below 15 and higher than 30-35 are not recommended.

qc_window_length(qcwl)=*integer* >= *10* Default is 30. This is the length of a window in bases for the quality clip.

maskedbases_clip(mbc)=*on*|*yes*|*1, off*|*no*|*0* Default is no, but is set automatically to yes when using the quickmode switches *-fasta* or *-phd* (can be turned off again by subsequent options afterwards). This will let *mira* perform a 'clipping' of bases that were masked out (replaced with the character X). It is generally not a good idea to use mask bases to remove unwanted portions of a sequence, the EXP file format has excellent possibilities to circumvent this. But because a lot of preprocessing software are built around *cross_match*, *scylla*- and *phrap*-style of base masking, the need arised for *mira* to be able to handle this, too. *mira* will look at the start and end of each sequence to see whether there are masked bases that should be 'clipped'.

mbc_gap_size(mbcgs)=*integer* >= *0* Default is 15. While performing the clip of masked bases, *mira* will look if it can merge larger chunks of masked bases that are a maximum of **-CL:***mbcgs* apart.

mbc_max_front_gap(mbcmfg)=*integer* >= *0* Default is 30. While performing the clip of masked bases at the start of a sequence, *mira* will allow up to this number of unmasked bases in front of a masked stretch.

mbc_max_end_gap(mbcmeg)=*integer* >= *0* Default is 60. While performing the clip of masked bases at the end of a sequence, *mira* will allow up to this number of unmasked bases behind a masked stretch.

B.4.6 -ALIGN (-AL)

The align options control the behaviour of the Smith-Waterman alignment routines. Only read pairs which are confirmed here may be included into contigs.

bandwidth_in_percent(bip)=*integer*>*0 and* <=*100* Default is 15. The banded Smith-Waterman aligment uses this percentage number to compute the bandwitdh it has to use when computing the alignment matrix. E.g., expected overlap is 150 bases, bip=10 -> the banded SW will compute a band of 15 bases to each side of the expected alignment diagonal, thus allowing up to 15 unbalanced inserts / deletes in the alignment. INCREASING AND DECREASING THIS NUMBER: *increase*: will find more non-optimal alignments, but will also increase SW runtime between linear and ^2. *decrease*: the other way round, might miss a few bad alignments but gaining speed.

bandwidth_max(bmax)=*integer*>*0* Default is 50. Maximum bandwidth in bases to each side.

bandwidth_max(bmin)=*integer*>*0* Default is 10. Minimum bandwidth in bases to each side.

min_overlap(mo)=*integer*>*0* Default is 15. Minimim number of overlapping bases needed in an alignment of two sequences to be accepted.

min_score(ms)=*integer*>*0* Default is 15. Describes the minimum score of an overlap to be taken into account for assembly. *mira* uses a default scoring scheme for SW align: each match counts 1, a match with an N counts 0, each mismatch with a non-N base -1 and each gap -2. Take a bigger score to weed out a number of chance matches, a lower score to perhaps find the single (short) alignment that might join two contigs together (at the expense of computing time and memory).

min_relative_score(mrs)=*integer*>*0 and* <=*100* Default is 65. Describes the min % of matching between two reads to be considered for assembly. Increasing this number will save memory, but one might loose possible alignments. I propose a maximum of 80 here. Decreasing below 55% will make memory and time consumption probably explode.

extra_gap_penalty(egp)=*on*|*yes*|*1, off*|*no*|*0* Default is yes. Defines whether or not to increase penalties applied to alignments containing long gaps. Setting this to 'yes' might help in projects with frequent repeats. On the other hand, it is definitively disturbing when assembling very long reads

containing multiple long indels in the called base sequence ... although this should not happen in the first place and is a sure sign for problems lying ahead.

gap_penalty_level(gpl)=*low|0, medium|1, high|2, est_splitsplices|10* Default is low. Has no effect if extra_gap_penalty is off. Defines an extra penalty applied to 'long' gaps. There are these are predefined levels: low - use this if you expect your base caller frequently misses 2 or more bases. medium - use this if your base caller is expected to frequently miss 1 to 2 bases. high - use this if your base caller does not frequently miss more than 1 base.

For some stages of the EST assembly process, a special value *est_splitsplices* is used.

extra_mismatch_penalty(emp)=*on|yes|1, off|no|0* Default is yes. Defines whether or not to reject alignments containing long stretches of mismatches. Setting this to 'yes' might help in projects with frequent repeats. Also very useful in EST projects to prevent the assembly of different splice variants.

emp_windowlen(empwl)=*integer>0* Default is 30. Has no effect if extra_mismatch_penalty is off. Defines the length of the window of bases that is looked for to contain too many mismatches.

emp_maxmitches(empmm)=*integer>0* Default is 15. Has no effect if extra_mismatch_penalty is off. Defines how many errors must occur in the window before an alignment is rejected.

B.4.7 -CONTIG (-CO)

The contig options control the behaviour of the contig objects.

analysis(an)=*none, text, signal* Default is signal. When adding reads to a contig, dangerous regions can get an extra integrity check.

none = no extra check.

text = check is only text-based.

signal = check is signal based, if the SCF trace is not available, fallback is 'text'.

For the time being, only regions tagged as ALUS or REPT in the experiment file are considered dangerous.

reject_on_drop_in_relscore(rodirs)=*integer*>*0 and* <=*100* Default is 7. When adding reads to a contig, reject the reads if the drop in the quality of the consensus is > the given value in %. Lower values mean stricter checking. This value is doubled should a read be entered that has a template partner (a read pair) at the right distance.

danger_max_error_rate(dmer)=*integer*>=*1 and* <=*100* Default is 1. When adding reads to a contig, reject the reads if the error in zones known as dangerous exceeds the given value in %. Lower values mean stricter checking in these danger zones.
For the time being, only regions tagged as ALUS or REPT in the experiment file are considered dangerous.

mark_repeats(mr)=*on*|*yes*|*1, off*|*no*|*0* Default is yes. One of the most important switches in MIRA: if set to yes, MIRA will try to resolve misassemblies due to repeats by identifying single base stretch differences and tag those critical bases as PRMB (Probable Repeat Marker Base). This switch is also needed when MIRA is run in EST mode to identify possible inter-, intra- and intra-and-interorganism SNPs.

min_prmb_coverage(mpc)=*integer* >= *2* Default is 3. Only takes effect when **-CO:***mr* (see above) is set to yes. This defines the minimum coverage needed for the PRMB (Probable Repeat Marker Bases) detection routines. Setting this to a low number increases sensitivity, but might produce a few false positives, resulting in reads being thrown out of contigs because of falsely identified possible repeat markers.

num_prmb_coverage(npz)=*integer* >= *2* Default is 2. Only takes effect when **-AS:***mr* is set to yes. This defines the minimum number of possible PRMB zones defined within a certain distance before these are really treated as PRMB.

min_groupqual_for_prmb_tagging(mgqpt)=*integer* >= *25* Default is 30. Takes only effect when **-AS:***mr* is set to yes. This defines the minimum quality of a group of bases to be taken into account as potential repeat marker. The lower the number, the more sensitive you get, but lowering below 25 is not recomended as a lot of wrongly called bases can have a quality approaching this value and you'd end up with a lot of false positives. The higher the overal coverage of your project, the better, and the higher you can set this number. A value of 35 will probably remove all false positives, a value of 40 will probably never show false positives.

min_groupqual_for_wrmbprmb_change(mgqwpc)=*integer* $>=$ ***30*** Default is 45. Takes only effect when **-AS:***mr* is set to yes. Probable repeat marker bases without supporting bases around them are tagged as WRMB (Weak Repeat Marker Base). These can be transformed into PRMBs if the quality value of the base group matches or exceeds the number given here.

endread_mark_exclusion_area(emea)=*integer* $>=$ ***0*** Default is 15. Takes only effect when **-AS:***mr* is set to yes. Using the end of sequences is always a bit risky, as wrongly called bases tend to crowd there or some sequencing vector relicts hang around. It is even more risky to use these stretches for detecting possible repeats, so one can define an exclusion area where the bases are not used when determining whether a mismatch is due to repeats or not.

use_template_information(uti)=*on*|*yes*|*1*, *off*|*no*|*0* Default is Yes. Two reads sequenced from the same clone template form a read pair with a known minimum and maximum distance. This feature will definitively help for contigs containing lots of repeats. Set this to 'yes' if your data contains information on insert sizes.

Information on insert sizes can be given via the SI tag in EXP files (for each read pair individually), or for the whole project using **-CO:***ismin* and **-CO:***ismax* (see below).

insertsizeminimum(ismin)=*integer* $>=$ ***0*** Default is 500. The minimum distance that read pairs may be apart. There is an additional error margin of 10% subtracted from this value during internal computations.

insertsizemaximum(ismax)=*integer* $>=$ ***0*** Default is 5000. The maximum distance that read pairs may be apart. There is an additional error margin of 10% added to this value during internal computations.

