# Local Aligner for Massive Biological Data

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree

Master of Science in Bioinformatics

Hannes Hauswedell

December 7, 2013

| | |
|---|---|
| **First Supervisor:** | Prof. Dr. Knut Reinert |
| **Second Supervisor:** | Dr. Bernhard Renard |
| **Advisor:** | Jochen Singer |

I hereby affirm in lieu of an oath that I have produced this work all by myself. Ideas taken directly or indirectly from other sources are marked as such. This work has not been shown to any other board of examiners so far and has not been published yet.

I am fully aware of the legal consequences of making a false affirmation.

| | |
|---|---|
| Place/Date | Signature |

## Abstract

**Motivation** Next-generation sequencing technologies produce unprecedented amounts of data, leading to completely new research fields. One of these is metagenomics, the study of large-size DNA samples containing diverse organisms. A key problem in metagenomics is functionally and taxonomically classifying the sequenced DNA, to which end the well known BLAST program is usually used. But BLAST has dramatic resource requirements at metagenomic scales of data, imposing a high financial or technical burden on the researcher. Multiple attempts have been made to overcome these limitations and present a viable alternative to BLAST, but as of yet, they have not gained widespread adoption.

**Results** In this work we present Lambda, our own alternative for BLAST in the context of sequence classification. In our tests Lambda is among the best tools at reproducing BLAST's results and is faster than all existing solutions at comparable levels of sensitivity.

# Contents

# List of Figures

# List of Tables

# 1 Introduction

A great part of all function that all known forms of life[1] exhibit, originates in *genes*. Genes are stretches of deoxyribonucleic acids (DNA) and in rare cases ribonucleic acids (RNA) that encode for proteins, which in turn are the main active compounds in the cell. The study of the variation, function, evolution and hereditary properties of genes is referred to as *genetics*.

The sum of all hereditary information, which an organism contains is known as the *genome* of an organism. This includes all the genes, but a genome contains varying proportions of nucleic acid sequences that are not protein coding, as well. It can also be organized in different more complex 3D structures and is subject to biochemical modification that is not represented in the sequence, but passed to offspring nevertheless. The study of genomes, their properties and interactions, is called *genomics*. Genetics and genomics in particular play an ever more important role in modern biological and medical research, with many diseases linked to genetic factors and personalized medicine on the rise.

Taking genetic analysis to an even more abstract level is the field of *metagenomics*. In metagenomics genetic material is collected from a large quantity of genomes and organisms at once, either from environmental samples or from more complex host organisms. The focus is often on the study of microbial biodiversity in relation to ecological factors (in the case of environmental samples) or health (in the case of sampling e.g. the human gut). But implications go beyond this and contribute to a broader, more general understanding of speciation and the development of life (Marco, 2011).

## DNA-sequencing

DNA molecules are built as long chains of four different compounds called (nucleo-)bases,[2], and are denoted by the letters (A, C, G, T) respectively. The process of determing this sequence of characters is referred to as *sequencing* and is *the* prerequisite for most of the aforementioned genetic research.

Sequencing is not trivial and there are very different biotechnological solutions to solving this problem. Sanger sequencing (Sanger and Coulson, 1975; Sanger et al., 1977) was the most widely used technology until the beginning of the 21st century and is among other projects responsible for making the sequencing of first human genome possible. Since the beginning of the 21st century, new technologies have become widely popular, subsumed under the terms Second-generation sequencing, Next-generation sequencing (NGS) or high-throughput sequencing. Of these technologies two popular ones are "pyrosequencing" by 454 (Roche)[3] and "sequencing by

[1]– including viruses or in addition to viruses, depending on whether they are classified as life –

[2]In fact DNA usually exists in the form of two complementary sequences bound by hydrogen-bonds ("double-strand DNA")

[3]`http://www.454.com/`

synthesis" by Illumina[4]. There are other popular approaches, but these two serve as good representatives.

While the setup of the methods is not important to this work, their properties are. A main difference between technologies in genreal, and the biggest difference to traditional Sanger sequencing is the price. While the cost has dropped for Sanger sequencing, from $\sim \$750,000$ per million bases in 1990 to $\sim \$2,400$ in 2012, the gap to NGS technologies is still wide, with 454 sequencing costing \$10 per million bases and Illumina sequencing less than 10¢ per million bases (Liu et al., 2012). Only through this giant drop in prices has genomic research, and especially metagenomic research become feasible.

But other features of sequencing technologies are also important, among them the length of the produced *reads*, the expected accuracy of the predicted sequences and the time needed to produce that data. No sequencing technology to date can reproduce a long genomic sequence in itself, everyone outputs only subsequences, so called reads. Sanger sequencing still produces comparatively long reads (up to and over 1000 bases) and is still used because of this feature. Of the NGS technologies, 454 reads are $\sim 700$ bases long and Illumina reads $50-150$ bases. Accuracy of Sanger sequences is $> 99.99\%$, while 454's accuracy is $\sim 99.9\%$ and Illumina's $\sim 98\%$.

## Bioinformatics

Analyzing the vast amounts of data that are produced by NGS technologies is where bioinformatics come into play. Although it is true that computers have become a lot faster in the last twenty years, this development not nearly rivals the leaps in which sequencing technology has developed, making it more and more difficult to scale data analysis with the available data. Furthermore, certain problems have become more difficult to solve with new sequencing data because it has different attributes than before. An example of this is *sequence assembly*: As previously explained, sequencing machines output only fragments of the original sequence, and for many types of analysis the original sequence is necessary in order to be reconstructed from the reads. This is an algorithmically complex task, which is much more difficult to solve with many shorter reads, than with fewer longer reads.

In metagenomics, sequence assembly also plays a role, but in this work we will focus on the field of *sequence classification*. Classification means assigning a read to a known, annotated and usually protein-coding sequence, to infer its function and/or its decent, i.e. to identify what it might do and where it comes from or at least what its closest relative is. Bazinet and Cummings (2012) give an overview of the various programs that have been developed to solve this problem. Of the similarity based approaches they compare, eight out of nine use BLAST in their pipeline and for the combined approaches three out of five.

BLAST (Altschul et al., 1997) is a local alignment search tool (more on this later) and one of the most popular bioinformatic tools to date. Bazinet and Cummings (2012) find out in their study that "*[the] BLAST step completely dominates the runtime for alignment-based methods*". For the two programs with the highest precision in their comparison, CARMA (Krause et al., 2008; Gerlach and Stoye, 2011) and MEGAN (Huson et al., 2007), the BLAST step actually made up 96.40% and 99.97% of the run-time. Another metagenomic study (Mackelprang et al., 2011) states that for their BLAST computations they required 800,000 CPU hours at a super computer center.

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

Obviously this is a place in dire need of improvement and a fundamental bottleneck in terms of required time, computing power and financial resources. Some tools have seeked to replace BLAST, with limited success, among them BLAT (Kent, 2002) and the more recent releases of UBlast (Edgar, 2010), RAPSearch (Ye et al., 2011; Zhao et al., 2012) and PAUDA (Huson and Xie, 2013). The latter three all claim to be magnitudes faster than BLAST and at least notably faster than BLAT – at the expense of some degrees of sensitivity. All contain specific algorithmic optimizations for searches against protein databases,[5] most notably the use of reduced alphabets, a technique that we will come across again later. While UBlast and RAPSearch seem to be designed as general replacements for BLAST-modes that contain protein searches, PAUDA is even more specifically designed for metagenomic tasks.

While arguably there are non-technical reasons that these programs have not yet replaced BLAST in many situations, we do believe that better technical solutions will find acceptance over time, simply because the performance deficits of BLAST are so blatant, a fact that will become even more apparent with growing amounts of sequence data becoming available.

## Our contribution

We decided to study the theoretical background of local alignment searches and comparatively analyze existing approaches, as well as modern trends in high-performance computing and optimization strategies. In order to further improve on the situation, both, in regards to speed and sensitivity, we will combine the best of existing algorithms in an efficient implementation of our own.

In the upcoming chapter, we will cover the fundamental background of alignments, search algorithms and related data structures. We will also compare the previously mentioned implementations and introduce parallelization as the central paradigm to efficient problem solving on modern computers.

Then, in chapter 3, we will elaborate on our own solution, which is **Lambda**, the **L**ocal **a**ligner for **m**assive **b**iological **d**at**a**. We will cover the techniques that we chose to be part of it and give details on the implementation. The means for comparing our solution to existing programs is described in chapter 4, together with the results of this comparison.

Finally, the overall outcome of this work is discused in chapter 5, with the quintessence being that we have developed a tool specifically geared for sequence classification, viable for different types of read data and superior to the three alternatives discussed in nearly all situations investigated. It combines modern algorithms with a high-performance implementation on top of the SeqAn-library (Döring et al., 2008) that already offers the prospect of further improvements and an even higher potential of the software.

[5] BLAST and BLAT on the other hand are more general purpose tools, which are optimized for pure DNA searches, as well.

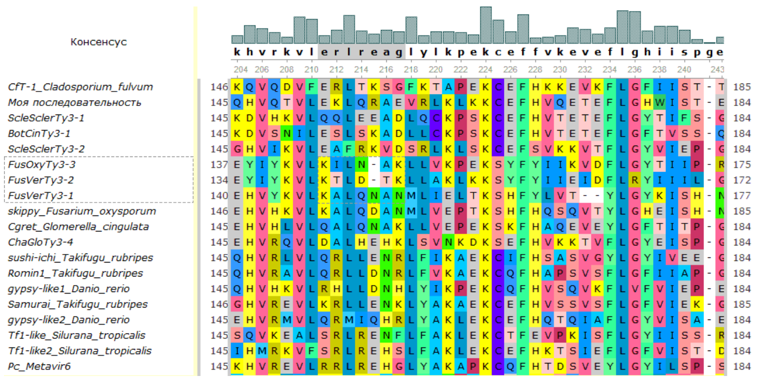
# 2 Research Context

As we illustrated in the introduction, the development of new algorithms and the efficient implementation of existing ones is of paramount importance if the computational challenges of next generation sequencing are to be met. Since the initial publication of BLAST many novel algorithms and data structures were developed, new technologies have become available and computing resources, most importantly main memory, have dropped magnitudes in price. In this chapter we will cover several of these developments and introduce some of the programs that utilize them.

Programs that search for local alignments usually perform their search in two steps, (1) identification of candidate regions for hits (Sec.2.2), and (2) verification of these candidate regions (Sec.2.3). Even if this distinction is not as clear or intended in every case, it makes classification and analysis easier, so we will apply the scheme to all software discussed in this work. Before elaborating on algorithms and software, we will introduce the basic terms and concepts in sequence alignment, popular data structures and statistical methods of assessment.

## 2.1 Sequence alignment background

### 2.1.1 Alignments, distances and scores



```
Консенсус                          k h v r k v l e r l r e a g l y l k p e k c e f f v k e v e f l g h i i s p g e
                                    204 206 208 210 212 214 216 218 220 222 224 226 228 230 232 234 236 238 240   243
CfT-1_Cladosporium_fulvum      146 K Q V Q D V F E R L T K S G F K T A P E K C E F H K K E V K F L G F I I S T - T 185
Моя последовательность         145 Q H V Q T V L E K L Q R A E V R L K L K K C E F H V Q E T E F L G H W I S T - E 184
ScleSclerTy3-1                 145 K D V H K V L Q Q L E E A D L Q C K P S K C E F H V T E T E F L G Y T I F S - G 184
BotCinTy3-1                    145 K D V S N I L E S L S K A D L L C K P S K C E F H V T E T E F L G F T V S S - Q 184
ScleSclerTy3-2                 145 G H V I K V L E A F R K V D S R L K L S K C E F S V K K V T F L G Y V I E P - G 184
FusOxyTy3-3                    137 E Y I Y K V L K I L N - A K L L V K P E K S Y F Y I I K V D F L G Y T I I P - R 175
FusVerTy3-2                    134 E Y I Y K V L K T L D - T K L L A K L K K S Y F Y I I E I D F L R Y I I I L - G 172
FusVerTy3-1                    140 E H V Y K V L K A L Q N A N M L I E L T K S H F Y L V T - - Y L G Y K I S H - N 177
skippy_Fusarium_oxysporum      146 E H V H K V L K A L Q D A N M L V E P T K S H F H Q S Q V T Y L G H E I S H - N 185
Cgret_Glomerella_cingulata     145 E H V H L V L Q A L Q N A K L L V E P E K S K F H A Q E V E Y L G F T I T P - G 184
ChaGloTy3-4                    145 E H V R Q V L D A L H E H K L S V N K D K S E F H V K K T V F L G Y E I S P - G 184
sushi-ichi_Takifugu_rubripes   145 Q H V R L V L Q R L L E N R L F I K A E K C I F H S A S V G Y L G Y I V E E - G 184
Romin1_Takifugu_rubripes       145 Q H V R A V L Q R L L D N R L F V K A E K C Q F H A P S V S F L G F I I A P - G 184
gypsy-like1_Danio_rerio        145 Q H V H K V L R H L L D N H L Y I K P E K C Q F H V S Q V K F L V F V I E P - E 184
Samurai_Takifugu_rubripes      146 G H V R E V L K R L L E N K L Y A K A E K C E F H V S S V S F L G F V I E K - G 185
gypsy-like2_Danio_rerio        145 E H V R M V L Q R M I Q H R L Y A K L E K C E F H Q T Q I A F L G Y V I S A - E 184
Tf1-like_Silurana_tropicalis   145 S Q V K E A L S R L R E N F L F A K L E K C T F E V P K I S F L G Y I I S S - R 184
Tf1-like2_Silurana_tropicalis  145 I H M R K V F S R L R E H S L F A K L E K C E F H K T S I E F L G F V I S T - D 184
Pc_Metavir6                    145 K H V R E V L R R L R E H G L Y A K A P K C Q F H T D S V E Y L G Y I L S P - S 184
```

***Figure 2.1:*** *Multiple sequence alignment of amino acid sequences*

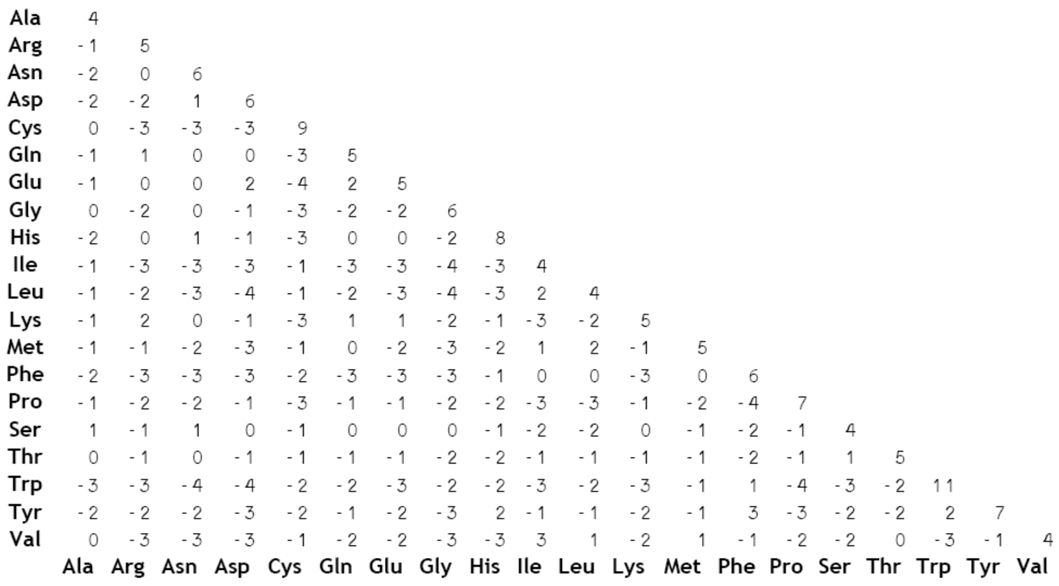
Clipped screenshot of Unipro UGENE (Okonechnikov et al., 2012)

In bioinformatics we frequently deal with biological *sequences*, i.e. strings of characters, which represent different biological compounds. Sequences include strings of DNA, RNA or amino acid, the central players in genomics and proteomics.

In this work we use the term *query sequences* to refer to the sequences we are searching *with*, i.e. the reads of sequencing run. The set of known and annotated sequences which we are searching *in* is referred to as the *database*, and individual sequences of the database are known as *subject sequences* (in other contexts also *target sequences*).

**Alignments** Two (or more) sequences can be arranged in a manner, where identical or similar characters are grouped in columns, while preserving the order of characters in each sequence. We call this kind of arrangement an *alignment*. An alignment indicates the degree of functional, evolutionary and/or structural relatedness, where high relatedness usually implies *homology*, i.e. common biological descent. An arbitrary number of sequences can be aligned and we speak of *multiple sequence alignment*, when more than two are involved (Fig. 2.1). In this work, however, we will focus entirely on algorithms and applications dealing with two sequences at a time, although much is applicable to the general case.

**Distances** There are different kinds of alignments and different ways of measuring relatedness or similarity. Especially for sequences in DNA or RNA alphabet, a common metric is the *hamming distance* (Hamming, 1950), which is the absolute number of character substitutions necessary to produce one sequence from the other. In this metric, sequences are always aligned character by character and each position is either a match or mismatch. Another popular metric is the *edit distance* or Levenshtein distance (Levenshtein, 1966), which considers insertions and deletions of characters (together often abbreviated as *indels*) in addition to substitutions. To represent these an additional character is introduced into the alignment, the gap character (often denoted by '-'), which represents a deletion at the position (or an insertion at the position in the other sequence). These distance metrics imply that single-character substitutions and indels (in the case of edit distance) are the core of natural evolution on sequence level, and thus by counting these evolutionary events, evolutionary distance can be derived. The applicability of the hamming distance over the edit distance is based on the assumption that insertions and deletions are less frequent than substitutions and the possibility of encountering them – especially in short sequences – is low.



| | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Ala** | 4 | | | | | | | | | | | | | | | | | | | |
| **Arg** | -1 | 5 | | | | | | | | | | | | | | | | | | |
| **Asn** | -2 | 0 | 6 | | | | | | | | | | | | | | | | | |
| **Asp** | -2 | -2 | 1 | 6 | | | | | | | | | | | | | | | | |
| **Cys** | 0 | -3 | -3 | -3 | 9 | | | | | | | | | | | | | | | |
| **Gln** | -1 | 1 | 0 | 0 | -3 | 5 | | | | | | | | | | | | | | |
| **Glu** | -1 | 0 | 0 | 2 | -4 | 2 | 5 | | | | | | | | | | | | | |
| **Gly** | 0 | -2 | 0 | -1 | -3 | -2 | -2 | 6 | | | | | | | | | | | | |
| **His** | -2 | 0 | 1 | -1 | -3 | 0 | 0 | -2 | 8 | | | | | | | | | | | |
| **Ile** | -1 | -3 | -3 | -3 | -1 | -3 | -3 | -4 | -3 | 4 | | | | | | | | | | |
| **Leu** | -1 | -2 | -3 | -4 | -1 | -2 | -3 | -4 | -3 | 2 | 4 | | | | | | | | | |
| **Lys** | -1 | 2 | 0 | -1 | -3 | 1 | 1 | -2 | -1 | -3 | -2 | 5 | | | | | | | | |
| **Met** | -1 | -1 | -2 | -3 | -1 | 0 | -2 | -3 | -2 | 1 | 2 | -1 | 5 | | | | | | | |
| **Phe** | -2 | -3 | -3 | -3 | -2 | -3 | -3 | -3 | -1 | 0 | 0 | -3 | 0 | 6 | | | | | | |
| **Pro** | -1 | -2 | -2 | -1 | -3 | -1 | -1 | -2 | -2 | -3 | -3 | -1 | -2 | -4 | 7 | | | | | |
| **Ser** | 1 | -1 | 1 | 0 | -1 | 0 | 0 | 0 | -1 | -2 | -2 | 0 | -1 | -2 | -1 | 4 | | | | |
| **Thr** | 0 | -1 | 0 | -1 | -1 | -1 | -1 | -2 | -2 | -1 | -1 | -1 | -1 | -2 | -1 | 1 | 5 | | | |
| **Trp** | -3 | -3 | -4 | -4 | -2 | -2 | -3 | -2 | -2 | -3 | -2 | -3 | -1 | 1 | -4 | -3 | -2 | 11 | | |
| **Tyr** | -2 | -2 | -2 | -3 | -2 | -1 | -2 | -3 | 2 | -1 | -1 | -2 | -1 | 3 | -3 | -2 | -2 | 2 | 7 | |
| **Val** | 0 | -3 | -3 | -3 | -1 | -2 | -2 | -3 | -3 | 3 | 1 | -2 | 1 | -1 | -2 | -2 | 0 | -3 | -1 | 4 |
| | **Ala** | **Arg** | **Asn** | **Asp** | **Cys** | **Gln** | **Glu** | **Gly** | **His** | **Ile** | **Leu** | **Lys** | **Met** | **Phe** | **Pro** | **Ser** | **Thr** | **Trp** | **Tyr** | **Val** |

***Figure 2.2:*** *Blosum62*

**Scores** For protein sequences the alphabet is much larger (≥ 20 vs. 4 – 5) and it makes sense to distinguish substitution events, since some amino acids are more akin than others. This is done by the means of *scoring matrices* that define a score[1] for all pairs of amino acids. Popular matrices include PAM (Dayhoff et al., 1978) and Blosum (Henikoff and Henikoff, 1992), both of which have versions with different denominating numbers for different expected levels of relatedness, e.g. PAM250 and Blosum62(Fig. 2.2). For PAM matrices a lower number suggests a higher expected relatedness, for Blosum matrices it is the other way around.

