

# Implementation of a Read Mapping Tool Based on the Pigeon-hole Principle



## Bachelor Thesis

Algorithmic Bioinformatics Group  
Freie Universität Berlin, 2008–08–20

*Author:*



Konrad Ludwig Moritz RUDOLPH  
⟨krudolph@mi.fu-berlin.de⟩

*Supervisors:*

Prof. Dr. Knut REINERT  
Dr. Gunnar KLAU

## **Abstract**

A new big challenge in computational biology is the mapping of large numbers of short oligonucleotide reads to a reference genome while allowing a few errors. This thesis describes a method applied in the successful ELAND tool to solve this problem and attempts an implementation in SEQAN, a bioinformatical C++ library.

This document is licensed under the  
Creative Commons license 3.0  .

# CONTENTS

<b>Contents</b>	<b>ii</b>
<b>I Introduction</b>	<b>I</b>
1.1 Motivation . . . . .	1
1.2 Read mapping . . . . .	2
1.3 Overview . . . . .	2
<b>II Background and Related Work</b>	<b>4</b>
2.1 Solexa . . . . .	4
2.2 Eland . . . . .	5
2.3 Similar tools . . . . .	5
2.4 Comparison . . . . .	6
<b>III Algorithm</b>	<b>8</b>
3.1 General description . . . . .	8
3.2 Index data structure . . . . .	9
3.2.1 Building the index . . . . .	10
3.2.2 Lookup in the index . . . . .	13
3.3 Gapped $q$ -gram . . . . .	14
3.4 Eland shapes . . . . .	14
3.5 Hashing . . . . .	15
3.6 Verification . . . . .	17
3.7 Synthesis . . . . .	21
<b>IV Implementation Details</b>	<b>23</b>
4.1 Code complexity . . . . .	23
4.2 SeqAn features . . . . .	23
4.3 Implementation . . . . .	24
4.4 Restrictions . . . . .	26

<b>V Results</b>	<b>27</b>
5.1 Test setup . . . . .	27
5.2 Data . . . . .	27
5.3 Interpretation . . . . .	28
5.4 Outlook and future work . . . . .	29
<b>A Acknowledgement</b>	<b>30</b>
<b>B Colophon</b>	<b>31</b>
<b>Bibliography</b>	<b>32</b>



# INTRODUCTION



## 1.1 Motivation

One of the largest problems in computational biology remains the analysis of whole genomes. Despite the now-finished sequencing of the human genome [VIR<sup>+</sup>01] and other accomplished whole-genome projects (e.g. [WLTBL02, LSV07]), sequencing large genomes remains an active challenge because it is becoming even more relevant in the context of research as well as diagnostics, where it has just begun to take foothold in the form of *personalised medicine*.

The challenge of sequencing genomes is actually twofold: First, the genomic data has to be generated. Generating the data requires machines that can carry out many chemical reactions very quickly and with minimal loss of information. The current trend towards miniaturization helps this greatly by allowing to manipulate more chemical samples at the same time and in smaller doses. The next section will explore how this particular challenge is met. Afterwards, these experiments have to be brought into digital form, calling for efficient signal processing algorithms. However, this is beyond the scope of this work.

The second great challenge is to process the raw data. In the case of sequencing, this means assembling relatively short reads of DNA into longer units, ideally whole genomes. There are several fundamentally different approaches that yield different types of raw data. Two varying factors are the length and the quality of the sequence reads emitted. Depending on these two factors, different algorithms have been developed to assemble the reads. Ideally, reads would be very long with very low error rates. Unfortunately, these two factors are mutually exclusive in practice: The longer the reads produced, the higher the error rate.

Over the years, there has been a constant advancement in the field of se-

quencing and read assembly. During the time of the human genome project alone, the cost dropped from \$10 down to \$0.1 per sequenced base, a hundred-fold decrease [CMP03]. Regrettably, this is still not enough to satisfy the demands of large-scale genomics which remains expensive and time-consuming.

To bypass this, one can fall back to existing genomes. For example, companies like “23andMe<sup>1</sup>” which offer \$999 genetic tests rely on the similarity of individual human genomes. In order to analyse single nucleotide polymorphisms (“SNPS”) it isn’t actually necessary to assemble a whole genome from scratch. Instead, the reads can be aligned to an existing reference sequence under the assumption that the two sequences are reasonably similar. This shifts the whole problem domain from one of assembly to one of genome mapping.

## 1.2 Read mapping

Unlike whole genome assembly, which has to create a DNA sequence *de novo*, read mapping is the process of aligning small oligonucleotide reads to a reference genome.

**Definition 1.** Let  $S$  be a reference sequence and let  $P$  be an oligonucleotide read of length  $|P| \ll |S|$ . Then mapping the read to the reference is the process of finding the best match position  $i$  that minimizes the distance (= number of differences) between  $P$  and  $S_{i\dots i+|P|-1}$ . At the outset, there is no restriction on the definition of *distance*: different metrics may be applied, depending on the specific requirements.

The focus of this thesis is the creation of such a read mapping tool that can handle large genomes (in an order of magnitude of 2 gigabases) and a large number (several millions) of reads of predetermined length efficiently.

## 1.3 Overview

To start off, we will examine the research that has been done in the field of comparative genomics and whole-genome mapping. We will have a look at

---

<sup>1</sup>[www.23andme.com/](http://www.23andme.com/)

ELAND, the tool that dominates the field and similar tools that perform similarly well. In order to delimit the scope of this thesis we will also have a look at the performance of these tools to underline the challenge and the expected reward of writing such a tool (chapter II). Next, the theoretical foundation for the implementation is laid out. This also offers a rationale for several feature trade-offs in the implementation (chapter III). After the theory we will focus on the realization of the algorithm and give an insight into the nooks and crannies of the implementation. There, we also offer reasons for using the SEQAN library in the implementation of such tools (chapter IV).

Finally, we will compare my implementation to the original ELAND and discuss the differences observed (chapter V).

# BACKGROUND AND RELATED WORK



## 2.1 Solexa

In response to the challenge outlined above, several strategies have spawned under the tag name of next generation sequencing (NGS), also called ultra-low cost sequencing (ULCS). Compared to the classical method that combines chain termination with electrophoresis, these methods have so far succeeded to cut the prices of sequencing by several orders of magnitude [Shao07]. It is important to emphasize that these methods are not necessarily related, nor are they all new methods. Rather, they have been developed independently from the Sanger chain termination method. However, due to their higher technical complexity, these methods have only recently become interesting in practice due to technical breakthroughs in several areas [SMVCo4].