B.4.8 -EDIT (-ED)

General options for controlling the integrated automatic editor.

automatic_contig_editing(ace)=*on*|*yes*|*1*, *off*|*no*|*0* Default is yes. Once contigs have been build, *mira* will call a built-in version of the automatic contig editor *EdIt*. *EdIt* will try to resolve discrepancies in the contig by performing trace analysis and correct even hard to resolve errors. This option is always useful, but especially in conjunction with **-GE:***nol* and **-GE:***ure* (see above).

strict_editing_mode(sem)=*on|yes|1, off|no|0* Default is yes. If set to yes, the automatic editor will not take error hypotheses with a low probability into account, even if all the requirements to make an edit are fulfilled.

confirmation_threshold(ct)=*integer, 0 < x <= 100* Default is 50. The higher this value, the more strict the automatic editor will apply its internal rule set. Going below 40 is not recommended.

B.4.9 -DIRECTORY (-DIR, -DI)

General options for controlling where to find or where to write data.

gap4da=<*directoryname*> Default is gap4da. Defines the extension of the directory where *mira* will write the result of an assembly ready to import into the Staden package (GAP4) in Direct Assembly format. The name of the directory will then be <projectname>_.<extension>

exp=<*directoryname*> Default is .. Defines the directory where *mira* should search for experiment files (EXP).

scf=<*directoryname*> Default is .. Defines the directory where *mira* should search for SCF files.

log=<*directoryname*> Default is miralog. Defines the directory where *mira* will write some log files to. Note that the name of the actual project will be prepended.

B.4.10 -FILE (-FI)

The file options allows you to define your own input and output files.

fofnexpin(fei)=*string* Default is <projectname>_in.fofn. Defines the file of filenames where the names of the EXP files of a project are located.

fofnphdin(fpi)=*string* Default is <projectname>_in.fofn. Defines the file of filenames where the names of the PHD files of a project are located. Note: this is currently not available.

phdin(pi)=*string* Default is <projectname>_in.phd. Defines the file of where all the sequences of a project are in PHD format.

fastain(fai)=*string* Default is mira_in.fasta. Defines the fasta file to load sequences of a project from.

fastaqualin(fqi)=*string* Default is mira_in.fasta.qual. Defines the fasta file to load base qualities of a project from. Although the order of reads in the quality file must not be the same as in the fasta or fofn, it is strongly recommended that they are (saves considerable time when loading big projects).

cafin(ci)=*string* Default is mira_in.caf. Defines the file to load a CAF project from. Filename must end with '.caf'.

straindatain(sdi)=*string* Default is mira_straindata_in.txt. Defines the file to load straindata from. Only used in EST projects (*miraEST*).

xmltraceinfoin(xtii)=*string* Default is mira_xmltraceinfo_in.xml. Defines the file to load a trace info file in XML format from. This can be used both when merging XML data to loaded files or when loading a project from an XML trace info file.

cafout(co)=*string* Default is mira_out.caf. Defines the file in CAF format to save an assembled project to. Filename must end with '.caf'.

B.4.11 -OUTPUT (-OUT)

Options for controlling which results to write to which type of files.
There are 3 types of results: result, temporary results and extra temporary results. One probably needs only the results. Temporary and extra temporary results are given as convenience for trying to find out why *mira* set some PRMBs or disassembled some contigs.
Output can be generated in these formats: CAF, Gap4 Directed Assembly, FASTA, ACE, HTML and simple text. Please note that the ACE output is experimental as I don't have the necessary programs (phrap, consed) to verify the output. (Anyone who wants to sponsor them? :-)
Naming conventions of the files follow the rules described in section **Input / Output**, subsection **Filenames**.

output_result_caf(orc)=*on|yes|1, off|no|0* Default is yes.

output_result_gap4da(org)=*on|yes|1, off|no|0* Default is yes.

output_result_fasta(orf)=*on|yes|1, off|no|0* Default is yes.

output_result_ace(ora)=*on|yes|1, off|no|0* Default is yes.

output_result_txt(ort)=*on|yes|1, off|no|0* Default is yes.

output_result_html(orh)=*on|yes|1, off|no|0* Default is yes.

output_tmpresult_caf(otc)=*on*|*yes*|*1*, *off*|*no*|*0* Default is no.

output_tmpresult_gap4da(otg)=*on*|*yes*|*1*, *off*|*no*|*0* Default is no.

output_tmpresult_fasta(otf)=*on*|*yes*|*1*, *off*|*no*|*0* Default is no.

output_tmpresult_ace(ota)=*on*|*yes*|*1*, *off*|*no*|*0* Default is no.

output_tmpresult_txt(ott)=*on*|*yes*|*1*, *off*|*no*|*0* Default is no.

output_tmpresult_html(oth)=*on*|*yes*|*1*, *off*|*no*|*0* Default is no.

output_exttmpresult_caf(oetc)=*on*|*yes*|*1*, *off*|*no*|*0* Default is no.

output_exttmpresult_gap4da(oetg)=*on*|*yes*|*1*, *off*|*no*|*0* Default is no.

output_exttmpresult_fasta(oetf)=*on*|*yes*|*1*, *off*|*no*|*0* Default is no.

output_exttmpresult_ace(oeta)=*on*|*yes*|*1*, *off*|*no*|*0* Default is no.

output_exttmpresult_txt(oett)=*on*|*yes*|*1*, *off*|*no*|*0* Default is no.

output_exttmpresult_html(oeth)=*on*|*yes*|*1*, *off*|*no*|*0* Default is no.

B.4.12 Quick mode switches

(-fasta | -fasta=<filename>) Sets some parameters suited for loading FASTA files. The version with =<filename> will also set the input file to the given filename. A double dash (–fasta) may also be used instead of a single one.

(-phd | -phd=<filename>) Sets some parameters suited for loading PHD files. The version with =<filename> will also set the input file to the given filename. A double dash (–phd) may also be used instead of a single one.

(-caf | -caf=<filename>) Sets some parameters suited for loading CAF files. The version with =<filename> will also set the input file to the given filename. A double dash (–caf) may also be used instead of a single one.

(-project=<name>) Same as **-GE:***project*. A double dash (–project) may also be used instead of a single one, also 'projectname' instead of 'project'.

B.4.13 Other switches

(-params=<filename>) Loads parameters from the filename given. Allows a maximum of 10 levels of recursion, i.e. a -params option appearing within a file that loads other parameter files (though I cannot think of useful applications with more than 3). A double dash (–params) may also be used instead of a single one, also 'parameterfile' instead of 'params'.

(-borg) Sets a bunch of parameters to have *mira* try to assemble as many reads as possible. Will slow down the assembly process.
"We are MIRA of borg. You will be assembled, resistance is futile!"

B.5 Input / Output

B.5.1 Filenames

<**projectname**>**_in.fofn** File of filenames containing the names of the experiment or phd files to assemble when the **-GE:***lj=FOFNEXP* or **-GE:***lj=FOFNPHD* option is used. One filename per line, blank lines accepted, lines starting with a hash (#) treated as comment lines, nothing else. Use **-FI:***fofnin(fei)* or **-FI:***fofnin(fpi)* to change the default name.

<**projectname**>**_in.phd** File containing the sequences (and their qualities) to assemble in PHD format.

<**projectname**>**_in.fasta** File containing sequences and ...

<**projectname**>**_in.fasta.qual** ... file containing quality values of sequences for the assembly in FASTA format.

<**projectname**>**_in.caf** File containing the sequences (and their qualities) to assemble in CAF format. This format also may contain the result of an assembly (the contig consensi).

<**projectname**>**_out.**<**type**> Assembled project written in type = (*gap4da* / *caf* / *ace* / *fasta* / *html* / *text*) format by *mira*, final result. Type *gap4da* is a directory containing experiment files and a file of filenames (called 'fofn'), all other types are files. *gap4da*, *caf*, *ace* contain the complete assembly information suitable for import into different postprocessing tools (*gap4*, consed and others). *fasta* contains the contig consensi (and .fasta.qual the consensus qualities). *html* and *text* contain visual representations of the assembly suited for viewing in browsers or as simple text file.

<**projectname**>**_info_contigreadlist.txt** This file contains information which reads have been assembled into which contigs (or singlets).

<**projectname**>**_info_contigstats.txt** This file contains statistics about the contigs themselves, their length, average consensus quality, number of reads, maximum and average coverage, average read length, number of A, C, G, T, N, X and gaps in consensus.

<**projectname**>**_info_consensustaglist.txt** This file contains information about the tags (and their position) that are present in the consensus of a contig.

<**projectname**>**_info_readstooshort** A list containing the names of those reads that have been sorted out of the assembly before any processing started only due to the fact that they were too short.