**Gap costs** As noted above, these scoring schemes rely on evolutionary events effecting one character at a time. While this is widely accepted for substitutions (Barreiro et al., 2008), it is a poorer model for insertions and deletions, as these can have more different biochemical causes that usually effect more bases (Gusfield, 1997). This is especially true for indels in protein-coding nucleic acid sequences, where single nucleotide indels are more strongly selected against, because they will introduce a shift of the translation frame (see Sec. 3.2 for details on translation). To remedy this lack in the model, *affine gap costs* have been introduced which do not assign the same score to individual alignment positions that contain a gap. Instead a penalty is applied that depends on the length of the contiguous stretch of gap characters. Among those *affine linear* functions assign each position the same penalty, but additionally impose a gap open penalty once for the entire stretch of gap positions (independent of length). These are the most widely used (Vingron and Waterman, 1994); they can be implemented more efficiently than others (see also Sec. 2.1.3).

### 2.1.2 Alignment statistics

The scoring metrics previously discussed have major deficiencies, when dealing with (local) alignments alone, because they do not indicate whether a score received is extra-ordinary, or could likely have been produced by chance. Therefore further measures are required to assert the statistical significance of results.

The most widely accepted statistical model for this purpose was developed by Karlin and Altschul (1990); later to be improved by Altschul and Gish (1996); Altschul et al. (1997, 2001). It is still used today, all programs discussed in this work use scoring methods based on this approach. The main outcome of this research – from the view of the user – is the definition of the *e-value* and the *bit-score*, which we will encounter more frequently in this work. Their definitions in respect to our use case are as follows:

$$\text{e-value} = K * m * n * exp(-\lambda * S) \tag{2.1}$$

$$\text{bit-score} = \frac{\lambda * S - lnK}{ln2} \tag{2.2}$$

where $m$ is the length of the query sequence, $n$ is the sum of the lengths of subject sequences and $S$ is the *raw score* attained by one of the methods described previously. The only difference between e-value and bit-score, is that the e-value is normalized for sequence lengths and the bit-score is not. $K$ and $\lambda$ are parameters of the distribution that is to be expected of the results.[2] Both constants depend on the scoring scheme used and must be precomputed empirically for most schemes, especially when dealing with gapped alignments (Altschul and Gish, 1996). The sequence lengths undergo an edge effect correction, but the specifics are not important to this work.

[1] As is the case in other contexts, score is defined as the counterpart to distance, i.e. a high score equals a low distance and vice versa. Objective functions either minimize distances or maximize scores, the problem has been proven to be identical (Sellers, 1974).

[2] Since we are dealing with maximally scored hits, these follow an extreme value distribution, with these parameters (Karlin and Altschul, 1990).

### 2.1.3 Alignment scopes & computation

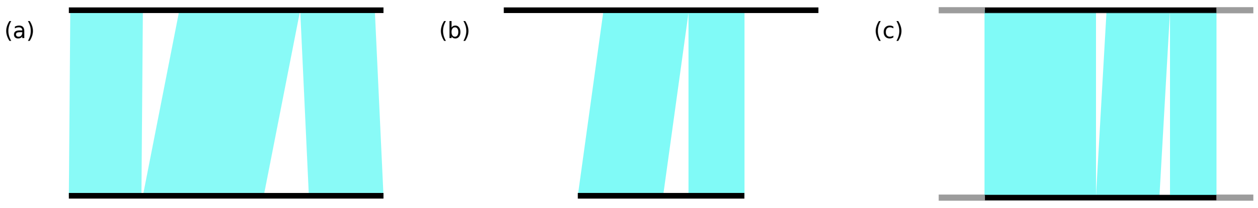


***Figure 2.3:*** *Global (a), semi-global (b) and local (c) alignments, with gaps*

Independent of the scoring scheme used, different algorithms can be used to produce alignments. But before choosing the method of computation it is important to decide on the scope of the alignment, i.e. whether one wants all characters of both sequences to be aligned, or only parts of either. When two entire sequences are being aligned, we speak of *global alignments*; if arbitrary small parts of either are aligned we speak of *local alignments*; and if we require one (usually short) sequence to map completely against a subsequence of a bigger one, we speak of *semi-global alignments* (see Fig.2.3).

Depending on the scope an appropriate algorithm can be used, most of which work by *dynamic programming*, often abbreviated as *dp*. Dynamic programming is a concept for efficiently solving problems that can be reduced to overlapping sub-problems (Dasgupta et al., 2006, p.169ff.). The dynamic programming algorithm which serves as a basis for computing all manner of sequence alignments is the Needleman–Wunsch algorithm (Needleman and Wunsch, 1970), formalized in the now popular form of matrix recurrences by Sellers (1974) and Waterman et al. (1976).

The most widely used version of the algorithm, the one for linear (non-affine) gap costs can compute the optimal global alignment between two sequences in $O(m*n)$ time and space, where $n$ and $m$ are the sequence lengths. It works by constructing a matrix between the two sequences and computing the cells as shown in Fig. 2.4 (where $c$ is the gap cost and $s(a_i, b_j)$ the score between the $i$-th character in sequence $a$ and $j$-th character in sequence $b$).

$$
\begin{aligned}
M(0,0) &= && 0 \\
M(i,0) &= && \text{i*c} && \boxed{0},\ 1 \le i \le m \\
M(0,j) &= && \text{j*c} && \boxed{0},\ 1 \le j \le n \\
M(i,j) &= && \max \begin{cases} M(i-1,j-1) + \ s(a_i,b_j) & \text{match/mismatch} \\ M(i-1,j) + c & \text{deletion} \\ M(i,j-1) + c & \text{insertion} \\ 0 & \text{not in ali.} \end{cases} && ,\ 1 \le i \le m, 1 \le j \le n
\end{aligned}
$$

***Figure 2.4:*** *Alignment algorithm (matrix computation)*
variations for global alignments and local alignments highlighted.

After the matrix is fully computed, the path from $M(m,n)$ to $M(0,0)$ is *backtracked* to produce the actual alignment. This step can be sped up, if a traceback matrix is constructed alongside the main matrix, with information on what case the maximum operation yielded for every cell. A version of the algorithm that has only linear space requirements was developed by Hirschberg

(1975). It achieves this property by computing the matrix column (or row) wise and keeping at most the last and the current column (or row) in memory.

As previously noted, the run-time of $O(m * n)$ is only true for non-affine gap costs; if affine gap costs are used running time becomes $O(max(m,n)^3)$, i.e. cubic. Gotoh (1981) improved this by extending the algorithm to compute two auxiliary matrices. This makes it possible to compute the best global alignment with affine linear gap costs in square time.

A method to further optimize running time is computing only a *band* of size $b$ around the main diagonal of the matrix (Chao et al., 1992). This is based on the assumption that the optimal alignment will likely not contain too many gaps and therefore be inside this band. It effectively reduces running time to $O(min(m,n) * b)$ and since $b$ is constant this can be considered linear. It should, however, be noted that this method can produce different (inferior) results than the previous ones, if it cannot be guaranteed that the optimal alignment is indeed inside the band.

Up until this point we have only considered global alignments, but the construction of local alignments happens in a very similar fashion (Smith and Waterman, 1981). The matrix computation has to be adapted, as shown in Fig. 2.4 and backtracking starts at the cell with the best score in the matrix instead of $M(m,n)$; it ends after encountering the first cell with value 0. All other improvements and modifications to the algorithm can also be applied. Semi-global alignments can be computed by only implementing the changes for a local alignment in one dimension of the matrix.

### 2.1.4 Data structures

Not inherently connected to alignments, but of great importance to many of the approaches subsequently discussed are modern data structures. They play a key role in storing, accessing and retrieving the sequence strings or substrings thereof. This section briefly introduces some that we deem necessary for understanding the applications and algorithms that will follow.

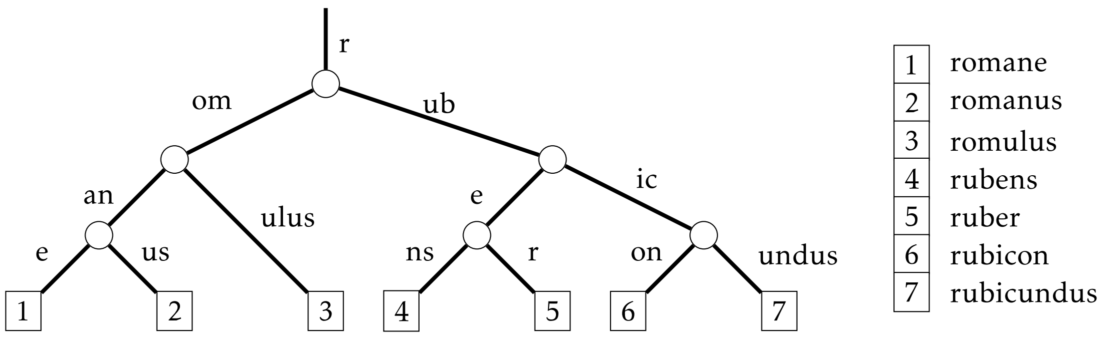


***Figure 2.5:*** *Example of a radix tree*

**Trie** A trie or prefix tree is a tree data structure that represents a set of lexicographically ordered strings. Each edge is labeled with a character and all paths from the root to a leaf represent one of the strings.

**Radix tree** A Radix tree or *compact prefix tree* or *patrica tree* (Morrison, 1968) is a trie, where every node that has only one child is merged with its child and edges contain substrings (the characters of the merged edges; see Fig. 2.5). This data structure is more space efficient than a trie.

**Index** A *substring index*, henceforth simply called *index*, is a data structure that allows searching occurrences of a substring of the target string in sublinear time.

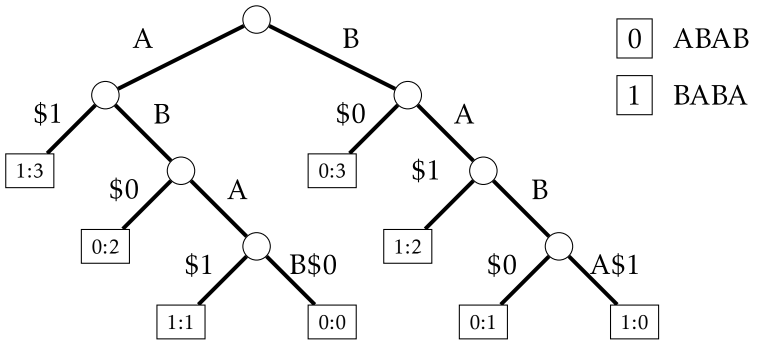


***Figure 2.6:*** *Generalized suffix tree*
$i is the terminator of the i-th string

**Suffix tree** The suffix tree (Weiner, 1973) of string $S$ is the radix tree built over all suffixes of $S$ (see Fig. 2.6). Since every substring of $S$ is the prefix of a suffix of $S$, we observe that the first occurrence of another string $T$ in $S$ can be found in $O(|T|)$ time, i.e. independent of the size the tree.[3] All $m$ occurences can be found in $O(|T| + m)$, thus the suffix tree can be used as an index. A *generalized suffix tree* is a version of the suffix tree that works on a set of strings, instead of a single string. The only difference is that the special unique termination character that is appended to $S$ prior to tree construction must be different for every string, so that leaves indicate where they originate from. In the following we will use the term suffix tree for generalized suffix trees.

**Suffix array** The suffix array (Manber and Myers, 1993) of the string S is simply the lexicographically sorted array of the suffixes of S, usually implemented as the array of the starting positions of these suffixes. It is related to the suffix tree, in that it represents the strings "emitted" by the suffix tree in depth-first traversal. It is more space efficient, but lookups are slightly slower ($O(log(|S|))$). With the help of an auxiliary data structure, the so called LCP-table suffix arrays can fully emulate suffix trees.

**FM-Index** The FM-Index (Ferragina and Manzini, 2000) is a more complex data structure that behaves similar to a suffix array, but that works with the so called Burrows-Wheeler transform. It's main characteristics are that it is highly compressed, containing also the full underlying data in the indexing structures and that the compressed index can be searched quickly.

**Hash table** A hash table is a very generic data structure that stores associations of keys to values and is then able to retrieve the value to a given key in constant time (on average or even worst-case, depending on the kind of hashing). It is used for a variety of tasks, beside indexing and outside of bioinformatics. When being used for indexing, the key is the search term and the value is a list of positions or (string, position)-pairs in the case of multiple target strings. It is also possible to combine a hash table with a suffix array, or construct the former through the latter, in which case the value contains a range on the suffix array instead. Due to the nature of the hash table all valid search terms must be known at the time of indexing. Thus it is impossible to provide an index

[3]This is true only for finite size alphabets, but this obviously applies for our cases.

for arbitrary search strings with hash tables. However, if the seed length(s) are known at the time of indexing the amount of keys shrinks to $|\Sigma|^q$, where $\Sigma$ is the alphabet and $q$ the seed length. A hash table that maps all occurrences of length $q$ (*$q$-grams* or *k-mers*) is a form of *Q-Gram-Index*.

## 2.2 Seeding & Filtration strategies

Identification of candidate regions for hits is usually called *seeding* if the logic behind the step implies the positive identification of these regions through short seeds, i.e. substrings of one or many query sequences.

If the method emphasizes the exclusion of regions unlikely to include search terms, this step is called *filtration*. This is however only a matter of terminology or perspective, the outcome is the same.

The result of seeding could look like in Fig. 2.7, where every hit is a candidate for extension or further evaluation before extension.

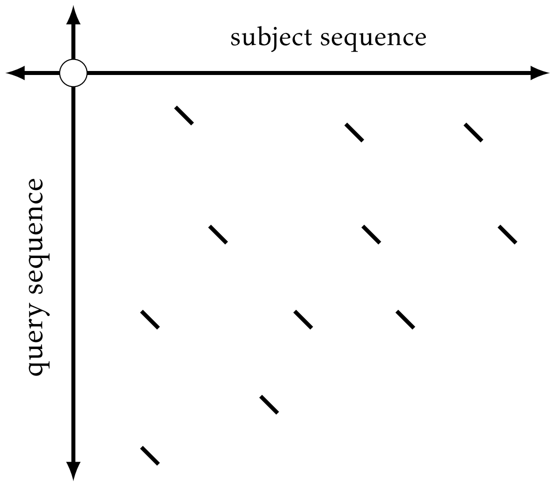


***Figure 2.7:*** *Result of seeding*

Most modern applications use one or multiple of the previously introduced data structures to achieve fast seeding. This comes at the expense of more memory (compared with simple databases like BLAST's), but memory has dropped magnitudes in price, since BLAST was first releases, so this seems feasible for most approaches. In general the aforementioned techniques are only applied to the subject sequences, but recently programs were also published that index the query-sequences in addition to the subject-sequences. Among those are SANS (Koskinen and Holm, 2012), which creates a suffix array (or more precisely an inverse suffix array) on the query and the Masai read mapper (Siragusa et al., 2012), whose approach will be discussed in Sec. 3.5.2. This promises further speed increases, especially where repetitive input data is expected.

When applied to subject sequences, indexing is usually out-sourced to an extra application or computation step, because this part of a setup is seen as constant, and indices to a database file could be distributed together with the file.

### 2.2.1 BLAST

All versions of BLAST perform seeding in the following steps (however parameters have changed over the course of time). For every query sequence:

1. generate the list of all k-mers of the query sequence, i.e. all overlapping substrings of length k; in protein space $k = 3$ by default
2. extend the list by adding "related" k-mers, where another k-mer is judged as related if it scores a certain threshold T with an existing k-mer
3. construct a finite state machine (Mealy, 1955; Hopcroft and Ullman, 1979) from the k-mers and do an online search on every subject sequence with it

The first version of BLAST (Altschul et al., 1990) considered every match reported by the finite state machine as a candidate region, but this resulted in very many meaningless verifications. The approach was changed for the second version of BLAST, also known as gapped BLAST (Altschul et al., 1997):

Instead of verifying every match, only regions of length A (40 by default) that contain at least two matches on the same diagonal are verified, i.e. the distances on both sequences must be identical and $\leq A$ (see Fig. 2.8). To compensate for the loss of sensitivity T was lowered in gapped BLAST.

BLAST+ (Camacho et al., 2009), the latest version of BLAST works in the same way as gapped BLAST.

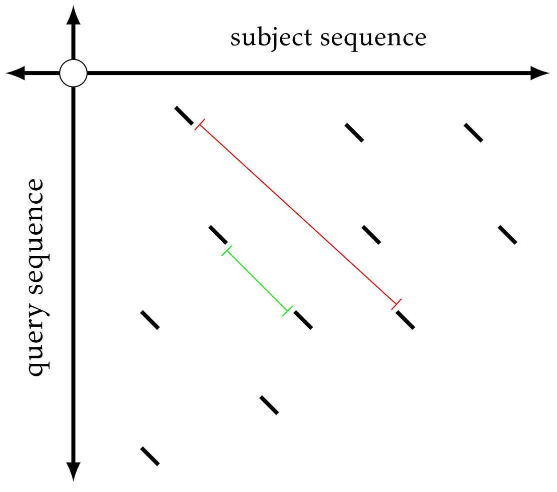


***Figure 2.8:*** *Seeding in BLAST2 and BLAST+*
In one case the hits are too far seperated (red), in the other they are close enough to be verified (green)

### 2.2.2 PAUDA

PAUDA(Huson and Xie, 2013) uses Bowtie2(Langmead and Salzberg, 2012) internally for the entire search and alignment steps, with only custom pre- and post-processing that will be explained in Sec.2.4.

Bowtie2 is run by PAUDA with a seed length of 18 and 0 or 1 allowed mismatches. The seeds are searched in an FM-Index and the candidate regions are ranked according to the abundance of a seed's hits, i.e. the less candidate regions a seed has, the higher these are ranked.

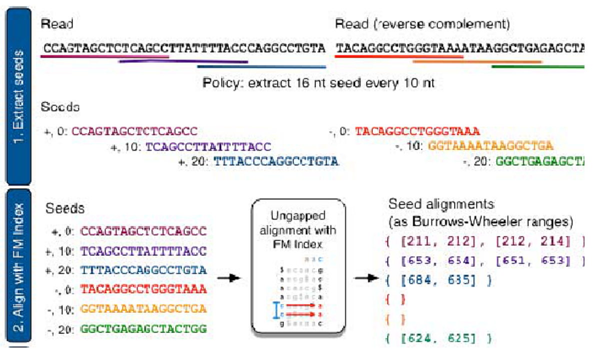


***Figure 2.9:*** *Seeding in Bowtie2 – The first two steps of the Bowtie2 pipeline. Taken from Suppl.Fig.1 of Langmead and Salzberg (2012); prioritization step omitted.*

### 2.2.3 RAPSearch2

RAPSearch stands for **R**educed **A**lphabet based **P**rotein similarity **S**earch; it employs an alphabet reduction that will be explained in Sec. 2.4. RAPSearch (Ye et al., 2011) used a suffix array to search for the seeds, but in RAPSearch2 (Zhao et al., 2012) the authors switched to a collision-free hash table. This is a form of hash table that guarantees constant-time lookups. In both versions of RAPSearch the seed length is 6 and no errors are allowed in the seed. Although the seed length is fixed at the time of indexing, the hash-table is built in a way, where positions are sorted according to the four residues following the seed, so lengths between 6 and 10 are supported, although their lookup is not possible in constant time anymore.

A bit-compressed representation of amino acid seeds is used in the database, that includes both the reduced alphabet version of the seed and the canonical one. This makes constant time lookups of unreduced query strings possible, i.e. reduction happens transparently while hashing.

### 2.2.4 UBlast

UBlast (Edgar, 2010) also utilizes one of the aforementioned indexing techniques, the q-gram-index. It shares this approach with other local aligners, like BLAT (Kent, 2002). Similar to RAPSearch2 it uses a seed length of 6, but in contrast to it, it uses so called gapped q-grams, also known as shapes or patterns (Burkhardt and Kärkkäinen, 2003).

These contain wildcard positions that are not checked in addition to regular positions. This method improves specificity, without decreasing sensitivity, but choosing a pattern can be difficult (Burkhardt and Kärkkäinen, 2003). The default shape for protein searches in UBlast is 10111011 (six positions must match, two particular ones need not). In addition to using shapes, UBlast also seeds on a reduced alphabet.

## 2.3 Extension / Verification

The *verification* of a candidate region is usually performed by *extending* the seed(s), scoring the resulting alignment and printing it if it fulfills previously defined criteria (e.g. minimum score, minimum sequence identity). Less focus is given to this step by many researchers, therefore unfortunately detailed information is not available for all of the techniques and algorithms used. Common to many approaches is the concept of *dynamic programming*, as explained in Sec. 2.1.3.