SOLEXA INC. has been the first company to announce the sequencing of one gigabase at the cost of \$100.000, achieving a per-base price of \$0.0001. The first step in their approach is the fragmentation of the extracted DNA into single-stranded pieces of 100 – 200 bp (base pairs). Contrary to similar cyclic-array sequencing techniques, no initial amplification step is needed, making the method faster. The fragments are attached to a primer sequence that, in turn, is attached to an anchor molecule. By means of this anchor molecule, the primed fragments are fixed to a Single Molecule Array. Single Array Molecule chips, unlike conventional arrays, are unaddressed and unordered. Additionally, each site only contains one molecule. Combined, this means that they have a much higher density of information. Each array thus can contain hundreds of millions of DNA fragments.

After this initial preparation, the following operations are repeated in cycles to sequence all fragments simultaneously: Fluorescently labelled nucleotides and a polymerase are added to the array. Each base pair uses one specific dye.

At the free site nearest to the primer, the labelled nucleotides form base pairs with the nucleotides on the fragments with the aid of the polymerase. Free nucleotides are then washed away. Now, the laser light of four different wave lengths excite the different dyes. The fluorescence is registered by a CCD camera for all fragments on the array as differently coloured spots. After that, the fluorescence (but not the paired base) is removed and the cycle is repeated [BBC<sup>+</sup>05, Stuo2].

This technique produces a large number of oligonucleotide reads of very short length (currently about 30 bp) with relatively few errors.

## 2.2 Eland

It is now necessary to map the resulting reads efficiently to the reference sequence. Conventional alignment tools like BLAST are too slow for this task by orders of magnitude. ELAND relies on the fact that the reads are very short (currently, ELAND handles lengths of up to 32 bases), contain very few errors (at most 2) and that these errors are only results of mismatches, not gaps. Chapter III elaborates on how the algorithm works in detail. For the moment, it's enough to say that ELAND's restrictions are born from a technological necessity: ELAND's good performance is anchored in the strict assumptions it makes for the input.

While conventional tools have to scan the whole reference sequence for matches, ELAND uses an index data structure that allows to jump directly to potential hit positions. As a consequence, large parts of the reference are not looked at in detail and so do not influence the running time heavily. Instead, ELAND's performance is largely bounded by the number of oligonucleotide reads and the expected number of corresponding hits. The resulting runtime is only a tiny fraction of BLAST's: ELAND is roughly 1000 times faster for typical input.

## 2.3 Similar tools

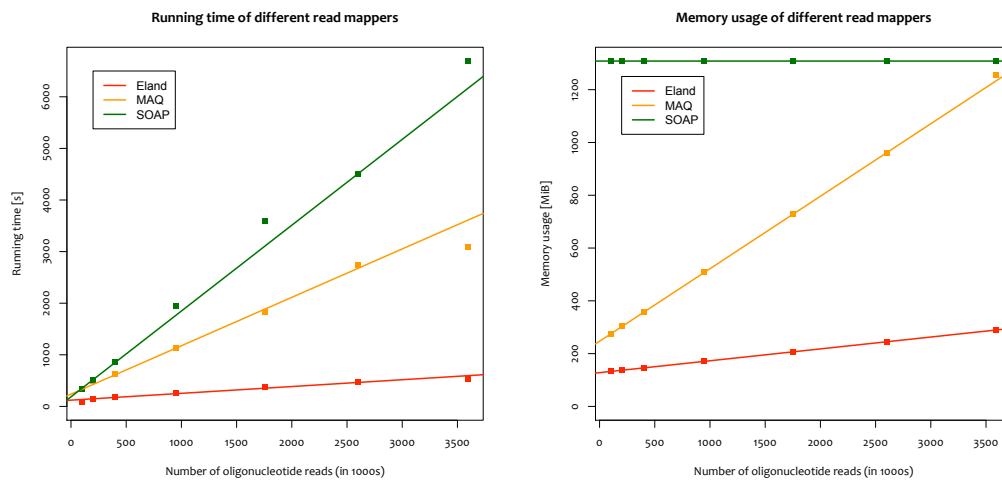
Efforts have been made to replicate the good performance of ELAND while allowing for more dynamic settings or higher error thresholds. Let the fol-

lowing two tools act as representatives for the rest. SOAP allows slightly larger oligonucleotide reads and also handles one continuous gap in the alignment. Furthermore, it gives better control over the handling of multiple matches and the selection of the best hit. The original publication gives running times on par with those of ELAND, and, depending on the settings, even about 20% faster [LLKW08].

Another approach is taken by MAQ. In addition to the mapping itself, MAQ calculates a quality for each alignment, based on which it estimates the probability of the alignment being wrong. Additionally, instead of just discarding less optimal hits like ELAND, these are kept for subsequent analysis. Thus, unlike ELAND, MAQ offers a framework of extended functionality around the mere task of oligo mapping.

## 2.4 Comparison

Despite drastically different functionality, performance is still a main concern. Figure 2.1 reproduces the results of a comparative study undertaken by Anne-Katrin Emde. Although both SOAP and MAQ are much slower than ELAND in all instances, one has to keep in mind that this performance is still orders of magnitude faster than the next best tool, BLAST. The SOAP publication gives a factor of over 300 while still aligning more reads than BLAST.



**Figure 2.1:** Comparison of the running time and memory usage of ELAND, MAQ and SOAP. The input sequence is chromosome 1 of the human genome. Data courtesy to Anne-Katrin Emde. Reproduced with permission.

# ALGORITHM



## 3.1 General description

Since ELAND allows the query reads to contain a small number of errors, exact string matching cannot be used directly here. On the other hand, inexact string matching is extremely slow. ELAND solves the problem in two phases. The first phase consists of a lookup that uses only exact string matching. This yields a number of possible hits of oligo reads  $R$  in the reference sequence  $S$ . In the second phase, these possible hits are verified using an inexact string comparison of the string at the hit position.

Why is this an improvement? If the first phase can be solved efficiently, the verification has been reduced from looking at the whole reference for each read to looking only at possible hit positions, which we expect to be relatively few per read. Before we can solve either of the problems, though, we have to define exactly what we mean by *possible hit*.

**Definition 2.** A possible hit of a pattern  $P$  of length  $m$  is a position  $i$  in the reference string  $S$  if there exists a pair of numbers  $0 < a < b \leq n$  such that  $S_{i+a\dots i+b} = P_{a\dots b}$ . In other words, there is at least a partial exact match of length  $b - a$  between the pattern and the sequence starting at position  $i + a$  if the first letter of the pattern is aligned with the  $i$ th letter of the reference.

**Corollary.** *The longer the partial exact hits, the higher the sensitivity, i.e. the ratio of correct hits to possible hits.*

We call filtering the process of finding such possible hits. We can filter efficiently by building an index data structure (section 3.2) to search for substrings of the patterns. Additionally, we can guarantee that this covers *all* correct hits by applying the pigeon-hole lemma.