<**projectname**>**_info_readtaglist.txt** This file contains information about the tags and their position that are present in each read. The read positions are given relative to the forward direction of the sequence (i.e. as it was entered into the the assembly).

<**projectname**>**_error_reads_invalid** A list of sequences that have been found to be invalid due to various reasons (given in the output of the assembler).

B.5.2 Fileformats

EXP Standard experiment files used in genome sequencing. Correct EXP files are expected. Especially the ID record (containing the id of the reading) and the LN record (containing the name of the corresponding trace file) should be correctly set. See `http://www.sourceforge.net/projects/staden/` for links to online format description.

SCF The Staden trace file format that has established itself as compact standard replacement for the much bigger ABI files. See `http://www.sourceforge.net/projects/staden/` for links to online format description.
The SCF files should be V2-8bit, V2-16bit, V3-8bit or V3-16bit and can be packed with compress or gzip.

CAF Common Assembly Format (CAF) developed by the Sanger Centre. `http://www.sanger.ac.uk/Software/formats/CAF/` provides a description of the format and some software documentation. They had a link to downloadable binaries of the important caf2gap and gap2caf tools, but it seems they now just offer the source.

ACE The assembly file format used mainly by *phrap* and *consed*. Support for .ace output is currently only in test status in *mira* as documentation on that format is ... sparse and I currently don' have access to consed to

verify my assumptions.
Using *consed*, you will need to load projects with -nophd to view them. Tags are rudimentary supported, this should improve over time. You might also want to try *clview* (`http://www.tigr.org/tdb/tgi/software/`) from TIGR to look at .ace files.

`HTML` Hypertext Markup Language. Projects written in HTML format can be viewed directly with any table capable browser. Display is even better if the browser knows style sheets (CSS).

`FASTA` A simple format for sequence data, see `http://www.ncbi.nlm.nih.gov/BLAST/fasta.html`. An often used extension of that format is used to also store quality values in a similar fashion, these files have a .fasta.qual ending.

`PHD` This file type originates from the *phred* basecaller and contains basically – along with some other status information – the base sequence, the base quality values and the peak indices, but not the sequence traces itself.

`traceinfo.XML` XML based file with information relating to traces. Used at the NCBI and ENSEMBL trace archive to store additional information (like clippings, insert sizes etc.) for projects. See `http://www.ncbi.nlm.nih.gov/Traces/trace.cgi?cmd=show&f=rfc&m=main&s=rfc` for a description of the fields used.

`CAML` Common Assembly Markup Language, an XML based assembly description language. This file type originates from the *PGA* (Paracel Genome Assembler) and *PTA* (Paracel Transcript Assembler) programs. *mira* is currently not able to read or write this file format.

B.5.3 STDOUT

The actual stage of the assembly is written to STDOUT, giving status messages on what *mira* is actually doing.

Some debugging information might also be written to STDOUT if *mira* generates error messages.

B.5.4 STDERR

During the assembly, *mira* might dump some messages to standard error. Basically, three error classes exist:

WARNING Messages in this error class do not stop the assembly but are meant as an information to the user. In some rare cases these errors are due to (an always possible) error in the I/O routines of *mira*, but nowadays they are mostly due to unexpected (read: wrong) input data and can be traced back to errors in the preprocessing stages. If these errors arise, you definitively **DO** want to check how and why these errors came into those files in the first place.
Frequent cause for warnings include missing SCF files, SCF files containing known quirks, EXP files containing known quirks etc.

FATAL Messages in this error class actually stop the assembly. These are mostly due to missing files that *mira* needs or to very garbled (wrong) input data. Frequent causes include naming an experiment file in the 'file of filenames' that could not be found on the disk, same experiment file twice in the project, suspected errors in the EXP files, etc.

INTERNAL These are true programming errors that were caught by internal checks. Should this happen, please mail the output of STDOUT and STDERR to the authors.

B.5.5 Logfiles

During assembly, *mira* will write a whole lot of logfiles which all will be placed into a subdirectory named "<projectname>_miralog"

Please do not delete the project where errors happened. I will get in touch with you for additional information that might possibly be present in temporary files and the logfile. They are not deleted in cases like this.

B.6 Tags used in the assembly by MIRA and EdIt

mira uses and sets a couple of tags during the assembly process. That is, if information is known before the assembly, it can be stored in tags (in the EXP and CAF formats) and will be used in the assembly.

B.6.1 Tags read (and used)

– **ALUS, REPT:** Sequence stretches tagged as ALUS (ALU Sequence) or REPT (general repetitive sequence) will be handled with extreme care

during the assembly process. The allowed error rate after automatic contig editing within these stretches is normally far below the general allowed error rate, leading to much higher stringency during the assembly process and subsequently to a better repeat resolving in many cases.

- **PRMB, WRMB: P**robable **R**epeat **M**arker **B**ase and **W**eak **R**epeat **M**arker **B**ase. These tags are used on an individual per base basis for each read. They denote bases that have been identified as crucial for resolving repeats, often denoting a single SNP within several hundreds or thousands of bases. While a PRMB is quite probable, the WRMB really is either weak (there wasn't enough comforting information in the vicinity to be really sure) or not supported by other probable repeat markes in the vicinity.
- **other:** Other tags in reads will be read and passed through the assembly without being changed and they currently do not influence the assembly process.

B.6.2 Tags set (and used)

- **PRMB, WRMB:** See "Tags read (and used)" above for a description what these tags mean.
  mira will automatically set these tags when it encounters repeats and will tag exactly those bases that can be used to discern the differences.
  Seeing such a tag in the consensus means that *mira* was not able to finish the disentanglement of that special repeat stretch or that it found a new one in one of the last loops without having the opportunity to resolve the problem.
- **ED_C, ED_I, ED_D:** EDit Change, EDit Insertion, EDit Deletion. These tags are set by the integrated automatic editor EdIt and show which edit actions have been performed.
- **PAOS, PROS, PIOS: P**ossible intr**A** **O**rganism **S**NP, **P**ossible inter**R** **O**rganism **S**NP, **P**ossible **I**nter- and intra **O**rganism **S**NP. These tags are set by *mira* when it runs in EST assembly-SNP discovering mode and denotes SNPs as they occur within an organism (PAOS), between two or more organisms (PROS) or within and between organisms (PIOS).

B.7 Requirements

To use *mira* itself, one doesn't need very much:

- the sequences in EXP, CAF, PHD, or FASTA format (ideally preprocessed)
- *mira*, some memory and disk space

Viewing the results or preprocessing the sequences:

- A browser who understands tables is needed to view the HTML output. A browser knowing style sheets (CSS) is recommended, as different tags will be highlighted. Konqueror, Opera, Mozilla, Netscape and Internet Explorer all do fine, lynx is not really ... optimal.
- A text viewer for the different textual output files.
- You'll want GAP4 (generally speaking: the Staden package) to preprocess the sequences, visualise and eventually rework the results when using gap4da output. The Staden package comes with a fully featured sequence preparing and annotating engine (*pregap4*) that is very useful to preprocess your data (conversion between file types, quality clipping, tagging etc.).
  See `http://www.sourceforge.net/projects/staden/` for further information.
- *phred*(basecaller) - *cross_match*(sequence comparison and filtering) - *phrap*(assembler) - *consed*(assembly viewer and editor). This is another package that can be used for this type of job, but requires more programming work. The fact that sequence stretches are masked out (overwritten with the character X) if they shouldn't be used in an assembly doesn't really help and is considered harmful (but it works).
  See `http://www.phrap.org/` for further information.
- Viewing .ace output without consed can be done with *clview* (`http://www.tigr.org/tdb/tgi/software/`) from TIGR.
- Paracel (`http://www.paracel.com`) also offers useful tools for working with sequences: TraceTuner (modified and improved basecaller based on *phred*), PFP (sequence filtering package) and PGA / PTA (genome and EST assembler based on the CAP4 assembler) and the corresponding viewers. The filtering package fortunately knows two modes of operation (masking bases and tagging them) and therefore integrates nicely into existing

workflows based on the masking mode while offering advanced features like tagging (through XML files) when used in that mode.

As always, most of the time a combination of several different packages is possible.

B.8 Speed and memory considerations

Memory consumption has increased since the last public V1.4.x versions, which is mainly due to temporary memory needed for an extremely fast "all against all" read comparison.