### 2.3.1 BLAST

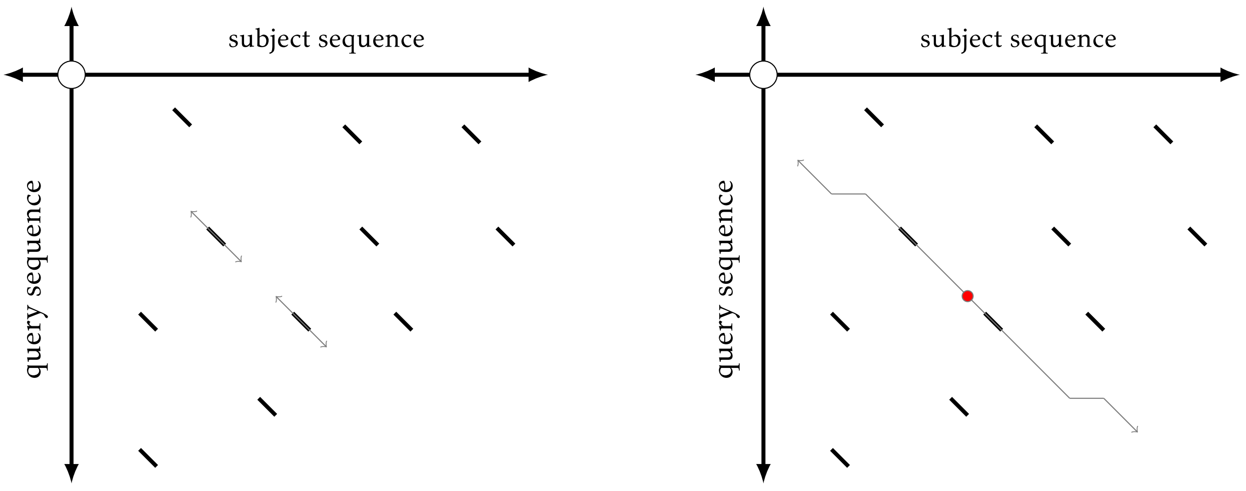


***Figure 2.10:*** *BLAST extension (ungapped and gapped)*

In this stage of the algorithm we have two or more k-mers within a certain distance (the explanation for the first version of BLAST is omitted, since it is deprecated). The extension steps are as follows:

1. perform an ungapped extension of each k-mer, as long as the overall score does not drop $X_u$ (default = 6) [4] below the maximum score seen
2. if the ungapped extensions together do not reach a certain threshold $S_u$, discard the candidate region
3. find the best scoring stretch of length 11 in the current ungapped alignment
4. from the center of this region (exemplified by a red dot in Fig. 2.10) start a gapped extension in both directions; this time a greater x-drop value $X_g$ (default = 27) is used; no alignment is computed at this point, only the score
5. again, if this score doesn't meet a threshold $S_g$, discard the candidate region
6. perform the gapped extension again, from the same starting point, using an even larger $X_f$ (default = 53) as drop off and compute the alignment
7. calculate the e-value of the alignment and discard if it is above the e-value threshold

This iterative approach postpones the most expensive tasks to the end and has various heuristic checks to avoid going through them for every candidate region. Behavior like this is characteristic of heuristic local alignment approaches, as we will see later on.

The dynamic programming method employed by BLAST, the so-called x-drop extension, is a special case of the banded approach, discussed in Sec. 2.1.3. It does not compute a constant size band, but instead it stops computing cells whenever the score of the cell drops $x$ below the global maximum. This has the advantage of dynamically following the currently best path, instead of enforcing the less flexible band. As with other heuristics, it does open the possibility of missing a globally best extension that would have been found with a full dynamic programming matrix, or even with a fixed band.

### 2.3.2 PAUDA

Bowtie2 also takes care of the extension phase in PAUDA, so all information regarding Bowtie2 applies. Bowtie2 verifies a region by selecting a range around the seed hit[5] and doing a full dynamic programming on the rectangular matrix. To speed up the calculation it divides the seed into chunks which it processes in parallel via SIMD features of modern processors (see Sec. 2.5). Additionally, it caches the possible score contributions of each of these chunks to save cell operations.

If more than a certain number of seed hits of one query have been rejected in a row (15 by default), no further extensions will be attempted for this query. This aims at reducing the amount of unnecessary extensions, and since seeds have been ranked, the chance of later seeds producing a significant alignment is perceived as diminished.

### 2.3.3 RAPSearch2

According to the publication, extension and alignment follow " *the same approach used in BLAST*" (Ye et al., 2011, p.4). Further details are not available.

[4] BLAST actually uses bit-score drop offs. For comparability we here give the raw scores for BlastX with default scoring scheme that we re-transformed from the bit-scores.

[5] Unfortunately it is not clear from the publication and the supplementary materials *how* it selects this region and how big it is.

### 2.3.4 UBlast

Before extending a hit *"UBLAST uses an unpublished, proprietary method to reduce the number of alignments that are constructed."*. [6] After the amount of hits was thus reduced, UBlast extends in a similar manner to BLAST, but in only two steps, one ungapped and one gapped extension. The x-drop for the first is 16 and for the second 32. It is unclear whether hits are also discarded in between both extensions and if yes, by which criteria.

## 2.4 Alphabet reduction

Research on the functional redundancy of amino acids dates back to the late 70s of the 20th century (Sander and Schul, 1979). It has mostly been used in structural research, i.e. the analysis and prediction of protein folds and the de-novo design of functional proteins (Regan and DeGrado, 1988). The main purpose of reducing the alphabet today is the reduction of computational complexity, while sacrificing as little sensitivity as possible. Many approaches are specific to certain protein families and derive their grouping of amino acids directly from the biochemical and physical properties of the respective amino acids (Regan and DeGrado, 1988). Other metrics include Miyazawa-Jernigan interactions (Miyazawa and Jernigan, 1996), used by Wang and Wang (1999) and substitution matrices, like Blosum (Henikoff and Henikoff, 1992), used by Murphy et al. (2000) and Li et al. (2003).

The latter approaches are especially useful for sequence alignment, because substitution matrices are also fundamental parts of scoring algorithms in most sequence alignment applications and the impact of reductions that are based on the same metric as the target function is intuitively clear. Beside the method of reduction, integral parameters are the size of the original alphabet and the desired output size of the target alphabet, i.e. the number of clusters that remain after reduction. All of the aforementioned methods begin the reduction on the canonical 20-letter amino acid alphabet that includes all proteinogenic amino acids, without the rare amino acids Selenocystein (U) and Pyrrolysine (O) and that does not include a character for the STOP-codon and non of the wildcard characters frequently encountered (X for "any amino acid"; B for "N or D"; Z for "Q or E"). Depending on the method the target size may be fixed or variable, some research indicating that sizes as low 5 are sufficient (Bacardit et al., 2009), most suggesting that 10-12 letters are required and/or most effective (Li et al., 2003; Murphy et al., 2000; Ye et al., 2011).

Of the programs previously introduced both RAPSearch2 and UBlast use the 10-character reduction developed by Murphy et al. (2000), subsequently referred to as *Murphy10*. Both programs apply the reduction only during seeding and do the extension on the regular alphabet to achieve a higher sensitivity in the final step. In combination with the size of the alphabet the seed length has a very strong influence on sensitivity, specificity and performance of the filter. See Fig. 2.11 for a comparison of different alphabets and seed lengths. Both RAPSearch2 and UBlast set a default seed length of six (see above).

PAUDA employs a different kind of reduction, which it calls pseudo-DNA or pDNA, because it reduces to target size 4, the size of the DNA/RNA-alphabet. This enables PAUDA to utilize Bowtie2 in its pipeline (which was built to work in DNA-space). To compensate for the the loss of specificity per character, the much longer seed length of 18 is chosen.

[6] `http://drive5.com/usearch/manual/ublast_algo.html`

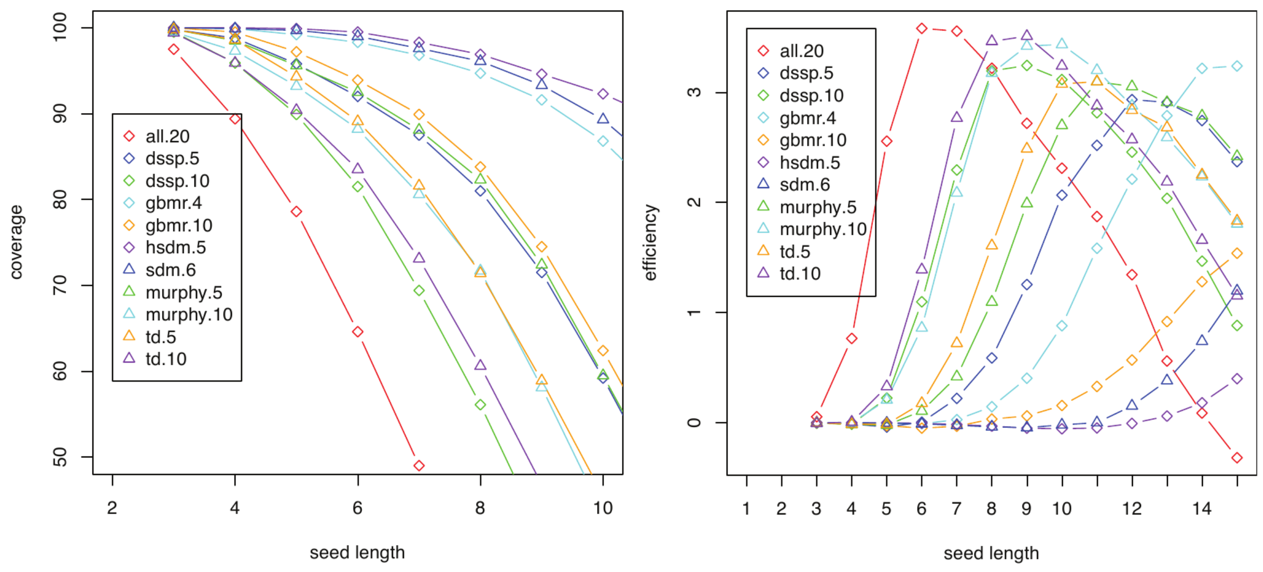


***Figure 2.11:*** *Comparison of different alphabet reductions in regard to speed and sensitivity*
Taken from Ye et al. (2011); all.20 is the full canonical set of amino acids, murphy.10 is the aforementioned; see the original publication for details on the other alphabets and definitions of *efficiency* and *coverage*.

## 2.5 Parallelization

Computing capacities have increased drastically in the course of late 20th century, first only in the commercial and academic space, later also in the form of personal computing and recently also in mobile computing. A pattern that has been observed in this development is the so-called Moore's law, which states that the number of components per chip doubles every 1-2 years (Moore, 1965). While having been formulated in 1965, it still holds today, although the practical terms that uphold it have changed. The performance gains associated with Moore's law have mostly been attributed to frequency scaling in the past, i.e. the increase of CPU cycles per time, but this is beginning to reach physical limits (Adve et al., 2008). While previously used only in high-performance computing, other areas have adopted *parallel computing* as the central paradigm to increase performance. The year 2004 is often cited as a turning point in this development (Asanovic et al., 2006).

Since this development has greatly influenced bioinformatics, we will briefly introduce some of the concepts that play a role in this field (and skip over related topics that have no part in the software discussed in this work). There are different ways of classifying concepts and technologies in parallel computing, we chose to present them in how they appear to the programmer.

**SMP** Symmetric multiprocessing (SMP) is available on most personal and academic computing devices nowadays, from desktop computers, over cellular phones to more powerful servers in business and research. SMP-capable systems have multiple identical central processing units (CPUs) and/or multiple identical processing cores per unit. Smaller local memory caches are usually available per core or subset of cores, but the main memory of the system is shared between all cores and all cores are managed by the same operating system. The operating system automatically assigns system processes and threads to cores, so to efficiently utilize the parallel computing capabilities of a system, the programmer needs to distribute work evenly on multiple threads. This task is non-trivial (Patterson and Hennessy, 1997). In addition to structurally and

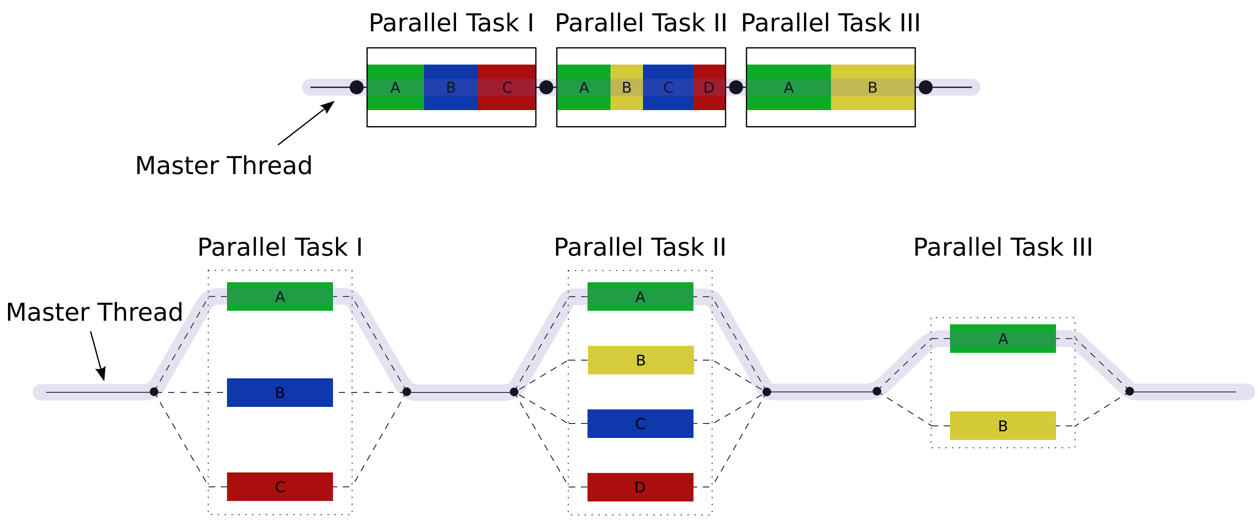


***Figure 2.12:*** *Pipeline of a program with and without parallelism*
Creative Commons licensed work(🅭🅯🄎), taken from Wikimedia Commons

conceptionally adapting a program for multi-threading, extensions or libraries have to be used to implement the threading in most languages, e.g. C and C++ did not have standardized support for multi-threading until their C11/C++11 specifications. Among these extensions OpenMP[7] is one of the most widely supported (GNU, Intel, Microsoft, Oracle and IBM compilers) and also used in this work. Although the source-code of UBlast is not available and this feature not documented, an analysis of the binary showed that it uses OpenMP, as well. Other popular approaches abstract operating system specific thread implementations, like the Boost library[8] and the threadpool library[9] which builds upon Boost and is used by e.g. RAPSearch2.

**CPU extensions** Many modern CPUs, including all mainline processors by AMD and Intel, contain built-in extensions that allow them to perform multiple instructions in parallel on one CPU/-core. Usually this is limited to SIMD – Single instruction, multiple data; this means that a series of operations of the same kind, e.g. floating point additions, can be performed on a set of values at the same time. Of the programs discussed here PAUDA (through Bowtie2) makes use of SSE, the Streaming SIMD Extensions, to speed up its dp computations.

**GPGPU** General-purpose computing on graphics processing units is a method that moves the work from the CPU of the system to the GPU. This differs more strongly from SMP on the CPU, because "*[a] CPU consists of a few cores optimized for sequential serial processing while a GPU consists of thousands of smaller, more efficient cores designed for handling multiple tasks simultaneously.*" [10] GPUs however are more difficult to program for and not every problem can be broken down to thousands of pieces. None of the previously mentioned programs make use of the GPU, however work is underway to bring support to the SeqAn library (see Sec. 5.3).

[7] http://www.openmp.org
[8] http://www.boost.org
[9] http://threadpool.sourceforge.net
[10] http://www.nvidia.com/object/what-is-gpu-computing.html

## 2.6 Summary

| | BLAST | PAUDA | RAPSEARCH2 | UBLAST |
|---|---|---|---|---|
| search-method | online-search | lookup in FM-Index | lookup in hash-table | lookup in hash-table |
| seed length | $2 \times 3$ | 18 | 6 | 6 |
| alph. red. seeding | ~ | ✓ | ✓ | ✓ |
| alph. red. extension | – | ✓ | – | – |
| red. alph. size | – | 4 | 10 | 10 |
| max. seed error | 0 | 0 or 1 | 0 | 0 |
| extension | ungapped & gapped x-drop | rectangular DP | ungapped & gapped x-drop | ungapped & gapped x-drop |
| parallelized | partly SMP | SMP, SSE | SMP | SMP |

***Table 2.1:*** *Overview of existing tools and their parameters*

Tab. 2.1 contains an overview over the presented tools and their most important parameters / features. BLAST has a '~' at "alph. red. seeding" to symbolize the word neighborhood expansion.

# 3 Methods & Implementation

Built on the foundations of the SeqAn library (Döring et al., 2008), written in C++ and OpenMP, we have developed **Lambda**, the **L**ocal **a**ligner for **m**assive **b**iological **da**ta. Lambda implements many of the previously presented ideas in an efficient manner. Core functionality is provided by existing features of the SeqAn library, some features were added to the library in the course of the development of Lambda and other features are currently only available in the application.

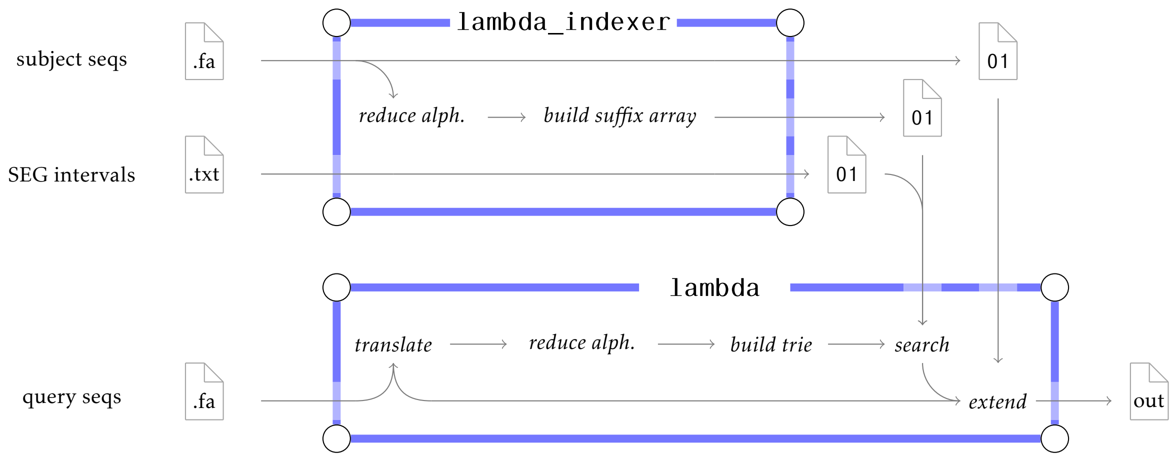


***Figure 3.1:*** *Pipeline of Lambda (BlastX-mode)*

The pipeline of the application, when operating in BlastX-mode, is displayed in Fig. 3.1. Lambda consists of two binaries, `lambda_indexer` and `lambda`. The indexer is responsible for pre-processing the subject sequences and providing various data structures in native binary formats to the main `lambda` executable. The latter then reads the query sequences, processes them, performs the search on the index, extends the hits and writes the results to disk.

This chapter will cover the specifics of the implementation and associated algorithms. App. A.1 contains a full overview over the program parameters. Where nothing contradictory is stated, information on run time parameters, given in monospaced text and in both long and short versions, refers to the main `lambda` executable.

It should be noted that since Lambda is a heuristic application the choice of parameters – both user-visible and internal – is the result of constant benchmarking. The development and the use of the benchmark (see Sec. 4.1) were an integral part of this work.

## 3.1 Input

Lambda supports FASTA (Lipman and Pearson, 1985; Pearson and Lipman, 1988) files for input of both query and subject sequences. For query sequences FASTQ (Cock et al., 2010) is also supported, however qualities are stripped, so in some applications it might be advisable to trim the query sequences (remove trailing low-quality bases). Depending on the sequence types in the input files and desired operations, different program modes can be used in NCBI BLAST (see Tab. 3.1). In contrast to other tools all of these modes are supported by Lambda (selectable with `--program` / `-p` parameter to both executables), but as the discussed metagenomic use cases revolve around BlastX, this mode of operation was the focus of this work. Except where otherwise noted, this documentation refers to the BlastX mode of Lambda.

| Program-Mode | Query-Alphabet | Subject-Alphabet |
|---|---|---|
| BlastN | Nucleotide | Nucleotide |
| BlastP | AminoAcid | AminoAcid |
| BlastX | translated Nucl. | AminoAcid |
| TBlastN | AminoAcid | translated Nucl. |
| TBlastX | translated Nucl. | translated Nucl. |

***Table 3.1:*** *BLAST program modes and corresponding input alphabets*

The SeqAn library has support for a wide variety of other formats, beside FASTA and FASTQ, including Embl flat file[1] and Genbank[2], as well as compressed versions of the aforementioned (gzip[3] and bzip2[4]). These were not tested with Lambda, but could be easily supported, when publishing a proper release.