**Lemma 1.** *Given a pattern  $P$  that matches a string  $S$  with  $k$  mismatches, the pigeon-hole lemma (applied to pattern matching) states that  $P$  can be divided into  $k + 1$  parts so that at least one substring has to match exactly (with 0 errors).*

*Proof.* Assume that  $P$  is divided into  $k + 1$  parts and that each part matches its counterpart in  $S$  with at least one mismatch. Then simple counting reveals that there are at least  $k + 1$  mismatches between  $P$  and  $S$ .  $\zeta$   $\square$

Now, how to choose the substrings to filter for? In a first step, all oligos are split into four fragments, denoted  $A, B, C, D$ , which are then combined pairwise to yield all possible combinations that still regard the original order (e.g. we don't allow the combination  $BA$ ). This yields the combinations  $AB, BC, CD, AC, BD$  and  $AD$ . This list is complete.

*Proof.* The number of combinations of choosing two out of four, disregarding order, is given by  $\binom{4}{2} = 6$ . In our case, order has to be regarded, but on the other hand, only one of the possible two orders for each pair (e.g.  $AB$  and  $BA$ ) is allowed. Thus, there are six unique solutions,  $AB, BC, CD, AC, BD, AD$ .  $\square$

For the sake of simplicity, it is assumed that  $w$ , the length of the reads, is divisible by 4. If this is not the case, the last few characters of each read are not regarded while filtering, thus restricting the reads to prefixes of length  $w$ .

In the next step, these six substring combinations can be searched in the reference sequence using exact string matching. In order to do this efficiently, an index data structure is maintained.

## 3.2 Index data structure

Indices are data structures for very efficient (sub-linear) text retrieval from a large database. For that reason, they play an important role in computational genomics. A simple and often used index is the suffix array that assigns an index to each suffix of the text, ordered alphabetically [MM90]. Retrieval of the position of a given search text takes  $\mathcal{O}(\log n)$  using a binary search on the sorted list of suffixes.

For the search of fragments of limited length this can be improved even further by storing a directory in addition to the suffix array. For each substring of a fixed length  $q$ , called a  $q$ -gram, the directory stores the position at which the first substring hit can be found in the suffix array. Now, instead of using a binary search, a  $q$ -gram can be looked up in the directory. For small values of  $q$ , this lookup consists of hashing the  $q$ -gram and using the hash as an index in the directory (see section 3.5). This strategy allows a lookup in an order of magnitude of  $\mathcal{O}(q) = \mathcal{O}(1)$  for bounded values of  $q$  which is almost always true because we expect  $q \ll n$ . Furthermore, the memory required is proportional to the length of the text, with a factor (on 32-bit architectures) of 4 bytes per entry required for the suffix array and an additional 4 bytes for each  $q$ -gram in the dictionary.

This data structure is known as the  $q$ -gram index and has the following formal definition.

**Definition 3.** Given a sequence  $S$  of length  $|S| = n$  over an alphabet  $\Sigma$  and a  $q$ -gram, the  $q$ -gram index stores, in alphabetical order, the list  $I_{SA}$  of positions  $i$  of all substrings  $S_{i..i+q}$  (hence the positions of all equal substrings are stored consecutively). Furthermore, it stores a list  $I_{dir}$  that, for every possible permutation  $P$  of  $q$  characters  $\in \Sigma$  stores the index  $i$  of the first occurrence of  $P$  in  $I_{SA}$ . For all  $P$  that are not substrings of  $S$ , it stores the index  $i$  of the first position of the lexicographically next permutation that *is* a substring of  $S$ .

*Note.* This implies the sizes  $|I_{SA}| = n - q + 1$  and  $|I_{dir}| = |\Sigma|^q$ . Note, further, that for a given  $P$  the set of occurrences in  $S$  is specified in the form of a half-open interval  $[b, e[$ , i.e.  $e$  is the first hit no longer belonging to  $P$ . This is convenient for the implementation because it conforms with the C++ and the SEQAN style for defining ranges.

### 3.2.1 Building the index

The index can be build efficiently in three steps by

1. counting the  $q$ -grams,
2. summing over all  $q$ -gram counts, and

3. filling the “suffix array”.

*Note.* While a  $q$ -gram index *can* use a full-blown suffix array this is not necessary, and indeed not possible for gapped shapes (see section 3.3). It is sufficient to build a  $q$ -gram array instead. In our case this is also more efficient. Still, we will denote this array as  $I_{SA}$  throughout the text for the sake of consistency.

### Counting the $q$ -grams

Walking over the text, we calculate the hash of the  $q$ -gram at each position and increment its counter in the directory:

---

#### Algorithm 3.1 Count the $q$ -grams

---

```

1 for  $i \leftarrow 1$  to  $n - q + 1$  do
2    $h \leftarrow \text{HASH}(S[Q_i])$                                 ▷ See definition 4 and section 3.5
3    $I_{dir}[h] \leftarrow I_{dir}[h] + 1$ 

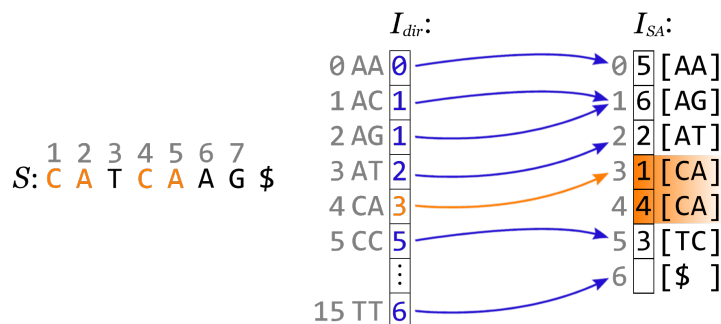
```

---

We call bucket the sequence of fields in the suffix array belonging to one  $q$ -gram. Then the numbers calculated in algorithm 3.1 correspond to the bucket sizes.

### Summing over all $q$ -grams

The first bucket in the suffix array always starts at index 0. The second bucket starts at the index given by  $I_{dir}[0]$  because that’s the number of fields *before* the



**Figure 3.1:** Schematic representation of a 2-gram index over the sequence  $S = \text{“CATCAAG”}$  using the nucleotide alphabet. Array indices are set in gray. The sequence positions are numbered  $1 \dots n$ ; by contrast, the array indices start at 0. Each entry in  $I_{dir}$  corresponds to an index in  $I_{SA}$ . The sequence end is denoted by the dummy character “\$” that does not occur in the sequence itself. A search for the  $q$ -gram “CA” is indicated by the orange colour.

second bucket. To get the start index of the third bucket, we have to add the number of items in the second bucket, i.e.  $I_{dir}[1]$ , and so on for all remaining buckets. Thus, by summing over the sizes of the previous buckets, we get the start index of the current bucket.

Now, this is exactly what the fields in the directory should contain. We therefore find the correct value of  $I_{dir}[i]$  by summing over all values in the fields 0 up to  $i - 1$ :

$$I_{dir}[i] = \sum_{j=0}^{i-1} I_{dir}[j] \quad \text{for all } 0 < i < |\Sigma| \quad (3.1)$$

By noting that this number can be reused to determine the value of  $I_{dir}[i + 1]$  we can efficiently calculate all values of  $I_{dir}$  in one loop. However, keeping in mind that we also still have to fill the suffix array, we can do better by storing the result of the calculation for  $I_{dir}[i]$  in  $I_{dir}[i + 1]$ , thus shifting all results one field down. We do this by introducing an intermediate counter  $\delta_p$  that delays the moment at which the current bucket size is added to the total sum by one iteration.