- As a rough rule of thumb, you'll have to provide about 20k to 25k of RAM fore each sequence (with approx 1000 bases).
- About 2/5 of this memory consumption is only needed during the fast read comparison part (SKIM) and is freed as soon as that part finishes.
- A certain amount of temporary RAM is used for Smith-Waterman matches. This amount depends on the size of your reads and is a square to their length. Maximum usage examples: 4 MB for reads of length 1000, 16 MB for length 2000, 100 MB for length 5000

The times given here are only approximate and were gathered on my small home development box (Athlon 900) using a single processor and some debug code compiled in, somewhat slowing down the whole process. Here are the times for a non-normalized (thus highly repetitive) EST project, 7651 reads with a mean length of 450 used bases,

- The fast filtering algorithm performs about 1 million sequence comparisons per second (57 seconds).
- Banded Smith-Waterman performs around 500 sequence alignments (with a 15% band to each side, which is quite generous), 4:26 for about 135000 alignment checks.
- Assembling the whole project in one main assembly loop, including resolving very high coverage contigs (>500 sequences) in multiple passes and splitting them into different SNP and splice variants took about 45 minutes.

Assembling for example a small genomic project with 720 reads forming a 35k bases contig in two main loop iterations, resolving minor repeat misassemblies, full read extension and automatic contig editing takes 5:30 minutes.

B.9 Usage

Mira can be used in two different ways: building assemblies from scratch or performing reassembly on existing projects.

B.9.1 Assembly from scratch with GAP4 and EXP files

A simple *GAP4* project will do nicely. Please take care of the following: You need already preprocessed experiment / fasta / phd files, i.e., at least the sequencing vector should have been tagged (in EXP files) or masked out (FASTA or PHD files). It would be nice if some kind of not too lazy quality clipping had also been done for the EXP files, *pregap4* should do this for you.

Step 1 Create a file of filenames (named ”mira_in.fofn”) for the project you wish to assemble. The file of filenames should contain the newline separated names of the EXP-files and nothing else.

Step 2 Execute the *mira* assembly, eventually using command line options or output redirection:
/path/to/the/mira/package/mira
or simply
mira
if *Mira* is in a directory which is in your PATH. The result of the assembly will now be in files named ’mira_out.caf’, ’mira_out.html’ etc. or in gap4 direct assembly format in the ’mira_out.gap4da’ directory.

Step 3a Have a quick look at how the project looks like by loading the file ’mira_out.html’ into your favourite web browser or ...

Step 3b ... change to the gap4da directory:
cd mira_out.gap4da
start gap4:
gap4
choose the menu ’File->New’ and enter a name for your new database (like ’test’). Then choose the menu ’Assembly->Directed assembly’. Enter the text ’fofn’ in the entry labeled ”Input readings from List or file name” and enter the text ’failures’ into the entry labeled ”Save failures to List or filen name”. Press ”OK”.
That’s it.

Out-of-the box example

Mira comes with a really small toy project to test usability on a given system.

Go to the example directory and follow the instructions given in the section for own projects above, but start with step 2. Eventually, you might want to start *mira* while redirecting the output to a file for later analysis.

B.10 Known Problems / Bugs

File Input / Output:

1. *mira* can only read unedited EXP files.
2. due to the fact that the Sanger Centre currently doesn't offer caf2gap and gap2caf precompiled (and compiling them is a non-trivial task), the option to reassemble GAP4 projects after working on them is currently not possible.
3. ACE output still is experimental and might contain errors, especially the tag format is shaky.
4. A minor problem might arise in conjunction with the CR, CL and CS tags in EXP files. CR and CL define right and left clips, CS a range. I have seen preprocessors mixing that up. So, don't worry if suddenly some of your reads are tagged as cloning vector. It's probably the preprocessors fault (but signal this to me anyway).

Assembly process:

1. The routines for determining *Probable Repeat Marker Bases* (PRMB) are sometimes too sensitive, which sometimes leads to excessive base tagging and preventing right assemblies in subsequent assembly processes. The parameters you should look at for this problem are **-CO:***mpc:npz:mgqpt:mgqwpc*. Also look at **-CL:***pvc* and **-CO:***emea* if you have a lot of sequencing vector relicts at the end of the sequences.

B.11 Caveats

– *mira* cannot work (yet) with EXP files resulting from GAP4 that already have been edited. If you want to reassemble an edited GAP4 project, convert it to CAF format and use the -GE=lj=CAF option.

– As also explained earlier, *mira* relies on sequencing vector being recognised in preprocessing steps by other programs. Sometimes, when a whole

stretch of bases is not correctly marked as sequencing vector, the reads might not be aligned into a contig although they might otherwise match quite perfectly. You can use **-CL:***pvc* and **-CO:***emea* to address this problem. Also having the assembler work with less strict parameters may help out of this.

– *mira* has been developed to assemble shotgun sequencing or EST sequencing data. There are no explicit limitations concerning length or number of sequences. However, there are a few implicit assumptions that were made while writing portions of the code:

1. Sequence data produced by electrophoresis rarely surpasses 1000 usable bases and I never heard of more than 1100. The fast filtering SKIM relies on the fact that sequences will never exceed 10000 bases in length.
2. The next problem that might arise with 'unnatural' long sequence reads will be my implementation of the Smith-Waterman alignment routines. I use a banded version with linear running time (linear to the bandwidth) but quadratic space usage. So, comparing two 'reads' of length 5000 will result in memory usage of 100MB. I know that this could be considered as a flaw. On the other hand - unless someone comes up with electrophoresis producing reads with more than 2000 usable bases - I see no real need to change this as long as there are more important things on the TODO list. Of course, if anyone is willing to contribute a fast banded SW alignment routine which runs in linear time and space, just feel free to contact the authors.
3. Current data structures allow for a worst case read coverage of maximally 16384 reads on top of the other. Anyone who wants to comment on that?

Though the pathfinder and alignment engine has almost completely changed from the V1.x versions, I am quite confident that the underlying strategy for the assembly problem contains no vital flaw. If you have suggestions or spot problems, feel free to mail.

Note: Versions with uneven minor versions (e.g. 1.1.x, 1.3.x, ..., 2.1.x, ... etc.) are test versions which might be unstable in parts (although I don't think so). But to catch possible bugs, test versions of *mira* are distributed with tons of internal checks compiled into the code, making it somewhere between 10% and 50% slower than it could be.

B.12 TODOs

These are some of the topics on my TODO list for the next revisions to come:

1. Hidden data (parts of a read that have been clipped off by quality clipping) is almost optimally used by the assembler, but it considerably slows down the process when searching for that. I am working on optimising it.
2. Reading edited experiment files for reassembling edited projects.
3. Writing and possibly reading CAML format. Although this feature is somewhat lower on the TODO list, it nevertheless is there.
4. Assembling ESTs allows to make some assumptions that could be beneficial to the potential gene sequence that is assembled.
5. Others nifty ideas that I have not completely thought out yet.

B.13 Working principles

To avoid the ”garbage-in, garbage-out” problematics, *Mira* uses a ’high quality alignments first’ contig building strategy. This means that the assembler will start with those regions of sequences that have been marked as good quality (high confidence region - HCR) with low error probabilities (the clipping must have been done by the base caller or other preprocessing programs, e.g. *pregap4*) and then gradually extends the alignments as errors in different reads are resolved through error hypothesis verification and signal analysis.

This assembly approach relies on some of the automatic editing functionality provided by the EdIt package which has been integrated in parts within *mira*.

This is an aproximate overview on the steps that are executed while assembling:

1. All the experiment / phd / fasta sequences that act as input are loaded (or the CAF project). Qualities for the bases are loaded from the SCFs if needed.
2. The high confidence region (HCR) of each read is compared with a quick algorithm to the HCR of every other read to see if it could match and have overlapping parts (this is the ’SKIM’ filter).
3. All the reads which could match are being checked with an adapted Smith-Waterman alignment algorithm (banded version). Obvious mismatches are rejected, the accepted alignments form one or several alignment graphs.

4. Optional pre-assembly read extension step: *mira* tries to extend HCR of reads by analysing the read pairs from the previous alignment. This is a bit shaky as reads in this step have not been edited yet, but it can help. Go back to step 2.
5. A contig gets built by building a preliminary partial path through the alignment graph (through in-depth analysis up to a given level) and then adding the most probable overlap candidates to a given contig. Contigs may reject reads if these introduce to many errors in the existing consenus. Errors in regions known as dangerous (for the time being only ALUS and REPT) get additional attention by performing simple signal analysis when alignment discrepancies occur.
6. Optional: the contig can be analysed and corrected by the automatic editor (EdIt).
7. Long term repeats are searched for, bases in reads of different repeats that have been assembled together but differ sufficiently (for EdIT so that they didn't get edited and by phred quality value) get tagged with PRMB.
8. Go back to step 5 if there are reads present that have not been assembled into contigs.
9. Optional post-assembly read-extension: *mira* extends the HCR of the reads that have been assembled into the contigs.
10. Optional: Detection of spoiler reads that prevent joining of contigs. Remedy by shortening them.
11. Optional: Write out a checkpoint assembly file and go back to step 2.
12. The resulting project is written out to files.

B.14 License, Disclaimer and Copyright

Stable versions are all versions with even minor versions, e.g. 1.4.x, 2.0.x ... etc.