## 3.2 Translation

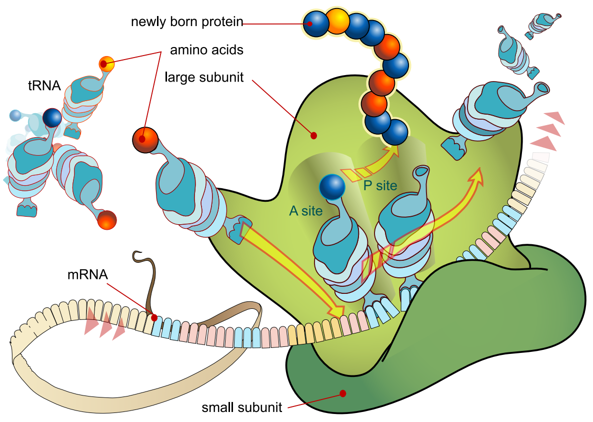


***Figure 3.2:*** *Translation in the cell*

Public Domain work, from Wikimedia Commons

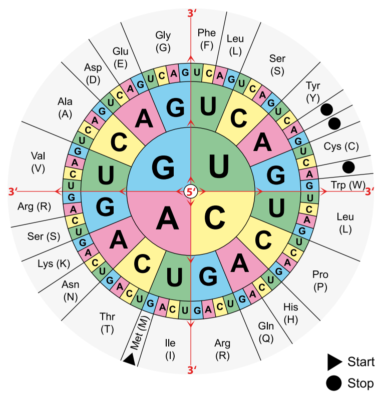


***Figure 3.3:*** *Universal genetic code*

Public Domain work, from Wikimedia Commons

In BlastN and BlastP mode the sequences have the same alphabet and can be compared without further pre-processing. However most metagenomic studies that are interested in the functional analysis and identification of the sample want to search protein databases with genomic or transcriptomic sequences, i.e. sequences in DNA or RNA alphabet.

[1] `http://www.ebi.ac.uk/ena/about/sequence_format`
[2] `http://www.ncbi.nlm.nih.gov/Sitemap/samplerecord.html`
[3] `http://www.gnu.org/software/gzip`
[4] `http://www.bzip.org`

To permit the comparison of these two sequence types, the sequences in DNA or RNA alphabet needs to be *translated* into amino acid space. *In vivo* the information contained in a gene goes through multiple steps, before a protein is created from it. This process contains the steps of transcription and translation[5], of whom the latter ultimately defines the relation between nucleic acid sequence and amino acid sequence. The translation process in vivo is depicted in Fig. 3.2; for every three mRNA characters, one amino acid is appended to the new protein. The process is facilitated by the ribosome and tRNAs of which different ones exist for different amino acids. This mapping happens according to the so called *genetic code*. Multiple genetic codes with slight variations exist in nature, the most common one, the *universal genetic code* is displayed in Fig. 3.3.

Translation *in silico* is a little more complicated, because (a) we don't know whether the strand that the query was read from is the one with the putative gene, or its reverse complement; and (b) since triplets are converted, but the beginning is not known, there are three different offsets, called frames, that conversion can start from. Thus, to avoid missing the right one, a total of six frames in protein space are generated from a single nucleic acid sequence (three frames on two strands).

As part of this work an efficient implementation for six-frame translation was added to the SeqAn library. It is adapted to and specialized for SeqAn's data structures and makes use of OpenMP, rendering it possible to translate 14 million reads of $\sim 100$bp length, with six frames in 2.3 seconds (including time to allocate memory for the target strings).[6] The implementation in the library offers all 19 genetic codes currently regarded relevant by the NCBI,[7] which could be an advantage over other applications, especially when working with "exotic" samples.

Translation in Lambda is performed on the query sequences in BlastX and TBlastX modes, and on the subject sequences in TBlastN and TBlastX modes. Lambda currently defaults to the universal code, but it would be trivial to add a parameter for selection.

## 3.3 Alphabet reduction

As discussed in Sec. 2.4, alphabet reductions play a key role in the design of modern local alignment software. Due to its popularity (Zhao et al., 2012; Edgar, 2010) and its success in empirical comparisons (Ye et al., 2011), we chose to implement the Murphy10 reduction (Murphy et al., 2000) as a possible default for Lambda. It is based on amino acids' correlation in the Blosum50 substitution matrix and canonical 20-letter alphabet. SeqAn works with a 24-letter amino acid alphabet by default, and both terminator-encoding codons and wildcard codons are to be expected in translated read data, since the former occur "naturally" and the latter are the result of unknown DNA-bases ('N'). Thus we chose to additionally implement new versions of alphabet reductions in the hope of increasing the specificity of the filter. These new reductions are based on Blosum62, instead of Blosum50, because the former is used for scoring as well, and therefore seems more adequate.[8]

[5]And in fact many other modification steps that cannot be predicted as easily and make local alignments all the more useful over global / semi-global alignments which depend on high sequence identity.

[6]This was measured with is the same hardware described in the results, Sec.4.1.3.

[7]As compiled by Andrzej Elzanowski and Jim Ostell, updated April 30, 2013, `http://www.ncbi.nlm.nih.gov/Taxonomy/Utils/wprintgc.cgi`

[8]The denominator of the Blosum matrix indicates to what degree sequences where maximally identical when computing the matrix elements (see Sec. 2.1.1). Since the alphabet reduction implies a putative alignment, the same assumptions should be made and thus the same matrix used. Blosum62 was used, because this is also the default in other local alignment software, most importantly BLAST.

| Scheme | Clusters | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Murphy10 | A | ST | KR | EQND | C | G | H | ILMV | FYW | | P | |
| Lambda12 | A | ST*X* | KRZEQ | ND*B* | C | G | H | ILMV | FY | W | P | * |
| Lambda10 | AST*X* | | KRZEQ | ND*B* | C | G | H | ILMV | FYW | | P | * |
| Lambda08 | AST*X* | | KRZEQND*B* | | C | G | H | ILMV | FYW | | P | * |

***Table 3.2:*** *Alphabet reductions available in Lambda*

Italic characters are not found in the 20-letter alphabet, '*' is the terminator symbol, other symbols according to IUPAC.

Unfortunately Murphy et al. (2000) are unclear on the exact algorithm of clustering. The description in the second paragraph of the "Materials and methods" section in the publication clearly suggests an agglomerative, hierarchical clustering approach[9]; an assumption which is supported by a dendrogram figure of a subset of the reductions. However this is contradicted by another paragraph in the text, wherein e.g. Proline is depicted as first grouping with HC (6 letter reduction), then with ASGT (5 letter reduction). Furthermore the clustering metric is described as the correlation coefficient, but the formula given differs significantly from the well known Pearson product-moment correlation coefficient, by omitting the square root and the subtraction of means. This might have been intended, however it is not explained, and – due to the large denominator – numbers become very small, encouraging rounding errors. Another factor of uncertainty is that the Blosum matrices are computed based on existing sequence data, and have changed over time, with no information available on the exact version used by Murphy et al. (2000); a reference Blosum50 is not included with Henikoff and Henikoff (1992).[10]

Due to these circumstances we were unable to reproduce the documented clustering results with either the metric described or the canonical correlation coefficient (and Blosum50, 20-letter amino acid alphabet). But we achieved *similar* results, using agglomerative, hierarchical clustering, with Pearson's correlation coefficient as metric and WPGMA as clustering method[11]. Based on this approach we generated alphabet reductions for target alphabet sizes 2-23 on the SeqAn amino acid alphabet and Blosum62. Together with the original Murphy10 reduction these where added to the SeqAn library in the form of a module.[12] A complete overview of the reductions is available in App. A.2.

Tab. 3.2 gives an overview over the alphabets available in Lambda. As can be seen from the table, the only main difference in this range (other than including extra characters) is "EQ" clustering in a different group. They can be selected by providing the `--alph` / `-a` parameter. Choosing a value of 0 will disable alphabet reduction and make Lambda use the regular alphabet for seeding. Note that the same parameter needs to be passed to `lambda_indexer`, as the index needs to be reduced in the same manner as the query.

[9] *"[...] second, the two amino acids with the highest correlation coefficient are grouped together, then the pair with the next highest correlation is either added to the first group if one member is already in the group or separated into a new group if not, and the process is repeated until all the amino acids are divided into the desired number of groups."* – Murphy et al. (2000, p.150)

[10] See also the documentation for R's biostring project on the inconsistency of Blosum matrices: `http://svitsrv25.epfl.ch/R-doc/library/Biostrings/html/substitution_matrices.html`.

[11] The publication suggests single-linkage as clustering method(see the quote in fn.9), but single-linkage exhibited strong chaining behavior in our tests and produced results that deviated heavily from the expectations.

[12] The clustering was implemented in GNU Octave(`http://www.gnu.org/software/octave/`) and C++ code for inclusion in SeqAn was generated with a converter, written GNU AWK.

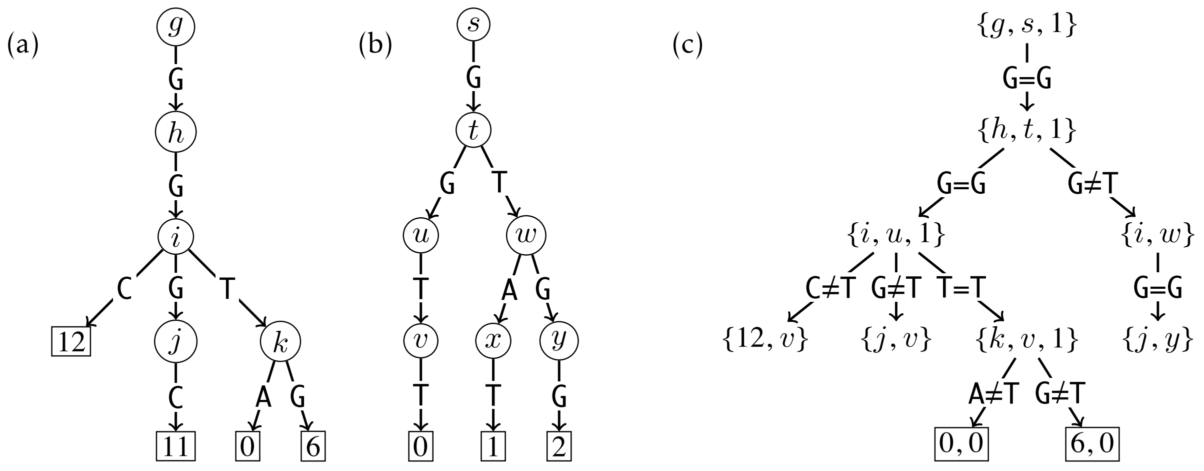


***Figure 3.4:*** *Multiple backtracking*

(a) A part of the suffix trie representing the text GGTAACGGTGCGGGC [. . . ]. Numbers on the leaves are suffix positions in the text, whereas letters on the inner nodes are arbitrary and serve to distinguish nodes from each other. (b) The trie representing the set of patterns GGTT, GTAT, GTGG, respectively numbered 0, 1, 2. Labels on the leaves show pattern numbers, whereas labels on the inner nodes are again arbitrary identifiers. (c) [. . . ] Edges represent comparisons performed [. . . ], nodes with curly brackets represent recursive calls, rectangular leaves represent approximate matches reported. In this example, pattern numbered 0 (GGTT) matches the text twice, at positions 0 and 6, within 1 mismatch. For simplicity, we omitted terminator symbols in the picture. [original figure and description from Siragusa et al. (2012)]

## 3.4 Seeding and search phase

### 3.4.1 Main methodology

After having reduced the alphabet of the protein sequences, Lambda divides the query sequences into seeds, of a given length $q$, selectable by the parameter `--seedlength` / `-sl`. This parameter is a major factor in sensitivity, specificity and performance, as will be seen later. Lambda creates non-overlapping seeds, i.e. it does not generate all $q$-mers of each query-sequence, but only $\lfloor \frac{l}{q} \rfloor$ seeds, where $l$ is the query length. It differs in this approach from the other applications discussed here, but short seed lengths in combination with the approximate seeding (explained below) should compensate for this.

Lambda uses a search approach, which was originally developed for the Masai read mapper (Siragusa et al., 2012) and later during the development of this work adapted for general use in the SeqAn libarary. Lambda already used early versions of this code and was the first project outside of Masai to adopt it. The approach is based on the concepts of *double indexing* and *multiple backtracking*. Double indexing refers to the fact that in contrast to all of the other tools discussed, Masai builds an index over both, the seeds (of the queries) and the subjects. The indexing structure built on the query sequences is a Radix tree and the indexing structure built over the subject sequences is a suffix array that is conceptionally used as a suffix tree.[13] The suffix array is constructed by `lambda_indexer` in an extra pre-processing step, then loaded from disk by the `lambda` executable; the index over the seeds is constructed on-the-fly. The suffix array is then searched by backtracking (Ukkonen, 1993), but with the trie of the seeds instead of a single sequence, so that

[13]Other data structures, like FM-Indices are also supported, but the Masai publication recommends the suffix array for speed reasons.

multiple seeds are processed in parallel. Fig. 3.4 illustrates the method, the algorithm is explained more thoroughly in Siragusa et al. (2012).

As previously noted this method allows for searching seeds with both substitutions and insertions/deletions. Lambda offers the parameter `--seeddelta / -sd` to specify the maximum distance allowed and `--ungappedseeds / -su` (0 or 1) to limit errors to substitutions, i.e. use hamming distance instead of edit distance. Setting `-sd 0` will results in exact seeding.

### 3.4.2 Post-processing of match-candidates

Extensions are computationally expensive, so it is important to prevent doing unnecessary ones. To this end one should try to remove extension candidates which will (a) lead to biologically insignificant alignments, or (b) produce alignments that have already been computed as the result of another hit.

**Masking** To identify candidates for the first sort of problem, so called *low complexity region filters* are frequently used, among them SEG (Wootton and Federhen, 1993) for protein sequences and DUST for nucleic acid sequences (see the supplementary data to Morgulis et al., 2006). Low complexity regions include repetitive or provenly insignificant sequence patterns that can score well nevertheless – and hinder efficient searches (Altschul et al., 1994). These filters can directly mask the sequences “on-line” or generate tables which contain the coordinates of masked regions. On-line masking consists of either replacing masked characters with the “unknown”-character (`'N'` for DNA/RNA and `'X'` for amino acids) or replacing the character with a lower case version, indicating to a program that this character is masked. If masking prevents only the successful seeding inside the region (but not extension into it), it is referred to as *soft-masking*, otherwise it is called *hard-masking* (Xiong, 2006, p.56). Replacing characters with the “unknown”-character is always hard-masking, since the information at this position is lost to the program reading the file.

Masking can be performed on both the query and the subject sequences, however only masking of the subject sequences is currently supported by Lambda. To this end `lambda_indexer` can be passed an interval-file generated by the `segmasker` utility included with BLAST+ (Camacho et al., 2009), the parameter is `--segfile / -s`. After seeding, Lambda will then filter out seeds that reach into a masked region with at least half of their length (soft-masking).

**Merging** In order to address the second problem, overlapping seeds and seeds within a certain proximity could be merged – since it is likely, that if both are extended to valid alignments, it will be the same one. Lambda offers the parameter `--seedgravity / -sg` to define this behavior: positive integers are distances, negative integers are overlaps. This is a very heuristic parameter, and its influence on results is not immediately intuitive,[14] however a value equal to the seed length has shown good results, both in terms of sensitivity and performance. This choice of default parameter is supported by the assumption that a contiguous match could be “interrupted” by exactly a seed’s length, since we don’t have overlapping seeds.

Lambda also provides the parameter `--seedminlength / -sm`, which can be used to specify a size longer than the original seed length, thereby filtering out all regions, that do not contain

[14]A larger “seed-gravity” can both, increase the final number of valid final results, because longer seed regions are more likely to pass the minimum score test (see below), and it can reduce the number of results, as obviously only at most one alignment can come from each region.

multiple seeds. This enables functionality similar to BLAST's seeding algorithm, however it did not prove advantageous, yet (see below).

### 3.4.3 Query partitioning

A drawback of the double indexing approach is, that it is necessary for the seeding step to complete, before seeds can be post-processed and verified. This is due to the nature of the algorithm: since seeds are not grouped by query, but instead integrated into a single trie, it is not possible to conclude that all hits for one sequence have been found, until the entire process is complete. This makes it necessary to cache all hits, which is very expensive memory-wise. Consider that a hit consists of query and subject identifier and begin and end positions on both sequences and it becomes evident that this can quickly exceed main memory.[15] Whether this situation arises depends strongly on the choice of seed length and the reduction used, as well as on the size of the search space. In general, a reduction to a smaller alphabet will yield more hits, as will a shorter seed length (Ye et al., 2011).

To remedy this problem we implemented a partitioning of the query, selectable with the `--querypart` / `-qp` parameter. It works by splitting the query sequences into $n$ blocks before the entire seed and search step, and then processing these blocks sequentially. Since the number of hits are reduced drastically by masking and merging, this reduces the maximally required memory. Of course, sequential processing reduces the speed of the program and multiple backtracking will become less effective, since a seed that appears in two different queries might be searched twice.

Parallelization of the seeding step is also implemented through query partitioning, as $m$ blocks will be searched in parallel, were $m$ is the number of CPUs/-cores present in the system ($m$ can be overridden by an environment variable). By default $n = m$, which is the best performing setting and results in no sequential execution and thus saves only very little memory.[16] As previously explained, the user should set `-qp` to a value higher than the available number of threads (ideally a multiple thereof) if he or she runs out of memory.

## 3.5 Extension phase

The extension phase consists of two steps, a local alignment on the seed region and a gapped extension. Both happen on the unreduced amino acid sequences, to guarantee full sensitivity. After the local alignment, and depending on its score, the seed hit is either extended or discarded. Especially on seeds from a reduced alphabet this ensures that hits with weak overall similarities are not extended and computation time is saved.

### 3.5.1 Local alignment

The seed hit that we process is not yet in an alignment format, we only have the coordinates of a putative alignment. Especially in cases where we allow gaps in the seeds, or where multiple seeds have been merged – spanning unknown intervals between them – building an alignment

[15] A match takes up 16 bytes of memory, so one billion hits equal more than 15GiB of memory consumption.

[16] It *can* save memory, as some blocks finish quicker than others and free some of their memory, before other blocks peak in their usage.

is crucial. Even if we only allow errors in hamming distance, these might be on the edge and it would be beneficial to clip them. We do this by computing the local alignment on the candidate region. The algorithm employed is the dynamic programming approach provided by the SeqAn librarary, in particular it is a banded version (Chao et al., 1992) of the Smith-Waterman algorithm (Smith and Waterman, 1981) with support for affine gap costs (Gotoh, 1981) and optimized for space efficiency (Hirschberg, 1975). This means among other properties that not the full dynamic programming matrix is computed, but only a band that is likely to contain the match (for more information see Sec. 2.1.3).

So long as the seed distance is $\leq 1$ the diagonals for the band of the local alignment can easily be computed as the difference between the lengths of the candidate regions on both sequences.[17] This is intuitively clear for our use cases: matches without insertions and deletions will be of the same lengths, matches with one edit operation will differ by one, merged seeds whose inter-seed regions are of different length are on diagonals of the lengths' difference away from each other.

The raw score that a local alignment needs to not be discarded can be set with the parameter `--seedminscore / -ss`. This parameter has a strong influence on sensitivity and performance, but due to its heuristic nature it is difficult to determine generic defaults. For Blosum62 we have found that values of 3-3.5 times the seed length show good results on different datasets.

### 3.5.2 Extension

Extension in BLAST and many other programs is achieved through an x-drop extension (see Sec. 2.3.1). Although an implementation for x-drop extension similar to BLAST exists in SeqAn,[18] – which is also used by the exact DNA local aligner Stellar (Kehr et al., 2011) – this was found to be inadequate for the tasks at hand, because it does not support affine scoring schemes. Instead of adapting this module, we chose to write an alignment extension module that builds on the generic SeqAn dynamic programming code. This foundation is tested more extensively and receives constant improvements.

Constructing an alignment extension is very much alike other DP algorithms (see Sec. 2.1.3). Consider a seed `d` on sequences `s1` and `s2` with begin and end positions `d.b1`, `d.b2`, `d.e1` and `d.e2`, then the extension to the right is the concatenation of the original seed alignment with a new alignment on `sr1 = s1[d.b1:end(s1)]` and `sr2 = s2[d.b2:end(s2)]`.[19] This new alignment shares properties of both global and local alignments, in that its begin position is fixed on the begin positions of `sr1` and `sr2`, but its end is "loose", as with local alignments. In its computation the right extension combines initialization and backtracking steps of the Needleman-Wunsch algorithm (Needleman and Wunsch, 1970) with the search for a maximum of the Smith-Waterman algorithm (Smith and Waterman, 1981). Extension to the left happens accordingly.