---

**Algorithm 3.2** Cumulative sum of all  $q$ -grams

---

```

1  $\sigma \leftarrow 0$ 
2  $\delta \leftarrow 0$ 
3  $\delta_p \leftarrow 0$ 
4 for  $i \leftarrow 0$  to  $|I_{dir}| - 1$  do
5    $\sigma \leftarrow \sigma + \delta_p$ 
6    $\delta_p \leftarrow \delta$ 
7    $\delta \leftarrow I_{dir}[i]$ 
8    $I_{dir}[i] \leftarrow \sigma$ 

```

---

**Filling the “suffix array”**

Now that we have the start index positions for all buckets (although they are still at the wrong position), we can begin to fill these buckets. Walking over the sequence, we will visit each  $q$ -gram and write its position into the suffix array. To account for the shift in the directory, we add 1 to the hash before

each lookup. Now the value in the directory can be used as the index in the suffix array at which to write the  $q$ -gram position.

In order to not overwrite existing values in the same bucket, the index found in the directory has to be incremented after each write operation. Thus, the directory serves as an array of current bucket pointers during the construction phase. At the end of construction, each directory field now contains the correct start index for the corresponding bucket.

---

**Algorithm 3.3** Fill the suffix array and adjust the pointer directory

---

```

1 for  $i \leftarrow 1$  to  $n - q + 1$  do
2    $h \leftarrow \text{HASH}(S[Q_i]) + 1$ 
3    $b \leftarrow I_{dir}[h]$ 
4    $I_{SA}[b] \leftarrow i$ 
5    $I_{dir}[h] \leftarrow b + 1$ 

```

---

Clearly, the three algorithms only use elementary operations in their loops (section 3.5 explains why this is true for the HASH function). Thus, the running time for the construction of the  $q$ -gram index is bounded in  $\mathcal{O}(n + |\Sigma|^q)$ .

### 3.2.2 Lookup in the index

Finding  $q$ -grams in the index is simple and requires time proportional to the number of hits. Figure 3.1 indicates how the lookup of CA is done in orange colour.

---

**Algorithm 3.4** Lookup of a  $q$ -gram in the index

---

**Precondition:**  $p$  is a search string of length  $q$

```

1 function LOOKUP( $p$ )
2    $h \leftarrow \text{HASH}(p)$ 
3    $b \leftarrow I_{dir}[h]$ 
4    $e \leftarrow I_{dir}[h + 1]$ 
5   return  $I_{SA}[b, e[$ 

```

---

### 3.3 Gapped $q$ -gram

While simple  $q$ -grams are only defined by their length parameter  $q$ , gapped  $q$ -grams are a generalization that is defined in terms of a shape. The definition as shapes can also be applied to simple  $q$ -grams, e.g. a 4-gram has a shape that could be written as ####. General shapes can have gaps, such as ##-#. When applying the shape to a text GATTACA, this yields the following four  $q$ -grams:

$$\begin{array}{r}
 S = \text{G A T T A C A} \\
 \text{G A - T} \\
 \text{A T - A} \\
 \text{T T - C} \\
 \text{T A - A}
 \end{array}$$

**Definition 4.** A shape  $Q$  is a set of integers including 0.  $q = |Q|$  is called the size of the shape and  $s = \max Q + 1$  its span. Such a shape is called  $(q, s)$ -shape or sometimes simply  $q$ -shape. We call positioned shape the set  $Q_i = \{i + j | j \in Q\}$  for any integer  $i$ . Now, a positioned shape can be applied to the string  $S = S_1 S_2 \dots S_n$ . Let  $Q_i = \{i_1 < i_2 < \dots < i_q\}$ . Then  $S[Q_i] = S_{i_1} S_{i_2} \dots S_{i_q}$  is the (gapped)  $q$ -gram at position  $i$  in  $S$ ,  $0 < i \leq n - s + 1$ .

For the above example, we have  $Q = \{0, 1, 3\}$ . This shape can be positioned at the second letter of  $S$ , yielding  $Q_2 = \{2, 3, 5\}$ , and  $S[Q_2] = \text{ATA}$ .

Gapped  $q$ -grams are described in detail in [BK03], along with a rigorous analysis of their properties in filtering.

### 3.4 Eland shapes

The  $q$ -gram index can be applied to the problem by realizing that the read substring recombinations of  $A$ ,  $B$ ,  $C$  and  $D$  all have an equal length,  $w/2$ . In the following, let  $b = w/4$ .

The read substring combinations defined above are recreated by consecutively building three  $q$ -gram indices. The oligos are chained to build one giant

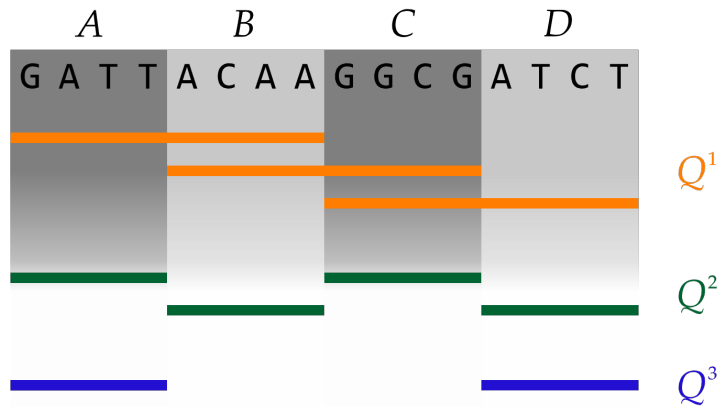


Figure 3.2: The three different shapes used in the algorithm to represent all possible combinations of the read substrings  $A$ ,  $B$ ,  $C$  and  $D$ .

string, and each index is built over the chained oligos (thus, the usual distribution of roles of pattern and sequence is *reversed*). Additionally, not all  $q$ -grams are indexed because we are only interested in those  $q$ -grams that start at the beginning of the substrings  $A$ ,  $B$ ,  $C$  and  $D$ , respectively.

The first index uses an ungapped shape  $Q^1$  of length  $w/2$ . This covers the possible combinations  $AB$ ,  $BC$  and  $CD$ . The second index uses a gapped  $(2b, 3b)$ -shape  $Q^2 = \{0, 1, \dots, b-1, 2b, 2b+1, \dots, 3b-1\}$ . This covers the cases  $AC$  and  $BD$ . The last possible case,  $AD$  is covered by a third index having a gapped  $(2b, 4b)$ -shape  $Q^3 = \{0, 1, \dots, b-1, 3b, 3b+1, \dots, w-1\}$ .