Test versions are version with uneven minor versions, e.g. 1.1.x, 1.3.x, ..., 2.1.x, ... etc. and version numbers declared as 'prerelease' or 'release candidate', e.g. 1.xrc1, 1.xrc2, ... etc. .

Restricted versions have as only restriction the number of sequences they allow to assemble. This limit is currently set between 3000 and 4000 sequences.

License Permission to use, copy and distribute test and restricted versions of

this software and its documentation for any purpose is hereby granted without fee, provided that this copyright and notice appears in all copies.

Disclaimer The author disclaims all warranties with regard to this software. Use it at your own risk.

External libraries MIRA uses the excellent Expat library to parse XML files. Expat is Copyright © 1998, 1999, 2000 Thai Open Source Software Center Ltd and Clark Cooper as well as Copyright © 2001, 2002 Expat maintainers.
See `http://www.libexpat.org/` and `http://sourceforge.net/projects/expat/` for more information on Expat.

B.15 Authors

Bastien Chevreux (*mira*): `bach@chevreux.org`
and Thomas Pfisterer (*EdIt*): `t.pfisterer@dkfz-heidelberg.de`

WWW:
`http://www.chevreux.org/projects_mira.html`
`http://www.dkfz-heidelberg.de/mbp-ased/`

B.16 Miscellaneous

If you find this software useful, please send me a postcard. If postcards are not available, a treasure chest full of spanish doubloons, gold and jewellery will do nicely, thank you.

B.17 See Also

EdIt(1), *gap4*(1), *pregap4*(1), *ttuner*(1), *scylla*(1), *pga*(1), *pta*(1), *cap4*(1), *phred*(1), *phrap*(1), *cross_match*(1), *clview*(1), *consed*(1), *caf2gap*(1), *gap2caf*(1), *compress*(1) and *gzip*(1).

Literature

References

Allex, C. F., Baldwin, S. F., Shavlik, J. W. and Blattner, F. R. (1996), Improving the Quality of Automatic DNA Sequence Assembly using Fluorescent Trace-Data Classifications. *Intell. Systems Mol. Biol.*, **4**, 3–14.

Allex, C. F., Baldwin, S. F., Shavlik, J. W. and Blattner, F. R. (1997), Increasing Consensus Accuracy in DNA Fragment Assemblies by Incorporating Fluorescent Trace Representations. *Proceedings, Fifth International Conference on Intelligent Systems for Molecular Biology*, AAAI Press, pp. 3–14, pp. 3–14.

Allison, L. (1993), A fast Algorithm for the Optimal Alignment of Three Strings. *Journal of theoretical Biology*, **164**, 261–269.

Althaus, E., Caprara, A., Lenhof, H.-P. and Reinert, K. (2002), Multiple sequence alignment with arbitrary gap costs: Computing an optimal solution using polyhedral combinatorics. *Bioinformatics*, **18**, S4–S16, Suppl. 2.

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

Anderson, I. and Brass, A. (1998), Searching DNA databases for similarities to DNA sequences: when is a match significant? *Bioinformatics*, **14**(4), 349–356.

Anson, E. L. and Myers, E. W. (1997), A Program for Refining DNA Sequence Multi-Alignments. *Journal of Computational Biology*, **4**(3), 269–283.

Armen, C. and Stein, C. (1995), Short Superstrings and the Structure of Overlapping Strings. *Journal of Computational Biology*, **2**(2), 307–332.

Arslan, A. N., Egecioglu, O. and Pevzner, P. A. (2001), A new approach to sequence comparison: normalized sequence alignment. *Bioinformatics*, **17**(4), 327–337.

Asayama, M., Saito, K. and Kobayashi, Y. (1998), Translational attenuation of the Bacillus subtilis spo0B cistron by an RNA structure encompassing the initiation region. *Nucleic Acids Research*, **26**(3), 824–830.

Baeza-Yates, R. A. and Gonnet, G. H. (1992), A New Approach to Text Searching. *Commun. of the Assoc. for Comp. Mach.*, **35**, 74–82.

Bailey, J. A., Yavor, A. M., Massa, H. F., Trask, B. J. and Eichler, E. E. (2001), Segmental Duplications: Organization and Impact Within the Currant Human Genome Project Assembly. *Genome Research*, **11**, 1005–1017.

Barker, G., Batley, J., O' Sullivan, H., Edwards, K. J. and Edwards, D. (2003), Redundancy based detection of sequence polymorphisms in expressed sequence tag data using autoSNP. *Bioinformatics*, 421–2.

Barton, G. J. (1993), An efficient algorithm to locate all locally optimal alignments between two sequences allowing for gaps. *Computer Applications in the Bioscience*, **9**(6), 729–734.

Baxevanis, A. and Ouellette, B. (eds.) (1998), *Bioinformatics: A Practical Guide to the Analisys of Genes and Proteins*. Wiley-Liss, Inc., ISBN 0-471-19196-5.

Berno, A. J. (1996), A Graph Theoretic Approach to the Analysis of DNA Sequencing Data. *Genome Research*, **6**, 80–91.

Bonfield, J. K., Rada, C. and Staden, Rodger, S. (1998), Automated detection of point mutations using fluorescent sequence trace subtraction. *Nucleic Acids Research*, **26**, 3404–3409.

Bonfield, J. K., Smith, K. F. and Staden, R. (1995a), The application of numerical estimates of base calling accuracy to DNA sequencing projects. *Nucleic Acids Research*, **23**(8), 1406–1410.

Bonfield, J. K., Smith, K. F. and Staden, R. (1995b), A new DNA sequence assembly program. *Nucleic Acids Research*, **23**(24), 4992–4999.

Bonfield, J. K. and Staden, R. (1996), Experiment files and their application during large-scale sequencing projects. *DNA Sequence*, **6**, 109–117.

Boyer, R. S. and Moore, J. S. (1977), A Fast String Searching Algorithm. *Commun. of the Assoc. for Comp. Mach.*, **20**(10), 762–772.

Bray, N., Dubchak, I. and Pachter, L. (2003), AVID: A Global Alignment Program. *Genome Research*, **13**, 97–102.

Bruce, A., Bray, D., Lewis, J., Raff, M., Roberts, K. and Watson, J. D. (1994), *Molecular biology of the cell*. Garland Publishing, 3rd edn.

Bucher, P. and Hofmann, K. (1996), A Sequence Similarity Search Algorithm Based on a Probabilistic Interpretation of an Alignment Scoring System. *Intell. Systems Mol. Biol.*, **4**, 44–51.

Camargo, A. A., Samaia, Helena P.B. Dias-Neto, E., Simao, D. F. and Migotto, I. A. e. a. (2001), The contribution of 700,000 ORF sequence tags to the definition of the human transcriptome. *Proceedings of the National Academy of Sciences of the United States of America*, **98**(21), 12103–12108.

Chan, S. C., Wong, A. K. C. and Chiu, D. K. Y. (1992), A survey of multiple sequence comparison methods. *Bulletin of Mathematical Biology*, **54**(4), 563–598.

Chao, K.-M., Hardison, R. C. and Miller, W. (1994), Recent Developments in Linear-Space Alignment Methods: A Survey. *Journal of Computational Biology*, **1**(4), 271–291.

Chao, K.-M., Zhang, J., Ostell, J. and Miller, W. (1995), A local alignment tool for very long DNA sequences. *Computer Applications in the Bioscience*, **11**(2), 147–153.

Chen, T. and Skiena, S. S. (2000), A case study in genome-level fragment assembly. *Bioinformatics*, **16**(6), 494–500.

Cheung, J., Estivill, X., Khaja, R., MacDonald, J. R., Lau, K., Tsui, L.-C. and Scherer, S. W. (2003), Genome-wide detection of segmental duplications and potential assembly errors in the human genome sequence. *Genome Biology*, **4**(4), R25.1–R25.10.

Chou, H.-H. and Holmes, M. H. (2001), DNA sequence quality trimming and vector removal. *Bioinformatics*, **17**(12), 1093–1104.

Dardel, F. (1985), A microcomputer program for comparison and alignment of DNA sequence gel readings. *Computer Applications in the Bioscience*, **1**(3), 173–175.

Dear, S., Durbin, R., Hilloier, L., Marth, G., Thierry-Mieg, J. and Mott, R. (1998), Sequence Assembly with CAFTOOLS. *Genome Research*, **8**, 260–267.

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

DOE (1992), DOE Human Genome Program: Primer on Molecular Genetics. Tech. rep., U.S. Department of Energy; Office of Energy Research; Office of Health and Environmental Research, Washington, DC 20585.

Durbin, R. and Dear, S. (1998), Base Qualities Help Sequencing Software. *Genome Research*, **8**, 161–162.

Eichler, E. E. (2001), Segmental Duplications: What's Missing, Misassigned, and Misassembled – and Should We Care? *Genome Research*, **11**, 653–656.