In order to not dispensively compute the full dynamic programming matrix after an alignment has degraded strongly, we added functionality similar to the original x-drop algorithm to SeqAn's dynamic programming code. By specifying an x-drop parameter, columns of the alignment will only be computed for as long as the current column maximum does not fall $x$ below the overall maximum seen. This algorithmic approach applies the x-drop paradigm only to the horizontal dimension, which means that theoretically more cells have to be computed; however a version with a fixed size band is also available to reduce the amount of these extra cells (see

[17] The computation for other cases is omitted, since larger seed distances did not prove advantageous.
[18] `http://www.inf.fu-berlin.de/inst/ag-bio-expired/file.php?p=ROOT/Teaching/Theses/Master/100,kemena.thesis.htm`
[19] End positions are one behind the last character in SeqAn.

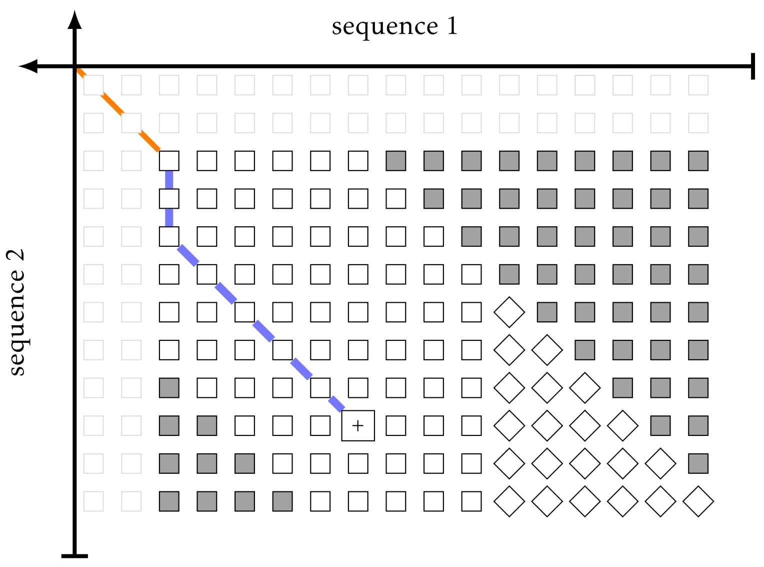


***Figure 3.5:*** *Banded right extension of alignment*

The seed alignment is orange, the right extension purple; white, solid cells are computed, grey cells are out of the band; tilted cells would have been computed, but were not, due to the x-drop; cell with + is the last and global maximum

Fig. 3.5). All of our tests showed that the new implementation was only negligibly slower than SeqAn's aforementioned x-drop implementation, while delivering up to 10% more valid results. This could be explained by the full x-drop being too opportunistic in following the current path and thereby missing alignments along the main diagonal or the choice of our band being more generous.

The alignment extension module is a core part of Lambda. It has already become part of the SeqAn library. Its parameters are important heuristic factors for the overall search in Lambda. The x-drop can be specified with `--xdrop` / `-x` and the upper diagonal of the band with `--band` / `-b` (the band then includes the main diagonal $\pm b$ and is of size $2b+1$). The parameter can be set as an absolute value, be dynamically computed from the query sequence's length (square root or base 2 logarithm of the length) or be deactivated. Choosing non-linear growth of the band accounts for the fact that the expected length of local alignments (and thereby the chance for gaps) does not grow linearly with sequence length. In fact, it grows logarithmically,[20] why this was chosen as default band parameter.

Other tools like NCBI BLAST and UBlast perform an ungapped extension with a smaller x-drop parameter before doing a gapped extension. BLAST even recomputes the entire alignment after both steps. This approach is intended to save unnecessary gapped extensions, which are the most costly, but it did not prove beneficial in Lambda. Very likely this is due to the optimized banded alignment being fast in the first place, and having to perform a second or even third alignment counterbalancing any positive effects on speed.

For the x-drop parameter a default of 30 was chosen, which is similar to UBlast(32) and BLAST (27, in its initial gapped extension).[21]

[20] Def.2.1 shows that the raw score influences the e-value exponentially, while the length is a simple factor; since the expected raw score grows linearly with length of the match (Altschul and Gish, 1996), the lengths of the matches need to increase logarithmically with the length of one of the sequences to maintain the same e-value.

[21] BLAST uses bit-score x-drops with 15 as default; a bit-score of 15.0086 corresponds to a raw score of 27 in the default scoring scheme.

### 3.5.3 Optimization

Since allocating memory for dynamic programming matrices is expensive, different steps where taken to further improve performance. Together with René Rahn versions of the DP algorithms were developed that can reuse memory and need not reallocate space at every iteration. This improved run-time in the extension phase up to 17%. Additionally, a custom local alignment function for cases of band size 1 (perfect or hamming distance seeding) was developed for Lambda. It comes without the overhead of the general implementation and decreases running time by another 22%.

The entire extension phase is parallelized per query sequence, processing the previously attained seed hits together. This makes post-precessing and output easier.

## 3.6 Output

After finding initial seeds, processing these and extending some to valid alignments, the results have to be written to disk. Before this is done they are sorted, duplicates purged and statistically insignificant ones dropped. For the latter BLAST-like statistical assessment is performed. As part of this work we also wrote support for BLAST output formats to achieve better comparability and interoperability with other software. These two components form the new SeqAn Blast module, which will soon become part of the library, as well.

### 3.6.1 BLAST statistics

The most widely used form of e-value statistics was implemented, i.e. the definitions are according to Sec. 2.1.2.[22] All blocks of extreme value distribution parameters where imported from the NCBI BLAST source code, and parameter selection happens automatically, so all combinations of scoring schemes and parameters supported by BLAST are supported. The bit-score calculations are in line with all other programs. The e-values differ from BLAST, but are very close to PAUDA and RAPSearch2; therefore it seems likely that the difference to BLAST is due to undocumented behavior in BLAST. An example of bit-scores and e-values calculated by the different programs can be found in App. B.3.

The default e-value cutoff in Lambda is 0.1, however this parameter is usually defined by the user, which can be done by setting `--evalue` / `-e`. More details on scoring and the choice of cutoffs in this work are available in Sec. 4.1

### 3.6.2 BLAST file formats

BLAST has twelve output file formats, which differ between versions of BLAST, and for different program modes. Most can also be customized further by user-set parameters, very few are standardized in any meaningful way, so that tools and pipelines concentrate on only a small subset of these file formats.

[22]This does not include changes made for BLAST+ (Camacho et al., 2009), since they are poorly documented, and it does not include conditional compositional score matrix adjustments (Altschul et al., 2005).

Two of the most portable, human- and machine-readable formats, and therefore probably also the most widely implemented, are the *tabular* format and the *commented tabular* format. These contain one match per line and different specifiers of each match in tab-separated columns, optionally with comment lines that have headers for the columns and contain some additional information. It can be defined by the user, which columns to show and in which order, but most programs only support the default set and order. An example of the output and the specifiers is available in App. B.3. The SeqAn Blast module offers support for writing the tabular format and the commented tabular format, of the latter, both the version of the traditional BLAST and BLAST+ are available. Support for reading the tabular formats and support for custom column setups (both for reading and writing) are also present in the library, however the latter cannot be set through Lambda's interface, yet.

The default output format in BLAST is called the *pairwise* format, because it contains the full alignments between all matching pairs of query and subject sequences. It is intended for good human readability and is still the preferred choice if the user needs the entire alignment, and not only the individual properties of the match. However machine readability of the format is very bad, and the BLAST developers have repeatedly stated that it is subject to unannounced and undocumented change. Due to this uncertainty no support for reading the format was added to the SeqAn Blast module, however support for writing it is available and output is very close to current versions of BLAST+ (see App. A.3).

The specifics of these formats in regard to the program mode (see Tab. 3.1) are all respected by the SeqAn Blast module and Lambda. Files created should be accepted as valid input to any parser which reads current output files of NCBI-Blast. In its support for multiple formats and its stronger adherence to the style set by NCBI-Blast for the pairwise format, Lambda is unique among the BLAST competitors tested.

# 4 Results

As with many heuristic approaches, it is difficult to define a "gold standard", and comparison with exact methods is often not possible or feasible, because these exact solutions do not exist or do not produce results in a reasonable amount of time. As we have previously explained, NCBI-BLAST is still the de-facto standard, and its statistical foundation seems to be widely accepted.

Therefore many other applications implement BLAST's statistical assessment of alignments and it is possible to compare results based on abundance and score. When we speak of sensitivity in the following, we refer to this relative measure, i.e. the quantity and quality of results in comparison with BLAST (and other applications that share its method of assessment). The absolute number of false negatives is never known and false positives do not exist in our model.

Since the answer depends strongly on the question asked, and comparison of different applications is not trivial, we will briefly explain the design and configuration of the benchmark we implemented, before continuing to the actual results.

## 4.1 Benchmark configuration

### 4.1.1 General notes

The purpose of our benchmark is to deliver evidence for or against using a particular piece of software in a particular scenario. Our interest is in maximizing the quantity of results, while maintaining the quality of alignments and minimizing the resource usage, most importantly running time. The quality of an alignment is indicated by either a bit-score or an e-value, of which the latter takes sequence lengths into account and the former doesn't. Most metagenomic studies recommend an e-value cutoff of $10^{-6}$ to $10^{-4}$ (Lamendella et al., 2011; Wommack et al., 2008; Mackelprang et al., 2011), but some go as low as $10^{-16}$ (Tetu et al., 2013) or $10^{-20}$ (Eikmeyer et al., 2013), so this is the general range that we will investigate. Unfortunately, although e-value calculation is widely adopted and accepted, different programs seem to be unable to reproduce BLAST's e-values, with results (for identical alignments) varying up to factor 1,000.[1] Due to changes in the statistical model even e-values calculated by BLAST and BLAST+ are not equal. This is not true for bit-scores, which are very close for identical alignments (see App. B.3 for an example).

Since the only statistical difference between bit-scores and e-values is the consideration of the total length of subject sequences – which is constant for each benchmark – and the length of the current query sequence – which only varies a little throughout each benchmark[2] – we chose to use a minimum bit-score instead of a maximum e-value as cutoff. The minimum bit-score selected was the average bit-score of all BLAST+ results that yielded an e-value between $10^{-6}$ and $10^{-5}$.

[1]This is independent of position-specific scoring matrices (PSSMs), which also influence e-value calculation in BLAST, but which were deactivated in this benchmark, since no other program implements them.

[2]Variance in query length is due to quality-based trimming of reads prior to publication.

This bit-score cutoff was applied to the results a posteriori, after an initial parameter of $10^{-1}$ was set as e-value cutoff for all programs in the benchmark. This ensures that no valid results are lost early on in the benchmark, and comparability is maintained, even if a longer-than-average query sequence (that would have a higher e-value) is encountered. None of the programs showed different run-times based on the target cutoff, so no discrimination should arise from applying a lower cutoff a posteriori.

| | |
|---|---|
| TotalResults | all results that score above the minimum bit-score |
| MatchedQueries | only one (the best) match per query |
| Recall | match between same query-subject pair as in BLAST+ |
| $\approx$ | with similar bit-score ($\pm 10\%$) |
| $>$ | with superior bit-score ($\geq +10\%$) |
| $<$ | with inferior bit-score ($\leq -10\%$) |
| $*$ | match between different query-sequence pair than BLAST+ |
| $\emptyset$ | match of BLAST+ missed |

***Table 4.1:*** *Categorization of results*

Different indicators can be used to define sensitivity, of which the total number of valid results is the most basic and maybe obvious one. It is relevant to research where all or up to $n$ matches per query are desired. But it is more susceptible to disturbance, e.g. a single query sequence that produces many hits might skew the results; or a program could gain an advantage by producing two (shorter) alignments from one region, although it optimally contains one (longer) one.

In sequence classification (and many other use-cases) we are only interested in the single best result for every query sequence (by definition we are looking for the closest matching sequence), so most approaches use the number of matched queries as the critical indicator (Huson and Xie, 2013; Ye et al., 2011).

Since we have defined BLAST as the gold standard, we will want see if other programs actually recall BLAST's hits, or if they classify the query sequence as something entirely different by assigning it to a different subject. Arguably, a query assigned to a different subject sequence does not necessarily mean that the match is inferior. However, no other tool to date has shown to be significantly more sensitive than BLAST, and none of the compared tools claims to be, so it is a fair assumption that a high BLAST recall is a predictive indicator of a high sensitivity. This is also theoretically founded by BLAST having a lower effective seed length, higher x-drop values and more steps of realignment (see Chap. 2).

The recalled matches are further categorized by the benchmark depending on their bit-score. These categories demonstrate whether the alignments are of comparable quality, which would be an indicator for the ability to reproduce results at more stringent cutoffs, and a general clue on the cababilities of the algorithms. For completeness, we give all indicators as displayed in Tab. 4.1, and we consider the number of matched queries most important, followed by the recall, its qualities and the amount of total results.

For further assessment of the results, the quartiles of the bit-score distribution and the median sequence identity are also calculated for the best-per-query results. In the end, all of this information should indicate which program is the most sensitive and which is the fastest, as well as which might be more suited under certain constraints.

### 4.1.2 Implementation notes

The benchmark is implemented as UNIX script, interpreted by the Bourne shell and GNU Awk[3]. It is designed in modular way, where applications can be easily added by providing a plugin, in form of a script that defines paths to the application, command line options and optionally custom pre- and post-processing. Run-time is measured by querying system time and memory usage is measured by periodically querying the process file system (`/proc`) for the virtual memory resident set size (VmRSS) of the target process and its children. The latter approach covers many cases, where programs spawn different sub-processes, but it doesn't cover all cases, e.g. PAUDA spawns different java-processes that are difficult to track. For PAUDA 20GiB appear as an upper bound, because this is the maximum reserved by the Java Virtual Machine; the publication states that memory usage is up to 16GiB.

Beside the input files, the benchmark can be passed different cutoff values, as well as the number of threads all applications shall be limited to (where they exist, the benchmark sets an application specific parameter, otherwise it expects the application to choose an adequate value automatically).

### 4.1.3 Applications, parameters & hardware

**Hardware** All tests were conducted on a Debian GNU/Linux 7.1 system[4], with 2× Intel® Xeon® X5650 CPUs @ 2.67GHz (a total of 12 physical and 24 virtual cores) and 142GiB of RAM. All temporary data, intermediate data, and both input and output files were read from and written to a *tmpfs*, i.e. a virtual filesystem in main memory. This prevents disk-caching effects from disturbing the benchmark, and increases overall performance. The latter effect is stronger on IO-heavy tools, but since memory is a very small cost-factor in bioinformatics-pipelines, we recommend this approach for general use, as well.

**General Parameters** All programs were run with twelve threads and an e-value cutoff of 0.1 (see above). For OpenMP-based tools the benchmark binds the twelve threads to fixed physical CPU cores to prevent contention between two virtual cores for the same physical core. Some tools contain smaller post- or pre-processing steps not explained below, see Appendix B.2 for full command lines.

**BLAST** The "traditional" NCBI-Blast, written in C, Version 2.2.26 (Altschul et al., 1997, blastall executable). The database was *hard-masked* (see Sec.3.4.2) with the standalone SEG program (Wootton and Federhen, 1993) before execution and BLAST's SEG-Filter (that works on the query) was deactivated (`-F F`). As previously noted, composition-based statistics were deactivated, since no other tool implements them (`-C 0`). Output format is set to tabular, no headers (`-m 8`).

**BLAST+** The new version of NCBI-Blast, partly written in C++, version 2.2.27+[5](Camacho et al., 2009, blastx executable). The database was soft-masked with the standalone segmasker program, a newer implementation of SEG, and BLAST+'s internal SEG-Filter (that works on the query) was

[3]`http://www.gnu.org/software/gawk/`
[4]`http://www.debian.org`
[5]Note that both versions of BLAST received updates for a certain period of time and identical version numbers for releases of either are in use.

deactivated (`-seg no`). Composition-based statistics were deactivated (`-comp_based_stats 0`), and output set to tabular (`-outfmt 6`). This version is used for computing the cutoffs and as main reference, because it is the current version and recommended by the NCBI.

**PAUDA** Pauda was used in version 1.0.1. Pauda has two modes of operation, either with parameter `--slow` or `--fast`; both are benchmarked, titled $\text{PAUDA}_{\text{sl}}$ and $\text{PAUDA}_{\text{fa}}$, respectively. Since PAUDA implements no form of masking, the input database was hardmasked with SEG. The Bowtie2 version used in conjunction with PAUDA was 2.1.0.

**RAPSearch2** Pre-compiled 64bit-binaries of RAPSearch2, version 2.09, were used for the benchmark.[6] Unfortunately it was not possible to deactivate masking of the query and it was unclear whether the database is masked. We do not expect the latter, because it is not the default for other programs, but we still decided not to mask the database prior to running RAPSearch2, to "counter" possible effects of the query masking. Since the database is larger in our tests than the sum of the query sequences, this might still influence RAPSearch2's measured sensitivity in its favor. Should the database have been masked internally, it could incur a slight sensitivity penalty for RAPSearch2.

**UBlast** Only 32Bit-binaries (and no source code) of UBlast are freely available, why benchmarks were conducted with these. Initially, version 7.0.1001 was tested, but since this produced very poor results (see Sec. 4.3), we chose to additionally include version 6.0.307. The database is soft-masked during creation (`-dbmask seg`) and masking of the query is deactivated when running UBlast (`-qmask none`). UBlast advertises the `-accel` parameter as a convenient way to tune sensitivity, so we chose to benchmark both, the default (0.8 according to the manual) and with `-accel 1`, which should be slower, but maximally sensitive. The tags for the benchmark are $\text{UBlast6}_{\text{sl}}$ (slow), $\text{UBlast6}_{\text{def}}$ (default), $\text{UBlast7}_{\text{sl}}$ and $\text{UBlast7}_{\text{def}}$.

**Lambda** Benchmarks were conducted using the current development version of Lambda. The `lambda_indexer` reads intervall-files generated by segmasker and soft-masks the database. No masking is performed on the query. The parameter space will be explored in depth, in the next sections. The benchmark will include three profiles, $\text{Lambda}_{\text{sl}}$ (slow), $\text{Lambda}_{\text{def}}$ (default) and $\text{Lambda}_{\text{fa}}$ (fast).

### 4.1.4 Datasets

| | DATASET I | DATASET II |
|---|---|---|
| Origin | Alaskan Permafrost | Sargasso Sea |
| Purpose | metagenomics | metagenomics |
| average read length | ~ 100bp | 818bp |
| No. of reads | 13,834,460 | 1,982,807 |
| No. of reads selected | 1,000,000 | 100,000 |
| minimum bit-score | 42.0695 | 48.2798 |

***Table 4.2:** Properties of the two datasets*

minimum bit-score as described in Sec. 4.1.1

[6] At a late point in writing this thesis we have become aware of a new release of RAPSearch2, which should be included in future benchmarks.

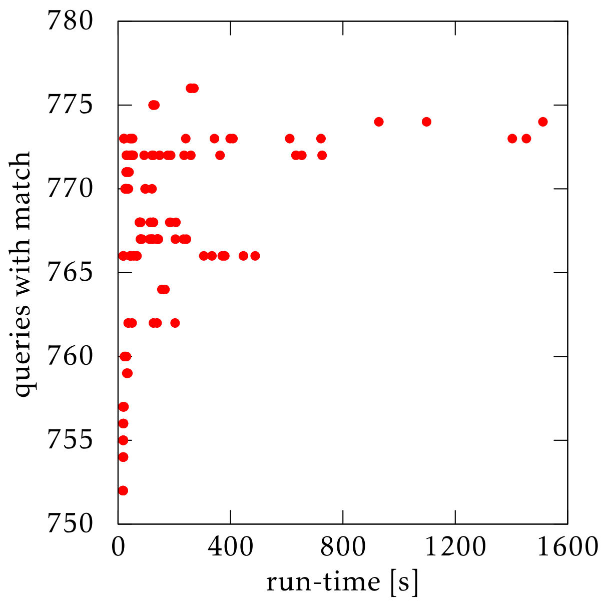
For Illumina reads, we tried to use the read data from Mackelprang et al. (2011), which is the reference dataset used by PAUDA (Huson and Xie, 2013). Unfortunately the server referenced in the publication's supplementary information did not contain the raw read data, so a dataset by the same group (Janet K. Jansson), also from Alaskan permafrost, was chosen. It was downloaded from the Earth Microbiome Project[7], its properties are displayed in Tab. 4.1.4.

The second dataset was retrieved from the servers of the J. Craig Venter Institute[8], it is the dataset presented in one of the most highly cited, early metagenomic studies (Venter et al., 2004). The reads were sequenced with Sanger's method. Since Sanger sequencing and 454 sequencing produce reads of similar accuracy and lengths (Liu et al., 2012), this dataset should have informative value for both technologies.