## 3.5 Hashing

The process of mapping a  $q$ -gram to its corresponding index in the  $q$ -gram dictionary is called hashing. As the index in the array is numeric, a way has to be found to generate a numeric key from a  $q$ -gram. For small alphabets, a straightforward method is to assign a serial number to each letter in the alphabet and interpret the  $q$ -gram as a number with  $q$  digits (i.e. we count only the non-gap position), in base  $|\Sigma|$ .

**Definition 5.** The hash  $h_i$  for an ungapped  $q$ -gram  $S[Q_i]$  can be expressed as

a polynomial of degree  $q$  for  $x = |\Sigma|$ :

$$\begin{aligned} h_i &= S_i + S_{i+1} \cdot x + \dots + S_{i+q-1} \cdot x^{q-1} \\ &= \sum_{j=0}^{q-1} S_{i+j} \cdot x^j \end{aligned} \quad (3.2)$$

For gapped  $q$ -grams, only non-gap positions are considered, thus:

$$h_i = \sum_{j=0}^{q-1} S_{i_{j+1}} \cdot x^j, \text{ where } Q_i = \{i_1 < i_2 < \dots < i_q\} \quad (3.3)$$

This number can be calculated by walking once over the  $q$ -gram which takes  $\mathcal{O}(q)$ . In the case at hand, this can be simplified even further because the shapes only consist of up to two contiguous blocks.

Since ELAND deals with very large sequences and big numbers of oligonucleotide reads, strings are represented by bit sequences (called a packed string). Each base requires exactly two bits, having a value of 0, 1, 2 or 3 for A, C, G and T, respectively. Hashing can now be implemented easily by masking the relevant bits from the packed string. For ungapped shapes this is done in two elementary bit operations as shown in equation 3.4.

$$h_i = (p \gg (i - 1)) \wedge (2^q - 1) \quad (3.4)$$

For the gapped shapes  $Q^a$  (with  $a$  being 2 or 3), the two bit chunks have to be merged together, making the computation more difficult (equation 3.5).

$$h_i = (p \gg (i - 1)) \wedge m \vee (((p \gg (i + ab - 1)) \wedge m) \ll b) \quad (3.5)$$

Where  $m = 2^b - 1$ .

Hashing occurs in two places. Once, to build the  $q$ -gram index, all relevant  $q$ -grams in the oligos are hashed and used to build the directory. Usually, every possible position is indexed in the directory. For ELAND, this is not the case as we are only interested in the positioned shapes that represent the six substring combinations listed above. For the equations 3.4 and 3.5 this means that  $i$  only takes the values  $\{1, b, 2b\}$  for  $Q^1$ ,  $\{1, b\}$  for  $Q^2$  and is always 1 for  $Q^3$ .

After the index has been built, filtering is done by considering the  $q$ -grams for each position in the sequence (i.e.  $n - q + 1$   $q$ -grams are considered). Thus,

$n - q + 1$  hashes have to be calculated for the sequence. However, it is actually unnecessary to compute the entire hash for each  $q$ -gram because two consecutive  $q$ -grams share all but two positions (the first and the last) if the shape is ungapped, and all but four positions if the shape contains a gap (the first, the last and the ones immediately before and after the gap).

We can use this fact by storing the previous hash value and using it to calculate the next hash. Thus, the hash  $h_{i+1}$  for the  $q$ -gram  $S[Q_{i+1}^a]$  can be calculated by the equations 3.6 for ungapped shapes (i.e.  $Q^1$ ) and 3.7 for gapped shapes (i.e.  $Q^2$  and  $Q^3$ ).

$$h_{i+1} = \frac{h_i - c_0}{|\Sigma|} + d_0 \cdot |\Sigma|^{q-1}, \text{ and} \quad (3.6)$$

$$h_{i+1} = \frac{h_i - c_0 - c_1 \cdot |\Sigma|^b}{|\Sigma|} + d_0 \cdot |\Sigma|^{q-1} + d_1 \cdot |\Sigma|^{b-1} \quad (3.7)$$

Where:  $c_0$  – the value of  $S_i$ ,

$d_0$  – the value of  $S_{i+q}$ ,

$c_1$  – the value of  $S_{i+b-1}$ ,

$d_1$  – the value of  $S_{i+b}$ .

Since computers are very good at performing integer divisions, these two equations can be further simplified by dropping the subtraction of  $c_0$ . The integer division guarantees that any value smaller than  $|\Sigma|$  is subtracted automatically from the result.

### 3.6 Verification

The previous steps have resulted in a set of possible hit positions. The task is now to verify these possible hits. Verification of a possible hit requires counting the local distance (i.e. the number of errors) between a given oligonucleotide read and the reference sequence at the given position.

Possible errors are substitution errors, insertions and deletions. The latter two create gaps that result in a positional shift between the reference and the read string. Unfortunately, finding correspondences between strings that contain gaps has a running time of  $\mathcal{O}(nm)$ , e.g. using the algorithm of Smith and

Waterman ([SW81]). Finding and counting substitution errors, on the other hand, is simple.

**Definition 6.** We call Hamming distance between two strings  $x$  and  $y$  of equal length  $n$  the minimum number of substitutions necessary to transform  $x$  into  $y$ :

$$\Delta(x, y) = \sum_{\substack{i \in \{1 \dots n\} \\ x_i \neq y_i}} 1 \quad (3.8)$$

**Corollary.** *It follows directly that the Hamming distance can be calculated by iterating over all characters and comparing each position and thus has a running time of  $\mathcal{O}(n)$ .*

It therefore is a reasonable trade-off to consider only substitution errors and accept that some very similar reads are not found if they contain a gap.

Since the algorithm will be implemented using packed strings for the oligos, this problem can be reduced further. Counting differences between two bit strings amounts to taking their exclusive-or and counting all remaining set bits. This can be done efficiently. However, since each character in the packed string requires not one but two bits, we don't count single bit differences but different pairs of bits: There is a mismatch between two characters in  $x$  and  $y$  if and only if their exclusive-or value has one or both bits set, i.e. if the *xor* value is 1, 2 or 3.

---

### Algorithm 3.5 Counting mismatches between two packed DNA strings

---

**Precondition:**  $x$  and  $y$  are packed DNA strings of equal length  $n$

```

1 function DISTANCE( $x, y$ )
2    $z \leftarrow x \oplus y$                                 ▷  $\oplus$ : bitwise exclusive-or
3    $\delta \leftarrow 0$ 
4   for  $i \leftarrow 1$  to  $n$  do
5     if  $z_i \neq 0$  then
6        $\delta \leftarrow \delta + 1$ 
7   return  $\delta$ 

```

---

Of course, algorithm 3.5 has the same runtime as the intuitive algorithm, so why bother? Because we can take advantage of one of two possible ways

to improve the runtime of the loop by applying the same technique as for ordinary bit-counting: bit parallel counting or a lookup table.