Engle, M. and Burks, C. (1993), Artificially generated data sets for testing DNA fragment assembly algorithms. *Genomics*, **16**, 286–288.

Engle, M. and Burks, C. (1994), GenFrag 2.2: New features for more robust fragment assembly benchmarks. *Computer Applications in the Bioscience*, **10**, 567–568.

Ewing, B. and Green, P. (1998), Base-Calling of Automated Sequencer Traces Using Phred. II. Error Probabilities. *Genome Research*, **8**, 186–194.

Ewing, B., Hillier, L., Wendl, M. C. and Green, P. (1998), Base-Calling of Automated Sequencer Traces Using Phred. I. Accuracy Assessment. *Genome Research*, **8**, 175–185.

Felsenstein, J., Sawyer, S. and Kochin, R. (1982a), An efficient method for matching nucleic acid sequences. *Nucleic Acids Research*, **10**, 133–139.

GCB99 (1999), *Computer Science and Biology: Proceedings of the Germanc Conference on Bioinformatics GCB '99*, GBF-Braunschweig, Dep. of Bioinformatics.

Giegerich, R. (2000), A systematic approach to dynamic programming in bioinformatics. *Bioinformatics*, **16**(8), 665–77.

Giegerich, R. and Wheeler, D. (1996), Pairwise Sequence Alignment. http://www.techfak.uni-bielefeld.de/bcd/Curric/PrwAli/prwali.html.

Giladi, E., Walker, M., Wang, J. and Volkmuth, W. (2002), SST: an algorithm for finding near-exact sequence matches in time proportional to the logarithm of the database size. *Bioinformatics*, 873–877.

Gordon, D., Abajian, C. and Green, P. (1998), Consed: A Graphical Tool for Sequence Finishing. *Genome Research*, **8**, 195–202.

Gotoh, O. (1993), Optimal alignment between groups of sequences and its application to multiple sequence alignment. *Computer Applications in the Bioscience*, **9**(3), 361–370.

Grice, J. A., Hughey, R. and Speck, D. (1997), Reduced Space Sequence Alignment. *Computer Applications in the Bioscience*, **13**(1), 45–53.

Grillo, G., Attimonelli, M., Luici, S. and Pesole, G. (1996), CLEANUP: a fast computer program for removing redundancies from nucleotide sequence databases. *Computer Applications in the Bioscience*, **12**(1), 1–8.

Gronek, G. (1995a), Ähnlichkeiten gesucht: Fehlertoleranter Suchalgorithmus 'Shift-AND'. *c't*, **5**, 294–301.

Gronek, G. (1995b), Optimal schnell: Schnelle Textsuche mit 'Optimal Mismatch'. *c't*, **3**, 278–284.

Guan, X. and Uberbacher, E. C. (1996), Alignments of DNA and protein sequences containing frameshift errors. *Computer Applications in the Bioscience*, **12**(1), 31–40.

Heber, S., Alekseyev, M., Sze, S., Tang, H. and Pevzner, P. (2002), Splicing graphs and EST assembly problem. *Bioinformatics*, S181–8, Suppl 1.

Huang, X. (1994), On global sequence alignment. *Computer Applications in the Bioscience*, **10**(3), 227–235.

Huang, X. (1996), An Improved Sequence Assembly Program. *Genomics*, **33**, 21–31.

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

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

Jaffe, D. B., Butler, Jonathan Gnerre, S., Mauceli, E., Lindblad-Toh, K., Mesirov, J. P., Zody, M. C. and Lander, E. S. (2003), Whole-Genome Sequence Assembly for Mammalian Genomes: Arachne 2. *Genome Research*, **13**(1), 91–96.

Johnston, R. E., Mackenzie, J. J. and Dougherty, W. (1986), Assembly of overlapping DNA sequences by a program written in BASIC for 64K CP/M and MS-DOS IBM-compatible microcomputers. *Nucleic Acids Research*, **14**(1), 517–527.

Katoh, K., Misawa, K., Kuma, K.-i. and Miyata, T. (2002), MAFFT: a novel method for rapid multiple sequence alignment based on fast Fourier transform. *Nucleic Acids Research*, **30**(14), 3059–3066.

Kececioglu, J. D. and Myers, E. W. (1992), Combinatorial algorithms for DNA sequence assembly. Tech. Rep. TR 92-37, University of California at Davis, University of Arizona.

Keith, J., Adams, P., Bryant, D., Kroese, D.P. andMitchelson, K., Cochran, D. and Lala, G. (2002), A simulated annealing algorithm for finding consensus sequences. *Bioinformatics*, 1494–1499.

Kent, J. W. (2002), BLAT – The BLAST-Like Alignment Tool. *Genome Research*, **12**, 656–664.

Kleinjung, J., Douglas, N. and Heringa, J. (2002), Parallelized multiple alignment. *Bioinformatics*, 1270–1271.

Klug, W. S. and Cummings, M. R. (1996), *Essential of Genetics*. Prentice Hall, 2nd edn.

Kumar, S. and Rzhetsky, A. (1996), Evolutionary relationships of eukaryotic kingdoms. *Journal of Molecular Evolution*, **42**, 183–193.

Lario, A., González, A. and Dorado, G. (1997), Automated Laser-Induced Fluorescence DNA Sequencing: Equalizing Signal-to-Noise Ratios Significantly Enhances Overall Performance. *Analytical Biochemistry*, **247**, 30–33.

Lassmann, T. and Sonnhammer, E. L. (2002), Quality assessment of multiple alignment programs. *FEBS Letters*, **529**(1), 126–130.

Lawrence, C. B., Honda, S., Parrott, N. W., Flood, T. C., Ghu, L., Zhang, L., Jain, M., Larson, S. and Myers, E. W. (1994), The Genome Reconstruction

Manager: A Software Environment for Supporting High-Throughput DNA Sequencing. *Genomics*, **23**, 192–201.

Lee, C., Grasso, C. and Sharlow, M. F. (2002), Multiple sequence alignment using partial order graphs. *Bioinformatics*, **18**(3), 452–464.

Lipshutz, R. J., Taverner, F., Henessy, K., Hartzell, G. and Davis, R. (1994), DNA Sequence Confidence Estimation. *Genomics*, **19**, 417–424.

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

Miller, W. (2001), Comparison of genomic DNA sequences: solved and unsolved problems. *Bioinformatics*, **17**(5), 391–397.

Morgenstern, B., Dress, A. and Werner, T. (1996), Multiple DNA and protein sequence alignment based on segment-to-segment comparison. *Proceedings of the National Academy of Science USA*, **93**, 12098–12103.

Morgenstern, B., Goel, S., Sczyrba, A. and Dress, A. (2003), AltAVisT: Comparing alternative multiple sequence alignments. *Bioinformatics*, **19**(3), 425–426.

Müller, W. E. (2001), How was metazoan threshold crossed: the hypothetical Urmetazoa (part A). *Comparative Biochemistry and Physiology*, **129**, 433–460.

Myers, E. W. (1991), An Overview of Sequence Comparison Algorithms in Molecular Biology. Tech. Rep. 29, Department of Computer Science; The University of Arizona, Tucson, Arizona 85721.

Myers, E. W. (1994), *Advances in Sequence Assembly*, Academic Press. pp. 231–238.

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

Myers, G. (1999), A Whole Genome Assembler for Drosophila. GCB99 (1999), p. 44.

Myers, G., Selznick, S., Zhang, Z. and Miller, W. (1996), Progressive Multiple Alignment with Constraints. *Journal of Computational Biology*, **3**(4), 563–572.

Needleman, S. and Wunsch, C. (1970), A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, **48**(3), 443–453.

Nickerson, D. A., Taylor, S. L. and Rieder, M. J. (2000), Identifying Single Nucleotide Polymorphisms (SNPs) in Human Candidate Genes. *Research Abstracts from the DOE Human Genome Program Contractor-Grantee Workshop VIII*, February 27-March 2, Santa Fe, NM.

Ning, Z., Cox, A. J. and Mullikin, J. C. (2001), SSAHA: A Fast Search Method for Large DNA Databases. *Genome Research*, **11**, 1725–1729.

Notredame, C. (2002), Recent progress in multiple sequence alignment: a survey. *Pharmacogenomics*, **3**, 131–144.

Notredame, C. and Higgins, D. G. (1996), SAGA: sequence alignment by genetic algorithm. *Nucleic Acids Research*, **24**(8), 1515–1524.

Notredame, C., Holm, L. and Higgins, D. G. (1998), COFFEE: an objective function for multiple sequence alignments. *Bioinformatics*, **14**(5), 407–422.

Otu, H. H. and Sayood, K. (2003), A divide-and-conquer approach to fragment assembly. *Bioinformatics*, **19**(1), 22–29.