We chose these two datasets to have representatives of the different sequencing approaches, especially different read lengths – as some programs might be adapted better to one use case. The difference in datasets also helps to prevent over-optimization of heuristic parameters. The database of subject sequences used for all runs was UniProtKB Swiss-Prot.[9]

[7] http://www.microbio.me/emp/
[8] https://moore.jcvi.org/sargasso/
[9] http://www.uniprot.org/

## 4.2 Lambda's parameter space



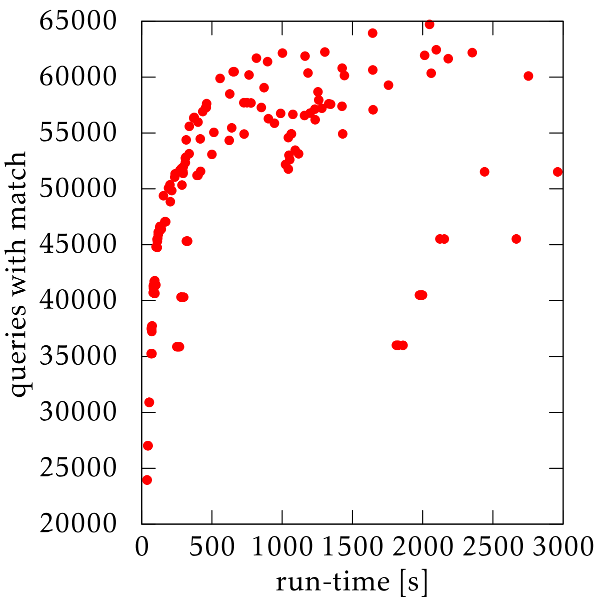


***Figure 4.1:*** *Matched queries vs. run-time (dataset I left, dataset II right)*

During the development of Lambda different parameter ranges were explored and the space narrowed down to a sensible sub-space. In this subsection we will present the different parameters and their influence on the results. For the seed length values of 6 to 12 were benchmarked, but values of 6 and 7 were only part of certain parameter combinations (as explained later). Seeds were searched with zero errors, hamming distance 1 or edit distance 1; longer distances did not proof to feasible. The unreduced alphabet ("Unred.24"), Murphy10 and the three Lamba-alphabets (8, 10 and 12) discussed in Sec. 3.3 were compared. As key heuristic parameter the minimum seed score was also analyzed, with values ranging from 3.0 over 3.25 to 3.5 times the seed length.

These parameter combinations yielded a search space of over 200 combinations, whose results we will present in the following sections. Some of these combinations exceeded the memory requirements of the test hardware. While we do offer a solution for these situations with query partitioning (as described in Sec. 3.4.3), the optimal query partitioning depends strongly on the parameter set, and a more exhaustive search will have to reveal sensible values. For the question at hand we decided that it seems unlikely that parameter combinations with memory demands of over 100GiB will be sufficiently fast to qualify for default parameters, considering that query partitioning in itself incurs a performance penalty and a higher memory usage indicates more hits that need to be verified. Thus, query partitioning was only set to perform optimal parallelization; runs that exceeded memory requirements were dropped from the analysis.

Fig. 4.1 shows the overall distribution of the most important results of Lambda, i.e. amount of queries with hits versus running time on both datasets. The graphs in Fig. 4.1 (and all of the following) are cropped on the x-axis, see App. B.4 for a full-scale version. It is evident that the run-time varies heavily with the choice of parameters and that at least for dataset II there is also a strong variance in sensitivity. Since our objective is to minimize running time and maximize results, we will want to have a closer look on the top left region of both plots and identify which parameters are responsible for these results. Ideally we will find a set of parameters that performs well for both datasets and can be chosen as a generic default for metagenomic analysis with Lambda.

But before we go there we will try to bring more order to the results and analyze the influence of individual parameters.

### 4.2.1 Alphabets

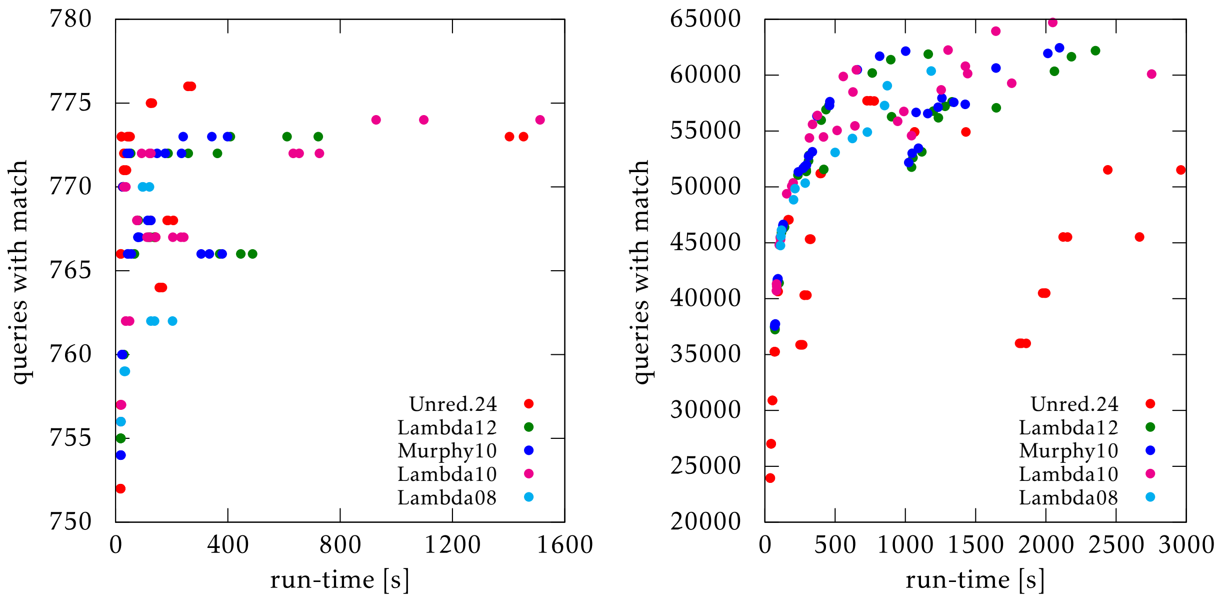


***Figure 4.2:*** *Influence of alphabets (dataset I left, dataset II right)*

Based solely on the alphabet no clear pattern can be deduced for the datasets, i.e. no alphabet shows unambiguous benefits or drawbacks. In fact, the unreduced alphabet is among the best in dataset I, but performs poor for dataset II. To judge alphabets, a more fine-grained analysis, i.e. with other parameters fixed, has to be performed.

| Alphabet | Total results | matchedQueries | bitScore | % Ident | Performance | |
|---|---|---|---|---|---|---|
| | | | Median | Median | time[s] | mem[MiB] |
| Unred.24 | 1981 | 766 | 64.69 | 100 | 19 | 4793 |
| Lambda12 | 2188 | 766 | 64.69 | 100 | 67 | 5363 |
| Murphy10 | 2271 | 766 | 64.69 | 100 | 43 | 5553 |
| Lambda10 | 2233 | 767 | 64.69 | 100 | 121 | 11353 |

***Table 4.3:*** *Sensitivity and performance in relation to choice of alphabet (dataset I)*
(seed length 9, exact seeds, min. seed score 29; Lambda08 exceeded mem usage)

In dataset I and at seed length 9 we do not see much difference for the number of matched queries, although an increase in overall hits can be seen with decreasing alphabet size (Tab. 4.3). The distribution of queries with hits also seems to be identical or close as indicated by the identical bit-score and percent-identity medians. With decreasing alphabet size memory usage and running time increase, although Murphy10 is notably faster than Lambda12. Lambda10 matches one more query than the others, but yields less total results than Murphy10.

| Alphabet | matchedQueries | bitScore | | | % Ident | Performance | |
|---|---|---|---|---|---|---|---|
| | | Quart1 | Median | Quart3 | Median | time[s] | mem[MiB] |
| Unred.24 | 30895 | 167.93 | 261.53 | 372.85 | 69.33 | 55 | 4792 |
| Lambda12 | 46273 | 130.56 | 213.38 | 320.08 | 58.10 | 131 | 5059 |
| Murphy10 | 46549 | 130.95 | 213.38 | 320.08 | 58.96 | 128 | 5045 |
| Lambda10 | 50070 | 124.40 | 204.14 | 310.84 | 56.73 | 191 | 5681 |
| Lambda08 | 54337 | 117.08 | 193.74 | 300.82 | 54.39 | 623 | 36636 |

***Table 4.4:*** *Sensitivity and performance in relation to choice of alphabet (dataset II)*
(seed length 10, exact seeds, min. seed score 32)

From Tab. 4.4 we can see an obvious trend for dataset II on the number of queries with matches, too, it increases significantly with decreasing alphabet size (total results accordingly, see Tab. 4.5, col. 4). The bit-score quartiles underline that mostly low-scoring hits are missing in larger alphabets and the median percent-identity shows that the relation between high scoring alignments and conservation. Sensitivity increases by $\sim 10\%$ from Lambda12 to Lambda10 and from Lambda10 to Lambda08, however the performance cost (both in terms of run-time and memory usage) of the latter step is dramatic.

| Alphabet | SeedHits | | | Time | |
|---|---|---|---|---|---|
| | total | extended | valid | Seeding | Extension |
| Unred.24 | 3,354,082 | 3,354,082 | 2,085,031 | 7.65s | 36.60s |
| Lambda12 | 24,423,806 | 11,756,525 | 4,831,599 | 10.11s | 107.59s |
| Murphy10 | 19,340,375 | 11,146,773 | 4,784,205 | 9.57s | 106.69s |
| Lambda10 | 67,687,890 | 17,447,699 | 5,708,901 | 11.89s | 165.33s |
| Lambda08 | 1,199,654,172 | 37,610,043 | 6,855,578 | 82.98s | 469.99s |

***Table 4.5:*** *Ratios of extended hits and times spent in seeding and extension (dataset II)*
(seed length 10, exact seeds, min. seed score 32)

A deeper look at Lamda's log-files reveals the time spent in the different phases of the program (Tab. 4.5). While total valid results of the Lambda alphabets increase by $\sim 20\%$ with each reduction step of 2, the amount of seed-hits rises more steeply, with a near twenty-fold jump between Lambda10 and Lambda08. This is directly reflected in the time spent in seeding, although the influence is clearly not linear. That the time in seeding is affected at all, hints at the cost of caching and post-processing the hits (since the search in itself obviously does not depend on the amount of results).

As explained in Sec. 3.5.1 not all seed-hits are extended, only those that pass a score test after local alignment (more on this and the relation between the number of extensions and the number of valid results also in Sec. 4.2.3). However this local alignment computation is still considered part of the "extension phase". Column 3 in Tab. 4.5 shows the number of actual extensions per run, and here we see an almost linear correlation to the time spent in the extension phase. An influence of the local alignment computation can be seen especially for Lambda08, since its time in the extension phase is higher than its number of actual extensions would warrant (but its count of local alignments is high).

Altogether we can see that sensitivity, but also resource usage rises with a smaller alphabet size and that Lambda08 is especially costly. Murphy10 seems to be closer to Lambda12 than to Lambda10 from both, its sensitivity and performance. In some cases it has both, better results and better performance than Lambda12. Sensitivity of the unreduced alphabet seems insufficient for dataset II.

### 4.2.2 Seed lengths and distances

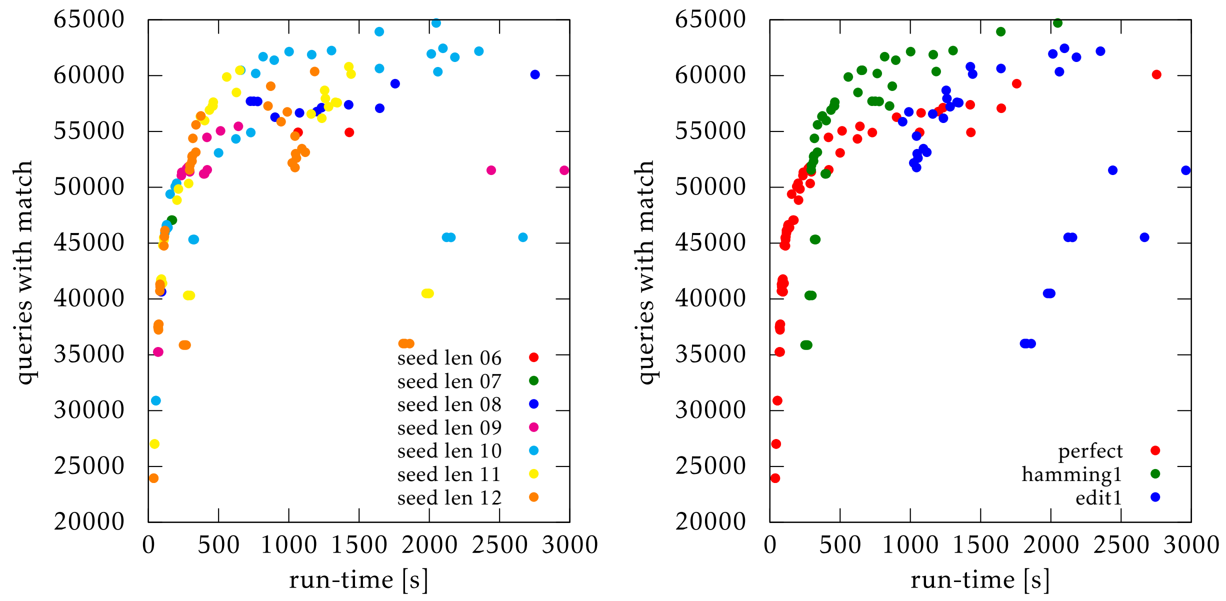


***Figure 4.3:*** *Influence of seed lengths and distances (dataset II)*

The influence on results by chosen seed length can be seen in the left half of Fig. 4.3. It seems that seed lengths 10 and 11 are the most abundant in the top left region of the plot, with seed length 10 clearly in the lead sensitivity wise. Smaller seed lengths are present, but less favored with good results. It should be noted that especially for seed lengths 6 and 7 only runs with the unreduced alphabet completed, because other combinations ran out of memory. Some results, especially with shorter seed lengths had run-times that placed them outside the clipped area of the plot.

The information on the seed distances' influence on results is the most unambiguous, yet, with hamming distance 1 clearly dominating the target area (second plot in Fig. 4.3). Perfect seeding has a few speed advantages in lower sensitivity areas, but seeding with an edit distance of 1 seems to have only disadvantages.

| Parameters | | matchedQueries | | bitScore | | % Ident | Performance | |
|---|---|---|---|---|---|---|---|---|
| sl | dist | | Quart1 | Median | Quart3 | Median | time[s] | mem[MiB] |
| 10 | 0 | 46549 | 130.95 | 213.38 | 320.08 | 58.96 | 128 | 5062 |
| 10 | h1 | 61695 | 109.38 | 179.10 | 283.10 | 51.94 | 817 | 22582 |
| 10 | e1 | 61948 | 108.61 | 178.33 | 282.72 | 51.15 | 2014 | 34135 |
| 11 | 0 | 41719 | 140.19 | 226.86 | 333.95 | 60.54 | 89 | 4698 |
| 11 | h1 | 57280 | 115.93 | 188.34 | 293.12 | 53.33 | 460 | 6483 |
| 11 | e1 | 57571 | 115.16 | 187.57 | 292.73 | 53.76 | 1345 | 9012 |

***Table 4.6:*** *Sensitivity and performance in relation to seed lengths and distances (dataset II)*

(Murphy10 alphabet, min. seed score 3.25×seed length)

Following up on the last paragraph and the last section, seed lengths 10 and 11 and Murphy10 as alphabet were selected for a more in-depth analysis. The results, with more precise information of the parameters are available in Tab. 4.6. The trends in this table are increasing sensitivity and increasing computational cost with increasing error tolerance, and vice versa with the seed length. It should come as no surprise that especially lower scoring matches, with lower sequence identity are missed by exact seeding (bit-score and percent identity-columns).

Within a constant seed length, allowing one substitution per seed increases sensitivity by over 30%, but also increases running time by factor 5 to 6. For seed length 10 memory usage quickly rises to over 20GiB, this is not the case for seed length 11. Allowing for the error to be an indel also increases running-time by another factor of ∼ 2.5 and memory usage by ∼ 1.5, however the gain in sensitivity is < 1%.

When introducing errors, the lead in sensitivity of seed length 10 over seed length 11 schrinks from over 11% to ∼ 7%, but is still significant. If we take the total number of results into account (Tab. 4.7, col. 4) the difference remains between 25% and 23%.

| Parameters | | SeedHits | | | Time | |
|---|---|---|---|---|---|---|
| sl | dist | total | extended | valid | Seeding | Extension |
| 10 | 0 | 19,340,375 | 11,146,773 | 4,784,205 | 9.57s | 106.69s |
| 10 | h1 | 633,717,665 | 81,291,675 | 10,523,732 | 225.29s | 547.56s |
| 10 | e1 | 1,073,831,122 | 88,731,915 | 10,594,636 | 1094.51s | 858.43s |
| 11 | 0 | 7,875,683 | 6,736,254 | 3,811,401 | 8.39s | 68.62s |
| 11 | h1 | 107,689,218 | 25,296,093 | 8,502,658 | 170.15s | 273.65s |
| 11 | e1 | 177,706,595 | 26,671,282 | 8,580,831 | 999.41s | 326.59s |

***Table 4.7:*** *Ratios of extended hits and times spent in seeding and extension (dataset II)*
(Murphy10 alphabet, min. seed score 3.25×seed length)

Tab. 4.7 gives more clues to previous observations: While going from hamming distance to edit distance has only neglegible benefits in sensitivity the amount of initial seed hits nearly doubles. Interestingly the time spent in seeding rises much more strongly (factor 5-6), hinting that it is not post-processing and caching of seeds that is responsible at this point, but indeed the search algorithm, that is significantly slower when increasing the error tolerance. In the extension phase we see an almost linear correlation to the actual amount of performend extension for both seed lengths and all parameter sets that do not involve edit distance seeding. For the latter the increase is significantly higher than linear, likely an effect of the previously mentioned local alignment computations of which there are more. But not only more have to be computed, the computation is more costly per se, since – as noted in the implementation section – the local alignment step for edit-distance seeds requires the computation of three diagonals instead of one.

Of the extra hits attained through edit distance seeding only a small fraction is actually extended, hinting at an explanation for the little gain in sensitivity. With edit distance seeding the ratio of time spent in the seeding phase and in the extension phase effectively tips towards seeding.

Interesting in this context is a fact observable from Fig. 4.4 on p.39: A seed length of 10, with allowed hamming distance of 1 seems to be significantly more sensitive than a read length of 8 without allowed errors. On the unreduced alphabet, which also has results for seed length 7 without allowed errors, this was not notably more sensitive than seed length of 10 with allowed hamming distance 1.

Since the variation in sensitivity is less pronounced in dataset I, some of the effects described in this section are less obvious there, however the trends, especially in performance are the same.

### 4.2.3 Minimum seed score

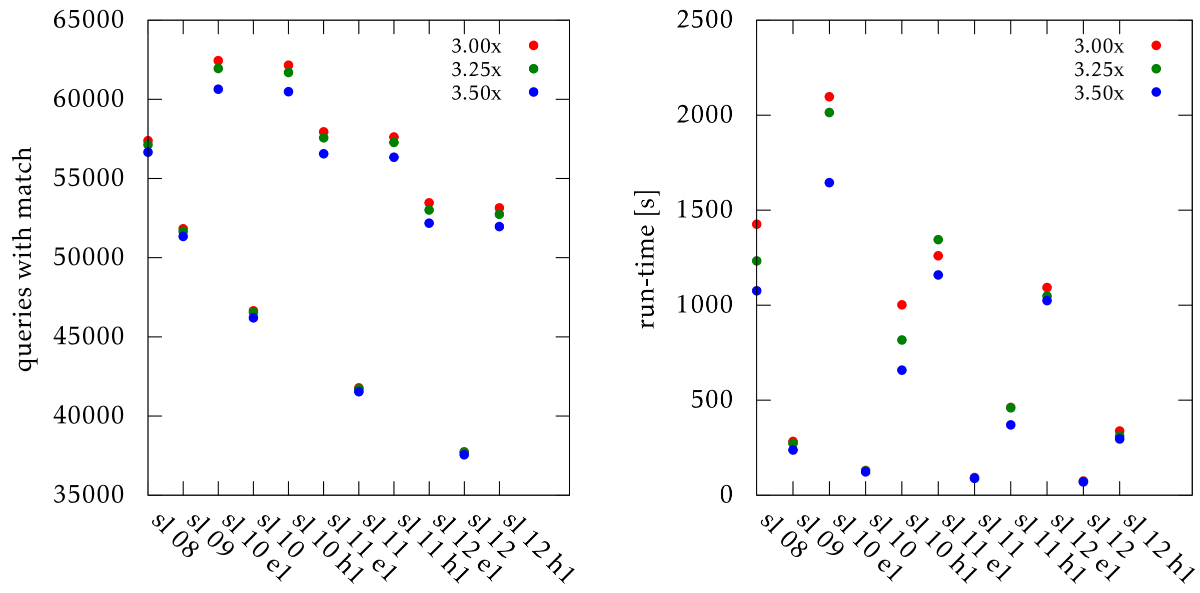


***Figure 4.4:*** *Influence of seed min score (dataset II)*
(Murphy10 alphabet, configuration on x-axis: sl = seed-length, h1=hamming distance 1, e1 = edit distance 1)

The heuristic parameter identified as central in Lambda's pipeline is the minimum seed score, it decides whether a seed will be extended or dropped before extension. As we have seen in the last sections the amount of extensions performed is the major factor influencing run-time for most parameter sets, but we also depend on not dropping the good candidates if we are to achieve a decent sensitivity. Empiric testing revealed a value of 3.25× the seed length to be a good balance between speed and sensitivity. Exempting the Unknown-character 'X', all amino acids have a self-score of ≥ 4 in the Blosum62 matrix, so an exact match always contributes to extending a seed. This also means, that the parameter does not effect seeding on the unreduced alphabet, when no errors are allowed.

| min. | | SeedHits | | matched | Time | |
|---|---|---|---|---|---|---|
| SeedScore | total | extended | valid | Queries | Seeding | Extension |
| 30 | 633,717,665 | 120,473,078 | 10,723,411 | 62,152 | 229.49s | 731.23s |
| 32 | 633,717,665 | 81,291,675 | 10,523,732 | 61,695 | 225.29s | 547.56s |
| 35 | 633,717,665 | 44,774,629 | 10,019,772 | 60,485 | 234.81s | 381.19s |

***Table 4.8:*** *Influence of the minimum seed score on number of extensions and performance (dataset II)*
(Murphy10 alphabet, seed length 10, seed distance h1)

Fig. 4.4 additionally shows factors of 3 and 3.5 to demonstrate the influence of the parameter. Evidently seeds with errors are effected more strongly; a difference between hamming and edit distance is present, but not notable. Important is the observation that the effect on performance is much stronger than on sensitivity, with some runs differing up to 60% in run-time between factor 3.0 and factor 3.5, while the difference in sensitivity for the same runs is less than 4%. Peculiarly we see a longer running time for a higher minimum seed score in run `sl11 e1`. Tab. 4.8 gives a more in-depth look on the effect of the minimum seed score, both on sensitivity and performance.