### Bit parallel counting

This algorithm, also known as *sideways addition*, appears in the “Software optimization guide for AMD 64 processors” and requires just 12 elementary operations for 32 bits [AMD05]. However, it doesn’t handle counting bit pairs. To amend this, a bit of preprocessing is required. First, two bit masks are defined, the first,  $m_1 = \dots 01010_b$ , masking all even positions and the second,  $m_2 = \dots 10101_b$ , all odd. Next, the following algorithm is applied:

---

**Algorithm 3.6** Preprocessing for the sideways addition of two-bit character strings

---

```

1  $z^1 \leftarrow z \wedge m_1$ 
2  $z^2 \leftarrow z \wedge m_2$ 
3  $z \leftarrow (z^1 \gg 1) \vee z^2$ 

```

---

In that fashion, all bit pairs have been entangled to yield a single bit representing a mismatch, and a 0 bit inserted after it. Now, conventional bit counting by sideways addition can be applied<sup>1</sup>. The algorithm is given in C++ rather than in pseudo code because it uses several assumptions about the size of the integer type that can’t well be conveyed in pseudo code. The code works for words up to 128 bits wide.

---

```

1 template <typename IntType>
2 IntType count_ones(IntType z) {
3     // ... perform preprocessing from algorithm 3.6.
4     IntType const ONES = ~static_cast<IntType>(0);
5     z = z - ((z >> 1) & ONES / 3);
6     z = (z & ONES / 15 * 3) + ((z >> 2) & ONES / 15 * 3);
7     z = (z + (z >> 4)) & ONES / 255 * 15;
8     return static_cast<IntType>(z * (ONES / 255)) >> (sizeof(z) -
9         1) * CHAR_BIT;

```

---

<sup>1</sup>Code modified from [graphics.stanford.edu/seander/bithacks.html#CountBitsSetParallel](http://graphics.stanford.edu/seander/bithacks.html#CountBitsSetParallel)

Listing 3.1: Sideways addition

The preprocessing takes four operations and the above, generalized version of the algorithm takes 23 operations, yielding a total of 27 operations for the whole calculation.

## Lookup table

Instead of relying on bit parallelism one can precompute the values for a reasonable range. In this case, a single byte (i.e. 8 bits) would be reasonable. The calculation then consists of up to eight (for 64 bit words) lookup operations. For each lookup, the byte in question has to be masked and right shifted and the results have to be added. This results in a total of 27 elementary operations, including the lookup. Additionally, the lookup table may have to be loaded from cache if it has been discarded.

```
1 uint32_t count_ones(uint32_t z) {
2     // 'TwoBitMismatches' is the precomputed lookup table.
3     return
4     TwoBitMismatches[ z          & 0xFF] +
5     TwoBitMismatches[(z >> 8)  & 0xFF] +
6     TwoBitMismatches[(z >> 16) & 0xFF] +
7     TwoBitMismatches[ z >> 24];
8 }
```

Listing 3.2: Lookup table

Depending on the machine, the code for 64 bits will look analogously or, on 32 bit machines, will calculate the low word and the high word separately by calling the 32 bit method above. Both methods perform virtually equally good. There might be differences on a different machine, which is why both methods were kept handy.

Finally, one of the above methods has to be used in the verification. Verification simply takes the two strings (assuming both are already converted to packed strings) and compares the difference count to the threshold:

In algorithm 3.7, the threshold value has been set to 2, consistent with our filtering. We therefore allow for two substitution errors.

**Algorithm 3.7** Verification

---

```

1 function VERIFY( $x, y$ )
2    $z \leftarrow x \oplus y$ 
3   return COUNTONES( $z$ )  $\leq 2$ 

```

---

## 3.7 Synthesis

Algorithm 3.8 contains the complete description of the ELAND algorithm.

**Algorithm 3.8** The ELAND algorithm

---

```

1  $H \leftarrow \emptyset$  ▷  $H$  is a list of hits.
2 for all  $\varphi \in (1, 2, 3)$  do
3   FILTER( $\varphi$ )
4 for all  $h \in H$  do
5   if  $\neg$ VERIFY( $P[h_1], S_{h_2 \dots h_2+w}$ ) then Remove  $h$  from  $H$ 
6 Report unique hits in  $H$ 

7 function FILTER( $\varphi$ )
8   Create index  $x$  over  $P$  with shape  $Q^\varphi$ 
9   TRAVERSE( $x, \varphi, S$ )
10   $R \leftarrow$  REVERSECOMPLEMENT( $S$ )
11  TRAVERSE( $x, \varphi, R$ )

12 function TRAVERSE( $x, \varphi, s$ )
13  for  $i \leftarrow 1$  to  $n - q + 1$  do
14     $h \leftarrow$  HASH( $s[Q_i^\varphi]$ )
15     $r \leftarrow x$ .LOOKUP( $h$ )
16     $H \leftarrow H \cup \{(r_1, i - r_2 + 1)\}$ 

```

---

### Annotations

The algorithm requires  $S$ , the reference sequence of length  $m$  and  $P$ , the set of packed oligo reads, all of the same length  $w$ .

Consider a hit without mismatches; such a hit will turn up for each filtering pass and for each combination of oligo substrings. Thus, it will be found and verified *six times*. In order to report this hit only once instead of six times, a dictionary has to be maintained to map all occurrences of the hit to the same item in line 6.

However, because this association is only needed once, most dictionary structures would be too expensive to maintain. The cheapest solution is to insert all items into a regular array that is then sorted, so all equal hits will be adjacent to each other.

The algorithm for the creation of the index has been laid out before. In line 8 it has to be taken care of the restrictions formulated in section 3.4. Additionally, the suffix array has to contain pairs of values to store the complete positioning information: The index is built over a set of oligo reads. Therefore, the suffix array has to contain two information; first, which oligo contains the  $q$ -gram; and second, at which position within the oligo it is contained.

As we have no directional information about the read data, we don't know if the reads belong to the forward strand or to the reverse strand. In order to find all possible associations, we also have to search for reverse complements of all reads, which is done simplest by reversing and complementing the reference sequence in line 10.

We store the position at which the conjectured hit begins within the sequence; therefore, the second index has to be corrected by the offset of the exact match within the oligo in line 16.

# IMPLEMENTATION DETAILS

# IV

## 4.1 Code complexity

So far, the implementation descriptions have been “platform agnostic” that is, they were not bound to any given framework. However, implementing such a program efficiently from scratch would usually be far above the scope of a bachelor thesis: the pure ELAND source code (without counting the surrounding pipeline) has over 4500 lines of code (LOC), the SOAP source code has over 4000 LOC and the MAQ source code has over 7000 LOC.

On the other hand, Bjarne Stroustrup, the inventor of C++, has astutely stated that, “without a good library, most interesting tasks are hard to do in C++; but given a good library, almost any task can be made easy.” [Stro2] Indeed, C++ offers some unique features that make it a very powerful language, given the right set of supporting tools. For bioinformatics, one of these tools is the SEQAN library.