Paracel (2002a), *Paracel Filtering Package User Manual*. Paracel Inc., 1055 E. Colorado Blvd; Pasadena, CA 91106.

Paracel (2002b), *PGA: Paracel Genome Assembler User Manual*. Paracel Inc., 1055 E. Colorado Blvd; Pasadena; CA 91106.

Paracel (2002c), *PTA: Paracel TranscriptAssembler User Manual*. Paracel Inc., 1055 E. Colorado Blvd; Pasadena, CA 91106.

Parsons, R., Forrest, S. and Burks, C. (1993), Genetic Algorithms for DNA Sequence Assembly. L. Hunter, D. B. Searls, J. W. S. (ed.), *Proc. of the 1st International Conference on Intelligent Systems for Molecular Biology*, AAAI, Bethesda, MD, USA, pp. 310–318, pp. 310–318, ISBN 0-929280-47-4.

Pearson, W. R. (1995), Comparison of Methods for Searching Protein Sequence Databases. *Protein Science*, **4**, 1145–1160.

Pearson, W. R. (1998), Empirical Statistical Estimates for Sequence Similarity Searches. *Journal of Molecular Biology*, **276**, 71–84.

Peltola, H., Söderlund, H. and Ukkonen, E. (1984), SEQAID: a DNA sequence assembling program based on a mathematical model. *Nucleic Acids Research*, **12**(1), 307–321.

Pevzner, P. A. and Tang, H. (2001), Fragment assembly with double-barreled data. *Bioinformatics*, **17**, S225–S233, Suppl. 1.

Pevzner, P. A., Tang, H. and Waterman, M. (2001), An Eulerian path approach to DNA fragment assembly. *Proceedings of the National Academy of Science USA*, **98**, 9748–9753.

Pfisterer, T. and Wetter, T. (1999), *Computer Assisted Editing of Genomic Sequences - Why and How We Evaluated a Prototype*. Puppe (1999), pp. 201–209.

Prunella, N., Luini, S., Attimonelli, M. and Pesole, G. (1993), FASTPAT: a fast and efficient algorithm for string searching in DNA sequences. *Computer Applications in the Bioscience*, **9**(5), 541–545.

Puppe, F. (ed.) (1999), *XPS-99: Knowledged-Based Systems*. Lecture Notes in Artificial Intelligence; Subseries of Lecture Notes in Computer Science, Springer-Verlag, Berlin Heidelberg New York.

Rajasekaran, S., Jin, X. and Spouge, J. (2002), The efficient computation of position-specific match scores with the fast fourier transform. *Journal of Computational Biology*, **9**(1), 23–33.

Reinert, K., Stoye, J. and Will, T. (2000), An iterative method for faster sum-of-pairs multiple sequence alignment. *Bioinformatics*, **16**(9), 808–814.

Richterich, P. (1998), Estimation of Errors in "Raw" DNA Sequences: A Validation Study (Letter). *Genome Research*, **8**, 251–259.

Rosenblum, B., Lee, L., Spurgeon, S., Khan, S., Menchen, S., Heiner, C. and Chen, S. (1997), New dye-labeled terminators for improved DNA sequencing patterns. *Nucleic Acids Research*, **25**(22), 4500–4504.

Sanders, J. Z., Petterson, A. A., Hughes, P. J., Connell, C. R., Raff, M., Menchen, S., Hood, L. E. and Teplow, D. B. (1991), Imaging as a tool for improving length and accuracy of sequence analysis in automated fluorescence-based DNA sequencing. *Electrophoresis*, **12**, 3–11.

Sanger, F., Nicklen, S. and Coulson, A. (1977), DNA sequencing with chain-terminating inhibitors. *Proceedings of the National Academy of Science USA*, **74**, 5463–5467.

Schlosshauer, M. and Ohlsson, M. (2002), A novel approach to local reliability of sequence alignments. *Bioinformatics*, **18**(6), 847–854.

Schuler, G. D. (1997), Pieces of the puzzle: expressed sequence tags and the catalog of human genes. *J Mol Med*, **75**, 694–698.

Schuler, G. D. (1998), *Sequence Alignmnent and Database Searching*. Baxevanis and Ouellette (1998), pp. 145–171, ISBN 0-471-19196-5.

Shpaer, E. G., Robinson, M., Yee, D., Candlin, J. D., Mines, R. and Hunkapiller, T. (1996), Sensitivity and Selectivity in Protein Similarity Searches: A Comparison of Smith-Waterman in Hardware to BLAST and FASTA. *Genomics*, **38**(2), 179–191.

Smith, T. F. and Waterman, M. S. (1981), Identification of Common Molecular Subsequences. *Journal of Molecular Biology*, **147**, 195–197.

Smith, T. F., Waterman, M. S. and Fitch, W. M. (1981), Comparative Biosequence Metrics. *Journal of Molecular Evolution*, **18**, 38–46.

Staden, R. (1984), Computer methods to aid the determination and analysis of DNA sequences. *Biochemical Society Transaction*, **12**(6), 1005–1008.

Staden, R. (1989), Methods for calculating the probabilities of finding patterns in sequences. *Computer Applications in the Bioscience*, **5**(2), 89–96.

Staden, R. (1996), The Staden Sequence Analysis Package. *Molecular Biotechnology*, **5**, 233–241.

Staden, R., Bonfield, J. and Beal, K. (1997), *The New Staden Package Manual - Part 1*. Medical Research Council, Laboratory of Molecular Biology.

Stoye, J. (1998), Multiple sequence alignment with the divide-and-conquer method. *Gene / GC*, **211**, 45–56.

Sunday, D. M. (1990), A very fast substring search algorithm. *Commun. of the Assoc. for Comp. Mach.*, **33**(8), 132–142.

Tammi, M. T., Arner, E., Britton, T. and Andersson, B. (2002), Separation of nearly identical repeats in shotgun assemblies using defined nucleotide positions, DNPs. *Bioinformatics*, **18**(3), 379–88.

Thompson, J. D., Plewniak, F. and Poch, O. (1999a), A comprehensive comparison of multiple sequence alignment programs. *Nucleic Acids Research*, **27**(13), 2682–2690.

Thompson, J. D., Plewniak, F. and Poch, O. (1999b), A comprehensive comparison of multiple sequence alignmnet programs. *Nucleic Acids Research*, **27**(13), 2682–2690.

Venter, J. C., Adams, M. D., Myers, E. W., Li, P. W., Mural, R. J., Sutton, G. G., Smith, H. O., Yandell, M., Evans, C. A., Holt, R. A., Gocayne, J. D., Amanatides, P., Ballew, R. M., Huson, D. H. and Wortman, J. R. e. a. (2001), The Sequence of the Human Genome. *Science*, 1304–1351.

Walther, D., Bartha, G. and Morris, M. (2001), Basecalling with LifeTrace. *Genome Research*, **11**, 875–888.

Wang, J., Ka-Shu, G. W., Wang, J., Ni, P., Han, Y., Huang, X., Zhang, J., Ye, C., Zhang, Y., Hu, J., Zhang, K., Xu, X., Cong, L., Lu, H., Ren, X., Ren, X., Dai, D., He, J., Tao, L., Passey, D. A., Yang, H., Yu, J. and Li, S. (2002), RePS: A Sequence Assembler That Masks Exact Repeats Identified from the Shotgun Data. *Genome Research*, **12**, 824–831.

Wang, L. and Jiang, T. (1994), On the Complexity of Multiple Sequence Alignment. *Journal of Computational Biology*, **1**(4), 337–348.

Wilbur, W. and Lipman, D. J. (1983), Rapid similarity searches of nucleic acid and proteins data banks. *Proceedings of the National Academy of Science USA*, **80**, 726–730.

Wu, S. and Manber, U. (1992a), Approximate Pattern Matching. *Byte Magazine*, **11**, 281–292.

Wu, S. and Manber, U. (1992b), Fast Text Searching Allowing Errors. *Commun. of the Assoc. for Comp. Mach.*, **35**(10), 83–91.

Xu, Y., Mural, R. J. and Uberbacher, E. C. (1995), Correcting sequencing errors in DNA coding regions using dynamic programming approach. *Computer Applications in the Bioscience*, **11**(2), 117–124.

Yu, Z., Li, T., Zhao, J. and Luo, J. (2002), PGAAS: a prokaryotic genome assembly assistant system. *Bioinformatics*, **18**(5), 661–665.

Zhang, C. and Wong, A. K. (1997), A genetic algorithm for multiple molecular sequence alignment. *Computer Applications in the Bioscience*, **13**(6), 565–581.

Own publications

Chevreux, B., Pfisterer, T., Drescher, B., Driesel, A. J., Müller, W. E., Wetter, T. and Suhai, S. (2004), Using the miraEST Assembler for Reliable and Automated mRNA Transcript Assembly and SNP Detection in Sequenced ESTs. *Genome Research*, **14**(6).