Since matches in dataset I have a very high sequence identity higher values of the minimum seed score could be chosen, to perform better in that case. However, this is posterior knowledge and does not help find generic defaults for the application.

### 4.2.4 Selection of defaults

| Profile | Alphabet | SeedLength | Dist | MinSeedScore |
|---|---|---|---|---|
| fast | Unred.24 | 8 | 0 | 26 |
| default | Murphy10 | 10 | h1 | 32 |
| slow | Lambda10 | 10 | h1 | 32 |

***Table 4.9:*** *Recommended parameter selections*

Based on the previous analysis of Lambda's parameter space we chose three profiles to compete against the other programs and as sensible user choices (Tab. 4.9). The default profile is based on the best speed/sensitivity trade-offs found in the last sections. The fast profile uses an unreduced alphabet and allows no errors in the seed, it is meant for situations where only very high-scoring matches are desired or resources do not permit a more sensitive search. Finally the slow profile is a similar, but more sensitive alternative to the default. It makes use of the Lambda10 alphabet.

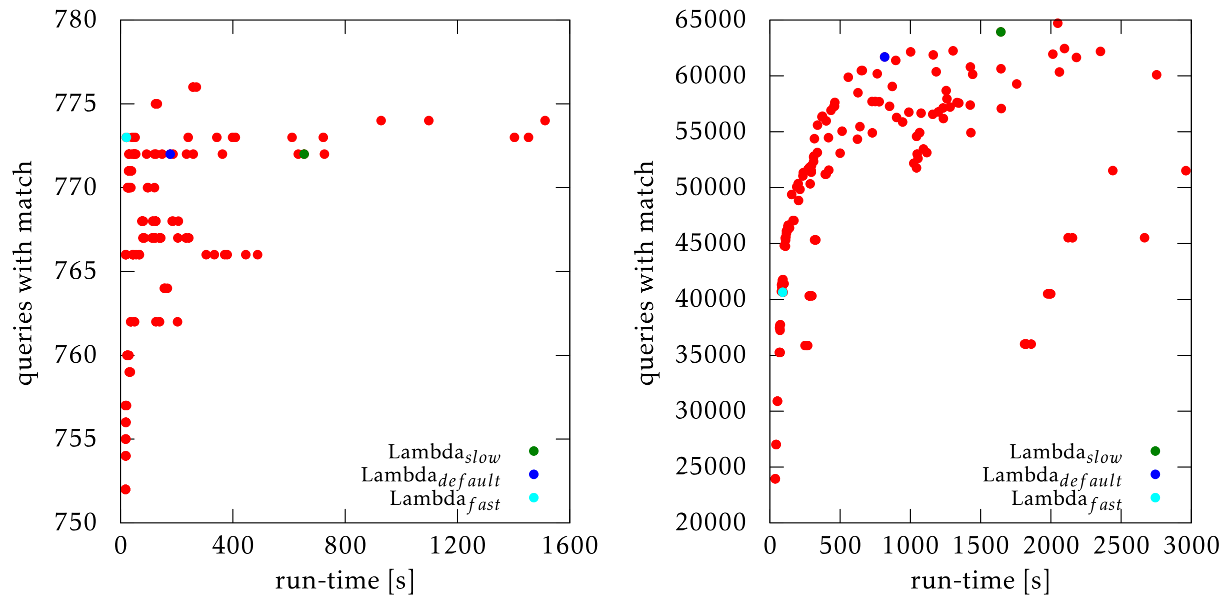


***Figure 4.5:*** *Recommended profiles and their location in the results (dateset I left, dataset II right)*

As can be seen in Fig. 4.5 the parameter selections represent points on the "outer curve" in dataset II, i.e. at the given level of sensitivity no other parameter combination is faster. For dataset I this is less clear, there are parameter combinations that are faster and/or are more sensitive, however variation in sensitivity is below 1% in the entire target region, and better scoring combinations in dataset I produced worse results on dataset II.

In the next section we will see how these profiles fare against BLAST and the other programs.

## 4.3 Comparison of programs

### 4.3.1 Dataset I

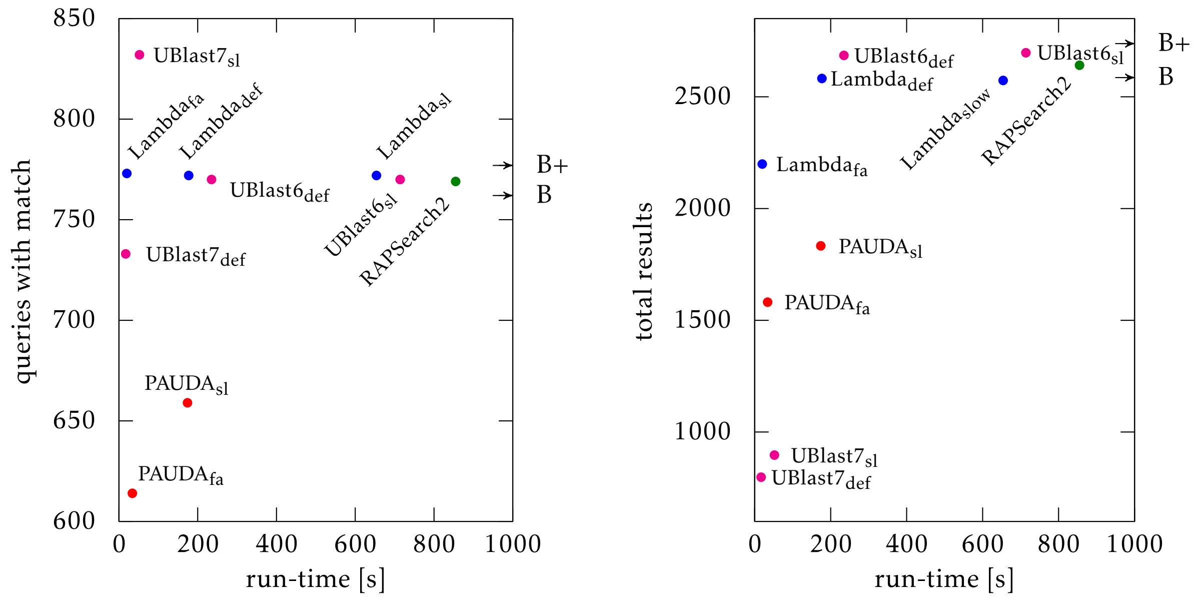


***Figure 4.6:*** *Comparison of applications (dataset I)*
BLAST and BLAST+ are far outside the displayed area, their y-coordinates are given on the right sides of the plots, as B and B+, resp.

Fig. 4.6 gives us the most important results on the left side, with the number of queries that have been assigned at least one valid match plotted against the running time. Sensitivity-wise it appears that UBlast7 comes out in front, with Lambda, UBlast6, RAPSearch2 and BLAST(+) next and PAUDA measurably behind. The less important overview over the total results (Fig. 4.6, right side), gives a similar picture, in that Lambda, UBlast6, RAPSearch2 and BLAST(+) group together sensitivity-wise (with the exception of Lambda$_{fa}$ and with a little more variance than in the left plot) and PAUDA fares a little bit better in comparison, but still significantly behind the others; UBlast7 comes out last however, which is an obvious difference to the first plot. Performance-wise Lambda$_{fa}$, Lambda$_{def}$, UBlast7 and PAUDA share the lead, with UBlast6 not far behind.

| Program | matchedQueries | | bitScore | | | % Ident | Compared to BLAST+ | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | abs | of BLAST+ | Q1 | Median | Q3 | Median | ≈ | > | < | * | ∅ |
| BLAST | 762 | 98.07% | 58.5 | 63.5 | 68.9 | 100.00 | 713 | 0 | 16 | 33 | 48 |
| BLAST+ | 777 | 100.00% | 59.3 | 64.7 | 68.9 | 100.00 | 777 | 0 | 0 | 0 | 0 |
| PAUDA$_{fa}$ | 614 | 79.02% | 62.8 | 66.6 | 70.5 | 100.00 | 314 | 7 | 1 | 292 | 455 |
| PAUDA$_{sl}$ | 659 | 84.81% | 61.6 | 65.9 | 69.3 | 100.00 | 394 | 5 | 2 | 258 | 376 |
| RAPSearch2 | 769 | 98.97% | 59.7 | 64.7 | 68.9 | 100.00 | 748 | 0 | 8 | 13 | 21 |
| Lambda$_{fa}$ | 773 | 99.49% | 59.3 | 64.3 | 67.3 | 100.00 | 756 | 0 | 0 | 17 | 21 |
| Lambda$_{def}$ | 772 | 99.36% | 59.3 | 63.5 | 68.6 | 100.00 | 768 | 0 | 0 | 4 | 9 |
| Lambda$_{sl}$ | 772 | 99.36% | 59.3 | 63.5 | 68.6 | 100.00 | 768 | 0 | 0 | 4 | 9 |
| UBlast6$_{def}$ | 770 | 99.09% | 59.7 | 64.7 | 68.9 | 100.00 | 769 | 0 | 1 | 0 | 7 |
| UBlast6$_{sl}$ | 770 | 99.09% | 59.7 | 64.7 | 68.9 | 100.00 | 769 | 0 | 1 | 0 | 7 |
| UBlast7$_{def}$ | 733 | 94.34% | 43.1 | 44.3 | 46.2 | 33.60 | 0 | 0 | 0 | 733 | 777 |
| UBlast7$_{sl}$ | 832 | 107.08% | 42.7 | 43.1 | 46.2 | 32.40 | 0 | 0 | 0 | 832 | 777 |

***Table 4.10:*** *Sensitivity of the different programs in detail (dataset I)*
For an explanation of the last 5 columns, see Tab. 4.1 on p.30

A more detailed overview of the sensitivity is available in Tab. 4.10. One of the most striking observations is that UBlast7 seems to produce erroneous results, or if not, at least very poor ones. This is visible in the bit-score distribution that reveals 75% of the results to score *below* a bit-score of 46.2, while for all other tools 75% of the results score *above* 59 bits. Also the median sequence identity is 100% for all other tools and only $\sim 33\%$ for UBlast7. It reproduces none of BLAST+'s matches. We therefore chose to discard UBlast7's results in all further analysis of this dataset.

In general, sensitivity is close to the observations from Fig. 4.6: Lambda, UBlast6 and RAPSearch2 produce $\sim 99\%$ of the amount of queries with matches, that BLAST+ produces and Lambda is in the lead between the three. Different profiles seem to have little influence on the amount of assigned queries and the distributions of bit-scores are very similar, also compared to BLAST+.

The amount of actual BLAST+ matches that is recalled (sum of $\approx$, $>$ and $<$ columns) varies slightly, with the default and slow profiles of Lambda and UBlast6 recalling $\sim 99\%$ of BLAST+'s matches (UBlast6 marginally better than Lambda); and Lambda$_{fa}$ and RAPSearch2 close behind, with $\sim 97\%$ and $\sim 96\%$ respectively. Curiously the traditional BLAST performs worse than these three programs compared to BLAST+, both, in terms of amount of matches and recalls.

PAUDA is far behind in all aspects, it produces only 79% and 85% of the amount of queries with matches and in total only recalls 41% and 51% of BLAST+'s results.

| Program | runTime[s] | rel. speedUp | memUsage[MiB] |
|---|---:|---:|---:|
| BLAST | 46815 | 1.07x | 198 |
| BLAST+ | 50091 | 1.00x | 225 |
| PAUDA$_{fa}$ | 34 | 1437.26x | $\leq 20000$ |
| PAUDA$_{sl}$ | 174 | 287.88x | $\leq 20000$ |
| RAPSearch2 | 855 | 58.59x | 1735 |
| Lambda$_{fa}$ | 20 | 2504.55x | 4989 |
| Lambda$_{def}$ | 177 | 283.00x | 13099 |
| Lambda$_{sl}$ | 654 | 76.59x | 65139 |
| UBlast6$_{def}$ | 235 | 213.15x | 902 |
| UBlast6$_{sl}$ | 714 | 70.16x | 902 |

***Table 4.11:*** *Performance of the different programs in detail (dataset I)*

More information on performance is available in Tab. 4.11. The greatest speed-up is obviously achieved by Lambda$_{fa}$, with PAUDA$_{fa}$ also being above factor 1000. Next, one can group Lambda$_{def}$, UBlast6$_{def}$ and PAUDA$_{sl}$ together, with seep-ups over 200x. And lastly Lambda$_{sl}$, UBlast6$_{sl}$ and RAPSearch2, with speed-ups between 58x and 76x. It should be noted, that within each of these groups the Lambda-profile is the fastest.

This overall lead in performance by Lambda comes at the cost of memory-usage, though, with Lambda$_{def}$ already requiring 13GiB of RAM and Lambda$_{sl}$ definetly exceeding the capabilites of an average scientific workstation. PAUDA's memory usage could not be tracked by the benchmark, but the Java Virtual Machine immediately claims 20GiB of memory which it doesn't seem to exceed, so this at least is an upper bound.

All in all, Lambda comes out amongst the most sensitive programs for dataset I and is definetely the fastest.

## 4.3.2 Dataset II

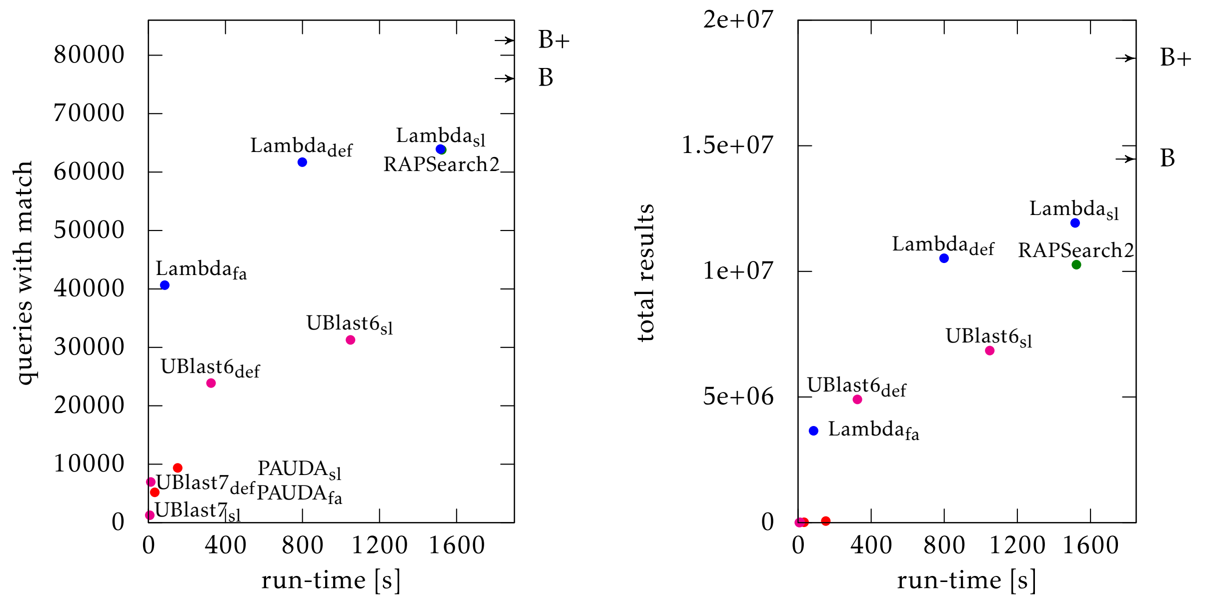


***Figure 4.7:*** *Comparison of applications (dataset II)*

BLAST and BLAST+ are far outside the displayed area, their y-coordinates are given on the right sides of the plots, as B and B+, resp.;
RAPSearch2 and Lambda$_{\text{sl}}$ are at (almost) identical coordinates in left plot;
UBlast7 and PAUDA profiles are all close to $(0,0)$ in right plot, PAUDA$_{\text{sl}}$ is right-most

First of all it has to be noted, that again, UBlast7 did not produce valid results and was thus exempted from further analysis. In general, the results on dataset II are more diverse with the spread of sensitivity very large, and no BLAST-competitor close to full sensitivity, as we can see from Fig. 4.7. As with the first dataset trends for the amount of matching queries are more pronounced for the total results, but the basic conclusions are that PAUDA performs worse for this dataset and UBlast6 also struggles with sensitivity. Lambda in its default and slow profiles, as well as RAPSearch2 produce reasonable results, with Lambda being much faster in one case and as fast as RAPSearch2 in the other.

| Program | matchedQueries |  | bitScore |  |  | % Id. | Compared to BLAST+ |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|---|
|  | abs | of BLAST+ | Q1 | Med | Q3 | Med | ≈ | > | < | * | ∅ |
| BLAST | 76012 | 92.11% | 87.0 | 146.0 | 244.0 | 45.9 | 51261 | 0 | 6094 | 18657 | 25167 |
| BLAST+ | 82522 | 100.00% | 81.3 | 140.0 | 242.0 | 43.4 | 82522 | 0 | 0 | 0 | 0 |
| PAUDA$_{\text{fa}}$ | 5192 | 6.29% | 98.6 | 137.1 | 164.1 | 50.0 | 227 | 0 | 190 | 4775 | 82105 |
| PAUDA$_{\text{sl}}$ | 9371 | 11.36% | 102.1 | 148.7 | 174.9 | 47.0 | 572 | 0 | 549 | 8250 | 81401 |
| RAPSearch2 | 63807 | 77.32% | 87.8 | 145.9 | 244.2 | 53.3 | 30243 | 0 | 15039 | 18525 | 37240 |
| Lambda$_{\text{fa}}$ | 40646 | 49.25% | 142.5 | 229.1 | 337.8 | 61.1 | 28486 | 0 | 487 | 11673 | 53549 |
| Lambda$_{\text{def}}$ | 61695 | 74.76% | 109.3 | 179.1 | 283.1 | 51.9 | 47240 | 0 | 1322 | 13133 | 33960 |
| Lambda$_{\text{sl}}$ | 63939 | 77.48% | 105.9 | 174.8 | 277.3 | 50.0 | 49403 | 0 | 1444 | 13092 | 31675 |
| UBlast6$_{\text{def}}$ | 23892 | 28.95% | 101.7 | 163.7 | 261.9 | 47.3 | 19386 | 64 | 737 | 3705 | 62335 |
| UBlast6$_{\text{sl}}$ | 31274 | 37.89% | 99.0 | 161.4 | 260.4 | 47.1 | 25701 | 95 | 926 | 4552 | 55800 |
| UBlast7$_{\text{def}}$ | 6957 | 8.43% | 67.4 | 102.1 | 183.0 | 26.2 | 0 | 0 | 0 | 6957 | 82522 |
| UBlast7$_{\text{sl}}$ | 1270 | 1.54% | 71.2 | 110.2 | 168.7 | 26.9 | 0 | 0 | 0 | 1270 | 82522 |

***Table 4.12:*** *Sensitivity of the different programs in detail (dataset II)*

For an explanation of the last 5 columns, see Tab. 4.1 on p.30

Tab. 4.12 underlines these assumptions: PAUDA produces less than 12% of BLAST+'s amount of matched queries, even in slow mode and it recalls less than 2% / 0.5% of BLAST+'s matches. UBlast6 is better, but still clearly below 40% of the matches, and at atmost 32% recall in slow mode. In its fast mode Lambda still wins out against these programs with ∼ 50% of the matched queries and 35% recall.