## 4.2 SeqAn features

What makes SEQAN interesting is its mechanism to achieve a very polymorphic, extensible design. Unlike most modern libraries relying heavily on object oriented principles, SEQAN uses classes solely as tags to establish a hierarchy of types, and as records to hold data members. Methods, on the other hand, are dropped in favour of global functions. On first glance, this seems counter-intuitive for a number of reasons: it breaks data encapsulation and it makes it impossible to overwrite methods in subclasses.

The latter is compensated by the mechanism of template subclassing: nested templates are used to model a type hierarchy that would usually require in-

heritance. Now, function overloading can be used to achieve a polymorphism resembling method overwriting, with the important distinction that function overloading is resolved at runtime. In combination with aggressive call inlining, this yields a big performance improvement.

In conclusion, SEQAN provides a very high level of abstraction and adaptability while having *no* negative performance impact whatsoever. Compared to the prevailing trend of buying abstraction with runtime indirection, this is a different approach. To demonstrate just how generic the library really is, [DWRR08] includes the re-implementation of the MUMMER tool that finds maximum unique matches (MUMS) in large sequences.

The comparison shows that the SEQAN implementation is roughly twice as slow as the original tool and consumes twice as much memory. On the other hand, the implementation is much shorter and more concise and, additionally, maintains indexes over multiple sequences, thus being able to find MUMS between different sequences, something MUMMER is unable to do.

### 4.3 Implementation

SEQAN already provides a Shape type. These shapes can be parametrised in two ways, namely at runtime or at compile time. For our purpose, we have implemented a specialisation of the Shape class, called OneGappedShape that can be used to model the shapes  $Q^1$ ,  $Q^2$  and  $Q^3$ :

---

```

1 template <unsigned int Len, unsigned int Pass>
2 struct Q { };
3
4 template <unsigned int Len>
5 struct Q<Len, 1> {
6     typedef seqan::OneGappedShape<Len / 4, 0> Type;
7 };
8
9 template <unsigned int Len>
10 struct Q<Len, 2> {
11     typedef seqan::OneGappedShape<Len / 4> Type;
12 };

```

```
13
14 template <unsigned int Len>
15 struct Q<Len, 3> {
16     typedef seqan::OneGappedShape<Len / 4, Len / 4 * 2> Type;
17 };
```

Listing 4.1: The three ELAND shapes

Notice that the length of the oligonucleotide reads is given as a template parameter. Lengths not divisible by four are automatically truncated. The listing shows that the shapes are notated in a style typical for C++ and SEQAN, namely metafunctions. Metafunctions are a concept that, depending on a template type, yield a different type. This allows polymorphic type aliases, an important technique in SEQAN. The tricky part has been the implementation of the specialized hash functions for these shapes.

Additionally, the routine that constructs the  $q$ -gram index had to be rewritten for our purpose, to prevent the indexing of all contiguous  $q$ -grams in the reads. In our implementation, this was done by creating a specialization `Index<TText, Index_QGram<TShape, Eland> >`. However, this is actually not necessary. It would have been sufficient and slightly simpler to specialize the involved methods for the `OneGappedShape`. This was done by oversight.

For verification, both algorithms proposed in section 3.6 have been implemented. There was no measurable performance difference. The final version uses the lookup table variant because the implementation is simpler (and thus less error-prone).

To find reverse complements, we used a nested `ModifiedString`. As these modifiers act lazily and in-place, such a strategy avoids the creation of expensive copies. On the other hand, it means that the operations necessary for the string modification have to be executed every time we access the string, instead of just once. Therefore, an actual reverse complement copy is created (instead of using the in-place calculation) during the filtering and verification phase. The lazy in-place variant has only been used in the last bookkeeping phase. Here, all mapped reads are written into the output file along with additional information (such as the number of errors).

The verification is handled differently than suggested by the pseudo-code. Instead of first collecting all possible hits and then verifying them, verification

is done immediately for each hit. This was done because the number of possible hits is still much larger than the number of actual hits and the memory consumption had to be reduced.

All in all, the resulting code is smaller than the original code, with less than 1300 LOC. The largest part consists of helper functions for the bit manipulation. Although SEQAN already defines packed strings, these abstractions mainly reduce memory but do not specialize code for hashing and the efficient low-level manipulation of bit patterns.

## 4.4 Restrictions

Like the current version of the original ELAND, the code does not handle IUPAC ambiguity codes. Neither does it handle wildcards (N letters). Additionally, the output file only includes oligo reads that have at least one approximate hit in the database. Oligos without hits are dropped from the output. Therefore, the results produced are different from ELAND's.



## RESULTS

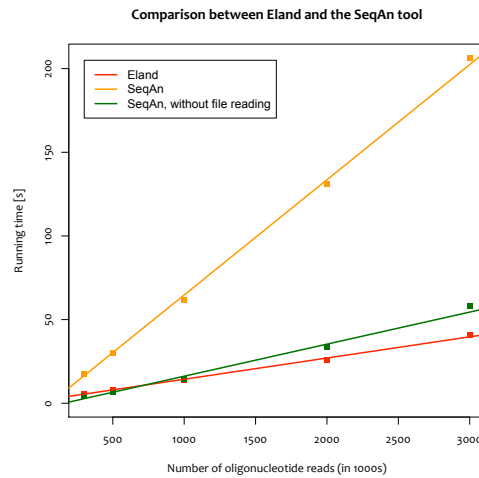
### 5.1 Test setup

For the tests, chromosome 3 of the yeast (*Saccharomyces cerevisiae*) with a length of 316617 bp has been used. The oligonucleotide reads have a length of 24 and were created randomly. The reads are uniformly distributed across the reference sequence, both the forward and the reverse strand with equal probability. The number of errors has been calculated with an average of 1.5 errors and a standard deviation of 1. Although this normal distribution of the number of errors does not reflect the actual read error distribution behaviour, this is not relevant for testing because the algorithm makes no assumptions about the distribution of the errors, other than that most reads will contain less than 3 errors.

For the ELAND test runs, only the actual running time of the tool itself has been measured. The preparation time of the genome using the squashGenome tool from the GERALD tool chain can be neglected.

### 5.2 Data

Figure 5.1 shows the read mapping against chromosome 3 of the yeast. The graph clearly shows that ELAND is faster by more than a factor 4 for larger numbers of reads. However, about 75% of the time is spent reading the input files. If we subtract this time, the SEQAN tool performs much better.



**Figure 5.1:** Runtime comparison between ELAND and the SEQAN tool. The sequence is chromosome 3 of the yeast. Because the largest part of SEQAN’s time is spent loading the sequence and reads into memory, the third data set (in green) indicates pure runtime without the loading delay.