Chevreux, B., Pfisterer, T. and Suhai, S. (2000), Automatic Assembly and Editing of Genomic Sequences. *Genomics and Proteomics – Functional and Computational Aspects*, Kluwer Academic/Plenum Publishers, New York, chap. 5, pp. 51–65.

Chevreux, B., Wetter, T. and Suhai, S. (1999), Genome Sequence Assembly Using Trace Signals and Additional Sequence Information. *Computer Science and Biology: Proceedings of the German Conference on Bioinformatics GCB '99*, GCB, pp. 45–56.

Index

∇-character, 12
$*$-character, 7

ADS, *see* aligned dual sequence
AFH, *see* atomic fault hypotheses
algorithms
  n, m recursive look-ahead, 66
  design techniques, 111
  DNA-Shift-AND, 41–44
  dynamic programming, 40
  dynamic programming, 16
    parametrising, 58
  fast fourier transform, 40
  greedy, 66
  heuristic string searching, 41
  optimal mismatch, 40
  read extension, 84
  requirements, 109
  string comparison, 40
  window search, 85
  ZEBRA, 45–53
aligned dual sequence, 84
aligned dual sequence, 61
alignment, 12
  calculation, 16
  coverage, 13, 65, 83
  expected score, 59
  global, 15, 40
  k-tuple, 13
  local, 15, 40, 54
  matrix, 16
  optimal, 67
  optimality, 28
  quality, 60
  score, 14, 59
  score ratio, 60
  SCS, 15
  weight, 61
alphabet, 6
  IUPAC, 7
  representations, 7
area pooling, 36
assembler
  comparison, 95
  GAP4, 95
  integrated assembler-editor strategy, 28
  MIRA, 94, 95
  PHRAP, 26, 95
  strategies, 25
  strategy, 4
assembly
  layout, 62, 83
  projects, 95
atomic fault hypotheses, 77
automatic editor, 31, 71
automatic editor, 4, 77
  strategy, 77

base

confidence value, 3
dideoxynucleotide termination, 10
extraction, *see* base-calling
probability, 3, 19, 71
quality, 2, 18, 19
base-calling, 11
alternative, 71
errors, 18
PHRED, 19
block-indel model, 59
building blocks, *see* read

chimera, 20, 66
clipping
quality, 36
sequencing vector, 36
splice variants, 36
complement
base, 6
complexity
banded dynamic programming, 56
DNA-Shift-AND, 42
dynamic programming, 16, 41, 56
consensus
constraints, 69
errors, 65
sequence, 65
contamination, 20
contig, 25, 67
building, 62
anchor, 64
anchor point, 68
iterative approach, 63
result, 64
starting point, 64
strategies, 64
minimum number, 62
object, 67
quality improvement, 86
read acceptance, 68
read acceptance, 67
read rejection, 67
coverage, *see* alignment
cross_match, 95
cytochromes, 23

data
burst, 50
speculative prefetch, 50
distribution
bias, 11
random, 47
uniform, 11, 80
DNASAND, *see* algorithms
dye
termination, 10
dynamic programming, *see* algorithms

EdIt, *see* automatic editor
electrophoresis, 10
endgaps, 12
exon, 21
expert system, *see* automatic editor

fast fourier transform, *see* algorithms
finishing, 3
fuguization, 26

gap
character, *see* $*$-character
penalty function, 59
genome, 6
graph
edge, 62
look-ahead, 66

node, 62
number of edges, 65
searching, 64
weighted, 62

hash
calculation, 45
distance, 46
distance histogram, 48
imprint, 46
index table, 46
position, 46
HCR, *see* regions
hidden data, 36
histogram, 48
Human Genome Project, 3

imprint, *see* hash
indel, 18, 20
insert, 8
intron, 21
IUPAC, *see* alphabet

layout, *see* assembly
LCR, *see* regions
Levenshtein distance, 42

matrix
score, 14, 59
weight, 14, 59
micro-satellites, *see* repeat
MIRA
acronym, 94
misassembly
detection, 82
prevention, 83

NP
complete, 16
hard, 15

optimal mismatch, *see* algorithms
overlap
detection, 38

pattern analysis, *see* repeat
PHRAP, *see* assembler
PHRED, *see* base-calling
plasmid, 8
point mutation, 20
polybase
poly-A / poly-T uncovering, 37
possible repeat marker base, *see* repeat
PRMB, *see* repeat

read, 11
building block, 68
chimeras, 67, 69
extension, 83
algorithm, 84
extra-contig, 84
intra-contig, 84
fast scanning, 38
region
error, 77
low confidence (LCR), 84
single-base error, 70
regions
hidden data, 83
high confidence (HCR), 28, 37, 84
low confidence (HCR), 37
low confidence (LCR), 28
regular expressions, 38
repeat
in graphs, 64
long, 79

micro-satellites, 78
pattern, 82
pattern analysis, 80
PRMB: possible repeat marker base, 83
recognition, 78, 80
short, 78
standard elements, 69
standard types, 31
types, 78
restriction enzyme, 8

Sanger
nucleotide labelling, 10
scaffolding, 26
score
alignment, 14
column, 14
of two bases, 14
ratio, 60
score matrix, *see* matrix
sequence, 7
comparison, 50
contamination, 20
definition, 7
error rate, 2
extended length, 13
length, 2, 7, 8
quality, 18
quality improvement, 86
sequences
comparing, 46
sequencing vector, 35
Shift-AND, *see* algorithms
shotgun method, 8
signal
base extraction, *see* base-calling
fluorescence detection, 10
quality, 17
quality measures, 77
shape information, 71
signal-to-noise ratio, 83
trace, 10, 71, 77
translation, *see* base calling
signal analysis
advantages, 4
black box, 71
signal analysis, 70
singlet, 77
SNP, 20
sonication, 8
splicing, 22

template
constraints, 72
sequencing, 72
trace, *see* signal

vector
cloning, 8
sequencing, 8

weight matrix, *see* matrix

ZEBRA, *see* algorithms

Curriculum Vitae

Particulars

Name:	Bastien Chevreux
Date of birth:	05.07.1972
Nationality:	french
Place of birth:	Duisburg, Federal Republic of Germany
Marital status:	single
Parents:	Bernard Chevreux
	Lydia Chevreux (born Simon)

Schooling

1978 – 1979	Grundschule Duisburg *Duisburg, Federal Republic of Germany* Primary school: first class
1979 – 1983	Deutsche Schule Brüssel *Brussels, Belgium* Primary school: second to fourth class Comprehensive secondary school: fifth class
1983 – 1991	Gymnasium Thomaeum Kempen *Kempen, Federal Republic of Germany* Comprehensive secondary school: 6th to 13th class
June 12th, 1991	Gymnasium Thomaeum Kempen *Kempen, Federal Republic of Germany* General qualification for university entrance (Abitur)

University

Summer term 1992	Universität Heidelberg / FH Heilbronn *Heidelberg / Heilbronn, Germany* Start of university studies in Medical Informatics
April 13th, 1994	Intermediate diploma
March 1st, 1997	Diploma of the University of Heidelberg in Medical Informatics

Acknowledgements

"I'm all in favour of keeping dangerous weapons out of the hand of fools. Let's start with typewriters." (Solomon Short)

I would first like to thank Thomas Pfisterer, Prof. Dr. Sándor Suhai, Dr. Bernd Drescher and Prof. Dr. Thomas Wetter for their invaluable input, hours of discussion, encouragement and a plethora of good ideas while working on this research project.

I also want to acknowledge and thank Dr. Gerald Nyakatura, Dr. Matthias Platzer and Dr. Uwe Menzel at the former genome sequencing group of the IMB Jena for their patience and numerous enhancement suggestions whilst explaining to me the mysteries of DNA and the shotgun sequencing process. Dr. Jacqueline Weber did the same for unravelling splicing in eukaryotic genomes. Needless to say that all errors that slipped into this thesis while writing about those topics are mine.

To Dr. Andrea Hörster I am most grateful for reading a preliminary version of this thesis very, very thoroughly.

I'd like to thank Prof. Dr. Werner E.G. Müller (University Mainz), Prof. Dr. Albert J. Driesel (VitiGen AG) and Prof. Dr. Jörn Bullerdiek (University Bremen) for kindly providing EST datasets

James Bonfield (now at the Wellcome Trust Sanger Institute) was most helpful for making `mira` compatible in output to the Staden package.

Without Volker Schmidt (DLR Lampoldshausen), I'd still sit – sometimes quite perplex – in front of the mysteries hidden within the depths of LaTeX and PDF creation. The same applies for helpful souls in the USENET de.comp.text.tex newsgroup.

Last, but not certainly not least, Silke Mink supported me all along the last two years of this thesis, cheering me up and encouraging me to continue whenever I needed it.

Thanks a lot . . . to all of you!