### 5.3 Interpretation

The results, especially the big discrepancy between the total runtime and the runtime without the reading the input files, show ELAND’s biggest advantage. ELAND uses memory-mapped files to prevent having to read all the reads at once. The SEQAN tool, on the other hand, starts by loading all read data. This has two consequences. First, a performance drop, because the SEQAN reading routines are slow. And second, because a lot of data gets written into the main memory, access becomes slower. For many reads, the tool spends a lot of runtime in idle mode, waiting for the hard drive to load swapped memory.

Still, the results also show the advantages of using SEQAN to aid the implementation of a bioinformatical tool. The library offers a rich tool box that can be heavily customized to meet the specific demands of the task while retaining high flexibility and excellent performance, for the most part. It is especially notable that while file access is slow, the other parts of the algorithm perform very efficiently. Although it has to be kept in mind that the ELAND tool currently does more work (see section 4.4) in the same amount of time, this doesn’t negate the overall good runtime.

## 5.4 Outlook and future work

The results clearly show some weaknesses in our implementation. Foremost, the file reading routine that costs about 75% of the overall performance could be vastly improved. One strategy would be to drop the file reading altogether in favour of the `SEQAN External` string specialization that allows the accessing of a string located on secondary memory. This should be investigated first, as it promises a good performance yield at relatively few expenses.

This could also avoid some (or even all) of the problems connected with swapping. However, since secondary memory access is very slow, this relies on the existence of a prefetch mechanism in place.

The next step is to extend the tool. Interesting points are the maximum length of the reads, although this is hard to extend due to memory limitations, if we want to continue using a  $q$ -gram directory, and the number of errors that are allowed, which would require adapting the shapes. Perhaps most important would be the extension to allow gaps in the reads. This would mean adapting the verification to handle edit distances. At the moment, the algorithm doesn't make any use of multiprocessor architectures. It therefore doesn't scale well on future computers. It would be desirable to parallelize the work done by the algorithm.

On a completely different note, the project has also allowed an insight into the current development of C++. Code complexity is a major challenge in programming. C++ helps to reduce complexity by facilitating the usage of efficient libraries. On the other hand, C++ has a complex syntax that has been often criticized [WC00]. While some of this complexity stems from its legacy C support, the flexible templates, resulting in ambiguous semantics for the compiler, make the syntax especially difficult. A substantial portion of the line count of the implementation consists of boilerplate template code. In this regards, C++ is still very actively developed and it will be interesting to see how C++0x's template syntax will reduce code complexity and further aid the development of high-performance libraries [The08].

# ACKNOWLEDGEMENT



For their great help and many interesting discussions I'd like to thank the whole algorithmic bioinformatics work group and all the graduate students working in office 009 alongside me, as well as Anne-Katrin Emde for the performance data comparing different read mapping tools.

I would especially like to thank Andreas Döring and David Weese for their support and incentive, for their patience during the sometimes tedious debugging sessions and for their ideas on optimization. Perhaps most importantly, for setting my priorities straight when I risked to lose myself in details.

I would also like to thank my supervisors Prof. Dr. Knut Reinert and Dr. Gunnar Klau for giving me the opportunity to work on this fascinating subject and generally for their efforts in teaching.

## COLOPHON

# B

This document was created using  $\text{\LaTeX}$  2 $\epsilon$  and Bib $\text{\TeX}$  and edited in the MAC-VIM environment with the  $\text{\LaTeX}$ -vim plug-in. The typesettings software used the X $\text{\TeX}$  distribution and the *fontspec* package. The text is set in Hoefler Text and Candara. The source code is set in Consolas.

The animal depicted on the cover is a *common eland* (*taurotragus oryx*). The original photograph was taken from [flickr.com/photos/paulmullett/505797443/](https://www.flickr.com/photos/paulmullett/505797443/) and is used and modified with permission under the Creative Commons “by-nc” license. The author of the original picture is Paul Mullett.

# BIBLIOGRAPHY

- [AMD05] AMD Advanced Micro Devices Inc., *Software optimization guide for AMD Athlon™ 64 and Opteron™ processors*, 2005.
- [BBC<sup>+</sup>05] Simon T. Bennett, Colin Barnes, Anthony Cox, Lisa Davies, and Clive Brown, *Toward the 1,000 dollars human genome*, *Pharmacogenomics* **6** (2005), no. 4, 373–382 (eng).
- [BK03] S. Burkhardt and J. Kärkkäinen, *Better Filtering with Gapped  $q$ -Grams*, *Fundamenta Informaticae* **56** (2003), no. 1, 51–70.
- [CMP03] Francis S. Collins, Michael Morgan, and Aristides Patrinos, *The human genome project: lessons from large-scale biology*, *Science* **300** (2003), no. 5617, 286–290.
- [DWRR08] Andreas Döring, David Weese, Tobias Rausch, and Knut Reinert, *SeqAn – An efficient, generic C++ library for sequence analysis*, *BMC Bioinformatics* **9** (2008), no. 1, 11.
- [LLKW08] Ruiqiang Li, Yingrui Li, Karsten Kristiansen, and Jun Wang, *SOAP: short oligonucleotide alignment program*, *Bioinformatics* (2008).
- [LSV07] Samuel Levy, Granger Sutton, and J. Craig Venter, *The diploid genome sequence of an individual human*, *PLoS Biol* **5** (2007), no. 10, e254 (eng).
- [MM90] U. Manber and G. Myers, *Suffix arrays: a new method for on-line string searches*, *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms* (1990), 319–327.
- [Shao7] Catherine Shaffer, *Next-generation sequencing outpaces expectations*, *Nature Biotechnology* **25** (2007), 149.

- [SMVC04] Jay Shendure, Robi D. Mitra, Chris Varma, and George M. Church, *Advanced sequencing technologies: methods and goals*, Nat Rev Genet **5** (2004), no. 5, 335–344.
- [Stro2] Bjarne Stroustrup, *C++ programming styles and libraries*, January 2002.
- [Stuo2] Caroline Stupnicka, *Solexa announces progress in its single molecule array technology at bioarrays europe conference*, 10 2002.
- [SW81] T. F. Smith and M. S. Waterman, *Identification of common molecular subsequences*, J Mol Biol **147** (1981), no. 1, 195–197 (eng).
- [Theo8] The C++ Standards Committee, *Working Draft, Standard for Programming Language C++*, Tech. report, ISO, 05 2008.
- [VIR<sup>+</sup>01] J. Craig Venter, . . . , Knut Reinert, et al., *The sequence of the human genome*, Science **291** (2001), no. 5507, 1304–1351.
- [WC00] Ben Werther and Damian Conway, *The Design and Implementation of SPECS: An alternative C++ syntax*, Tech. report, Department of Computer Science, Monash University, 2000.
- [WLTBL02] Robert H Waterston, Kerstin Lindblad-Toh, Ewan Birney, and Eric S Lander, *Initial sequencing and comparative analysis of the mouse genome*, Nature **420** (2002), no. 6915, 520–562 (eng).