Of better sensitivity are Lambda's two other modes and RAPSearch2 that achieve 75% -78% of BLAST+'s amount of results and a recall of 55% - 62%. Within this group the recall of both of Lambda's profiles is higher than RAPSearch2's and although RAPSearch2 does assign the same query to the same subject sequences in 55% of the cases, one third of these matches score significantly worse than with BLAST+. This is not the case for Lambda which reaches a similar score for nearly every recalled match (see columns 8 and 10 in Tab. 4.12).

The recall is close to that of the traditional BLAST, which surprisingly deviates from BLAST+ rather strongly. Of the total amount of assigned queries it does however reach 92%.

Concerning the distribution of results, it seems from the bit-score quartiles that RAPSearch2 misses BLAST results over the entire range of bit-scores (it has a similar distribution), while for Lambda mostly lower scoring results are missing (the quartiles are higher). This is especially true for the Lambda$_{fa}$ profile. BLAST and BLAST+ also find more results with lower sequence identity than all competitors.

| Program | runTime[s] | rel. speedUp | memUsage[MiB] |
|---|---:|---:|---:|
| BLAST | 52669 | 1.49x | 241 |
| BLAST+ | 78354 | 1.00x | 354 |
| PAUDA$_{fa}$ | 33 | 2374.36x | ≤ 20000 |
| PAUDA$_{sl}$ | 152 | 515.48x | ≤ 20000 |
| RAPSearch2 | 1524 | 51.41x | 1733 |
| Lambda$_{fa}$ | 85 | 921.81x | 4972 |
| Lambda$_{def}$ | 799 | 98.07x | 22431 |
| Lambda$_{sl}$ | 1516 | 51.68x | 82485 |
| UBlast6$_{def}$ | 325 | 241.09x | 1025 |
| UBlast6$_{sl}$ | 1049 | 74.69x | 1044 |

***Table 4.13:*** *Performance of the different programs in detail (dataset II)*

In regard to performance, the programs behave similarly to dataset I, with Lambda and RAPSearch2 a little bit slower by comparison. This would be expected regardless of algorithm, because they produce so many more results than the other programs. In any case the general trends are preserved, PAUDA$_{fa}$ and Lambda$_{fa}$ are the quickest by a large margin, with the different default and slow profiles following and RAPSearch2 in the end. Memory usage is higher for all the applications, however the effect is most notable on the already high values of Lambda.

For dataset II, Lambda appears a good choice, as well: regarded by the amount of matched queries and the recall of BLAST+'s results, it is the most sensitive in profiles default and slow, while still being faster/as fast as the only viable alternative RAPSearch2. At lower levels of required sensitivity Lambda$_{fa}$ serves as a good alternative to UBlast6, because it is both, significantly faster and more sensitive. It should, however, be noted that the gap towards BLAST+ is larger than in dataset I.

# 5 Discussion

In this last chapter, we will discuss the most important results and their implications, as well as give an outlook on possible future developments.

## 5.1 Lambda's parameter space

| Alphabet | SeedLength | ErrorTolerance |
|---|---|---|
| Murphy10 | 10 | hamming distance 1 |

***Table 5.1:*** *Default values for Lambda's main parameters*

Since we decided to chose parameter defaults based on their performance in the benchmark, these are empirically justified. It does, however, make sense to analyze the choices and discuss some of the more interesting observations from the tests.

Murphy10 appeared to be the best-performing of the alphabets, which is in line with other comparisons made (Ye et al., 2011), and confirms RAPSearch2's and UBlast's choice of it as default. The Lambda alphabets had a higher sensitivity in certain areas, though, and further research will show whether they can be improved speed-wise. As expected, the unreduced alphabet was only useful for identifying matches with higher sequence identity, but the relative level of sensitivity that it yielded on our datasets was satisfactory.

A choice of seed length 10 for the default profile might seem high, but in combination with an allowed hamming distance of 1 the sensitivity was comparable to an error-free seed length of 7 – at a significantly better speed. Since this is close to the default seed length of 6 for UBlast and RAPSearch2, this choice seems to have some theoretical support, as well.

Edit distance seeding did proof the most sensitive, as would be expected (because it contains at least all seed hits which other error models find at equal configurations). However, in all cases the margin was too small to justify the increase in run-time. The only small gain in additional results is likely due to the same heuristic cutoff being applied, as with perfect and hamming distance seeding. This cutoff is based on the score of the seed alignment, and by definition, a seed with a gap will score significantly lower than one without – especially with the gap open penalty used in the default scoring scheme. Continued analysis will have to show whether other heuristics perform better in this situation, or if seeds with one indel in protein space just aren't abundant enough to make this approach viable.

## 5.2 Comparison of programs

While the first chapters have illustrated that the basic concepts and ideas that are used by the different applications are very similar, the benchmark has shown that this does not necessarily lead to similar results.

**BLAST** and **BLAST+** differed slightly in the amount of matched queries and the actual best assignment. Some of this may be explained through the difference in soft and hard masking. Although the original publication (Camacho et al., 2009) claims speed improvements for BLAST+ over BLAST, it was significantly slower on the second dataset. This observation is shared by Zhao et al. (2012).

Based on our results **PAUDA** seems to be an unsuitable choice, when recalling BLAST's results is important. Beside not finding the same matches, it also produces significantly less total results than any other program. This is especially true for longer read lengths. One possible explanation for the poor performance in the latter case is that Bowtie2 – which works at PAUDA's core and is designed for read mapping – performs poor in its local alignment mode, when the expected local alignment size is much shorter than the original read. Further investigation into Bowtie2's behavior and the distribution of alignment lengths in PAUDA's results could possibly clarify this. Another factor of why PAUDA likely misses many hits early on, is that its seeds are very long. The increased likelihood of substitutions in these may be compensated by the lower size alphabet reduction, but gaps are not accounted for. Finally the bad rate of BLAST recalls might be explained by PAUDA apparently not performing a realignment in regular protein space, i.e. since the alignment was computed in the four-letter-reduced protein space, it is very likely to be sub-optimal and a different subject sequence appearing as best, probable. A slightly lower recall could have been the outcome of hard-masking (as with traditional BLAST), but not to this extent (only 0.5% of BLAST+'s best matches are recalled on the second dataset). PAUDA's speed is comparatively high, but even if taking the measure of *results per time* into account (which is discussed in PAUDA's publication) there is always a program/parameter combination that performs better.

**RAPSearch2** is the most sensitive of the previously existing BLAST alternatives with decent results on both datasets. It produces results from the same range of bit-scores as BLAST, making a bias for a certain quality range unlikely. The measured speed-ups of 50x - 60x over BLAST are similar to those measured by Huson and Xie (2013), but less than the speed-ups of 160x-1000x claimed in the original publication (Zhao et al., 2012). With these speedups it comes in last among the BLAST-alternatives compared.

Of **UBlast** we had to compare two versions, because the newest (UBlast7) did not produce significant results. This was surprising as UBlast is a commercially supported application. Further investigation will have to show, whether we encountered a bug or missed subtle changes in the application's behavior. UBlast6 had mixed results with a high sensitivity on the Illumina reads and a comparatively low one on the Sanger reads. With speed-ups of over 200x compared to BLAST+, UBlast6 is notably faster than RAPSearch2.

The fact that RAPSearch2 and UBlast6 produce vastly different results on the Sanger reads and also slightly different results on the Illumina reads underlines the importance of heuristic parameters for the search, because both programs are designed very similarly: they both use hash-tables for lookup, with the same alphabet reduction, the same effective seed length and an extension step that includes ungapped and gapped x-drop extension on the regular alphabet. Unfortunately there is no documentation on the parameters and the checks in between these steps, so that we cannot explain the programs' behavior more thoroughly.

As part of this work, we developed **Lambda** that is of comparable overall sensitivity with RAPSearch2, but with a higher recall of BLAST+'s results for both datasets. Based on the goals of our benchmark, this means Lambda is superior to the other applications in regard to sensitivity. With speed-ups of 100x - 280x it is significantly faster than RAPSearch2 and also faster than PAUDA and UBLast6 at settings and datasets where they produce relevant amounts of results. This gain in speed currently comes at the price of a notably higher memory usage.

For situations where a higher speed is an absolute priority and a lower sensitivity is acceptable – currently covered by e.g. PAUDA for Illumina reads or UBlast6 for Sanger reads – Lambda offers a fast parameter profile. This is capable to achieve speed-ups of 900x-2500x while being significantly more sensitive than PAUDA on both datasets, significantly better than UBlast6 on the second, and not significantly worse than UBlast6 on the first.

A noteworthy observation is that the bit-scores of Lambda's matches for dataset II are distributed in a higher range than BLAST's and RAPSearch2's. This is positive on the one hand, because it means that Lambda finds more / misses less high-scoring results. On the other hand, it could indicate that Lambda is less good at finding lower scoring results, a possible outcome of the current minimum seed score or the longer effective seed length. Although the allowed hamming distance of one compensates much of the latter (see above) and sequence identities don't necessarily suggest this either. Furthermore, UBlast6's bit score quartiles are similar, so this would apply there, as well. In any case, one might argue that Lambda is over-optimized for the particular bit-score threshold / low e-value cutoff. While this could explain performance leads of Lambda in some areas it would still not explain the comparatively lower sensitivity of other programs. After all, high-scoring hits that are missing there now, will not reappear at higher cutoffs (as previously explained the cutoff is applied a posteriori). Moreover the chosen cutoff is well supported for metagenomic use cases (see Sec. 4.1), even PAUDA's reference publication Mackelprang et al. (2011) advertises an e-value cutoff of $10^{-5}$.

## 5.3 Conclusion & Outlook

Goals of this work included investigating state of the art local alignment programs, studying the theoretical background of the employed algorithms and exploring optimizations strategies in high performance computing, before realizing our own solution to the problem. Furthermore, a basis for comparatively analyzing different applications had to be devised and implemented.

We were successful in all of these areas, as we have gained comprehensive insight into the algorithmic challenges of local alignment searches, have contributed numerous features to the SeqAn library and have developed a high-performance tool that is a viable alternative to existing solutions.

In many cases our application is both, superior in speed and in sensitivity to all other BLAST alternatives; in no situation does it perform noticeably worse than the best. The benchmark has been very useful in assessing different attributes of the applications' behavior and will serve future comparisons well, that require more information than a count of results. In contrast to some of the other programs Lambda is Free and Open Source software, has all of its parameters documented and requires no external libraries beyond SeqAn.

In its current form, performance benefits of Lambda are indisputable, but it does require a fair amount of memory to achieve this. This requirement of the search approach has already been observed in the SeqAn project, and a new finder-module is being developed as part of the upcoming release of the Masai read mapper. The new finder module does not work with double indexing, but uses a more traditional search approach, which does not require the caching of hits before verification. This approach gives up certain theoretical speed advantages (of the double indexing), but through an implementation more thoroughly adapted for parallelization still achieves speed-ups over the method used in this work. Additionally the data structures and algorithms were ported to run on graphics cards (GPGPU), yielding speed-ups of up multiple factors over current high-end CPUs. Were Lambda to adapt and incorporate this approach, this would not only solve the issue of high memory usage, but also gain Lambda further speed ups in the important seeding phase. With a smaller memory footprint Lambda could also explore lower seed lengths and thus detect more distantly related sequences, increasing the sensitivity.

For most configurations the extension phase is still the most computationally expensive step, why improving this further would be a high priority, too. And, also in this area, there are promising developments in the SeqAn library, as support for CPU extensions is currently being worked on for the dynamic programming code. Others projects (Alachiotis et al., 2013) have shown how seed extension specifically can benefit from SSE and achieved an 6x speed-up, which scales with the number of CPU-cores. Having the seeding take place on the GPU *while* the extension is done on the CPU, would allow for an even higher level of parallelization.

Lastly, as with most heuristic applications, the sweet spot between sensitivity and speed must be found by continuous testing and tuning. Further heuristics for avoiding unnecessary extensions (and retaining needed ones) should be evaluated.

# 6 Acknowledgments

First of all, I want to thank Prof. Dr. Knut Reinert and Dr. Bernhard Renhard for giving me the opportunity to work on such an interesting topic and supervising this work. Sincere gratitude goes to Jochen Singer, my advisor, who was very helpful and always available with valuable hints and suggestions. Of course, other members of the AG Algorithmische Bioinformatik were also supportive, and only by being able to discuss and seek advice with them, was I able to achieve the results in the given time. Special thanks go to René Rahn, who introduced me to the dynamic programming code and prepared necessary changes in the library. I also want to thank Sabrina Krakau for cross-reading my thesis.

Over the course of my studies, there have been many people who supported me, and I want to thank my family, friends and comrades who were there in these times. Special thanks go to my parents for making my studies in this form possible and for encouraging me to step up to the different challenges that life holds.

Finally I want to mention some of the many Free and Open Source Software projects that were fundamental to the development of Lambda and the writing of this thesis. Among others I want to thank the GNU-Project[1], KDE e.V.[2] and the LaTeX-Project[3].

[1] http://www.gnu.org
[2] http://www.kde.org
[3] http://www.latex-project.org

# References

Adve, S. V., Hart, J. C., et al. (2008). Parallel Computing Research at Illinois: The UPCRC Agenda. Technical report.

Alachiotis, N., Berger, S., Flouri, T., Pissis, S., and Stamatakis, A. (2013). libgapmis: extending short-read alignments. *BMC Bioinformatics*, 14(Suppl 11):S4.

Altschul, S., Gish, W., Miller, W., Myers, E., and Lipman, D. (1990). Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–410.

Altschul, S. F., Boguski, M. S., Gish, W., and Wootton, J. C. (1994). Issues in searching molecular sequence databases. *Nat. Genet.*, 6(2):119–129.

Altschul, S. F., Bundschuh, R., Olsen, R., and Hwa, T. (2001). The estimation of statistical parameters for local alignment score distributions. *Nucleic Acids Res.*, 29(2):351–61.

Altschul, S. F. and Gish, W. (1996). Local alignment statistics. *Meth. Enzymol.*, 266:460–80.

Altschul, S. F., Madden, T. L., Schaffer, A. A., Zhang, 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(17):3389.

Altschul, S. F., Wootton, J. C., Gertz, E. M., Agarwala, R., Morgulis, A., Schäffer, A. A., and Yu, Y.-K. (2005). Protein database searches using compositionally adjusted substitution matrices. *FEBS J.*, 272(20):5101–9.

Asanovic, K., Bodik, R., Catanzaro, B. C., Gebis, J. J., Husbands, P., Keutzer, K., Patterson, D. A., Plishker, W. L., Shalf, J., Williams, S. W., and Yelick, K. A. (2006). The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley.

Bacardit, J., Stout, M., Hirst, J. D., Valencia, A., Smith, R. E., and Krasnogor, N. (2009). Automated Alphabet Reduction for Protein Datasets. *BMC Bioinformatics*, 10.

Barreiro, L. B., Laval, G., Quach, H., Patin, E., and Quintana-Murci, L. (2008). Natural selection has driven population differentiation in modern humans. *Nat. Genet.*, 40(3):340–345.

Bazinet, A. L. and Cummings, M. P. (2012). A comparative evaluation of sequence classification programs. *BMC Bioinformatics*, 13:92.

Burkhardt, S. and Kärkkäinen, J. (2003). Better Filtering with Gapped q-Grams. *Fundam. Inform.*, 56(1-2):51–70.

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

Chao, K.-M., Pearson, W. R., and Miller, W. (1992). Aligning two sequences within a specified diagonal band. *Computer Applications in the Biosciences*, 8(5):481–487.

Cock, P. J. A., Fields, C. J., Goto, N., Heuer, M. L., and Rice, P. M. (2010). The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants. *Nucleic Acids Research*, 38(6):1767–1771.

Dasgupta, S., Papadimitriou, C., and Vazirani, U. (2006). *Algorithms*. McGraw-Hill.

Dayhoff, M., Schwartz, R., and Orcutt, B. (1978). A model of evolutionary change in proteins. In Dayhoff, M., editor, *Atlas of Protein Sequence and Structure*, volume 5, pages 345–352. National Biomedical Research Foundation, Washington, D. C.

Döring, A., Weese, D., Rausch, T., and Reinert, K. (2008). SeqAn An efficient, generic C++ library for sequence analysis. *BMC Bioinformatics*, 9.

Edgar, R. C. (2010). Search and clustering orders of magnitude faster than BLAST. *Bioinformatics*, 26(19):2460–2461.

Eikmeyer, F. G., Rademacher, A., Hanreich, A., Hennig, M., Jaenicke, S., Maus, I., Wibberg, D., Zakrzewski, M., Puhler, A., Klocke, M., and Schluter, A. (2013). Detailed analysis of metagenome datasets obtained from biogas-producing microbial communities residing in biogas reactors does not indicate the presence of putative pathogenic microorganisms. *Biotechnol Biofuels*, 6(1):49.

Ferragina, P. and Manzini, G. (2000). Opportunistic Data Structures with Applications. In *FOCS*, pages 390–398. IEEE Computer Society.

Gerlach, W. and Stoye, J. (2011). Taxonomic classification of metagenomic shotgun sequences with CARMA3. *Nucleic Acids Res.*, 39(14):e91.

Gotoh, O. (1981). An Improved Algorithm for Matching Biological Sequences. In *Journal of Molecular Biology*, volume 162, pages 705–708.

Gusfield, D. (1997). *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, New York, NY, USA.

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

Henikoff, S. and Henikoff, J. (1992). Amino acid substitution matrices from protein blocks. *Proceedings of the National Academy of Sciences of the United States of America (PNAS)*, 89:10915–10919.

Hirschberg, D. S. (1975). A linear space algorithm for computing maximal common subsequences. *Commun. ACM*, 18(6):341–343.

Hopcroft, J. E. and Ullman, J. D. (1979). *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley.

Huson, D. H., Auch, A. F., Qi, J., and Schuster, S. C. (2007). MEGAN analysis of metagenomic data. *Genome Res.*, 17(3):377–86.

Huson, D. H. and Xie, C. (2013). A poor man's BLASTX–high-throughput metagenomic protein database search using PAUDA. *Bioinformatics*.

Karlin, S. and Altschul, S. F. (1990). Methods for Assessing the Statistical Significance of Molecular Sequence Features by Using General Scoring Schemes. *Proceedings of the National Academy of Sciences of the United States of America*, 87(6):pp. 2264–2268.

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

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

Koskinen, P. and Holm, L. (2012). SANS: high-throughput retrieval of protein sequences allowing 50% mismatches. *Bioinformatics*, 28(18):438–443.

Krause, L., Diaz, N. N., Goesmann, A., Kelley, S., Nattkemper, T. W., Rohwer, F., Edwards, R. A., and Stoye, J. (2008). Phylogenetic classification of short environmental DNA fragments. *Nucleic Acids Res.*, 36(7):2230–2239.

Lamendella, R., Domingo, J. W. S., Ghosh, S., Martinson, J., and Oerther, D. B. (2011). Comparative fecal metagenomics unveils unique functional capacity of the swine gut. *BMC Microbiol.*, 11:103.

Langmead, B. and Salzberg, S. L. (2012). Fast gapped-read alignment with Bowtie 2. *Nat. Methods*, 9(4):357–9.

Levenshtein, V. (1966). Binary Codes Capable of Correcting Deletions, Insertions, and Reversals. *Soviet Physics-Doklady*, 10(8):707–710.

Li, T., Fan, K., Wang, J., and Wang, W. (2003). Reduction of protein sequence complexity by residue grouping. *Protein Eng.*, 16(5):323–30.

Lipman, D. and Pearson, W. (1985). Rapid and sensitive protein similarity searches. *Science*, 227(4693):1435–1441.

Liu, L., Li, Y., Li, S., Hu, N., He, Y., Pong, R., Lin, D., Lu, L., and Law, M. (2012). Comparison of next-generation sequencing systems. *J. Biomed. Biotechnol.*, 2012:251364.

Mackelprang, R., Waldrop, M. P., DeAngelis, K. M., David, M. M., Chavarria, K. L., Blazewicz, S. J., Rubin, E. M., and Jansson, J. K. (2011). Metagenomic analysis of a permafrost microbial community reveals a rapid response to thaw. *Nature*, 480(7377):368–71.

Manber, U. and Myers, E. W. (1993). Suffix Arrays: A New Method for On-Line String Searches. *SIAM J. Comput.*, 22(5):935–948.

Marco, D. (2011). *Metagenomics: Current Innovations and Future Trends*. Caister Academic Press.

Mealy, G. H. (1955). A Method for Synthesizing Sequential Circuits. *Bell System Technical Journal*, 34(5):1045–1079.

Miyazawa, S. and Jernigan, R. L. (1996). Residue-residue potentials with a favorable contact pair term and an unfavorable high packing density term, for simulation and threading. *J. Mol. Biol.*, 256(3):623–44.

Moore, G. E. (1965). Cramming more components onto integrated circuits. *Electronics*, 38(8).

Morgulis, A., Gertz, E. M., Schäffer, A. A., and Agarwala, R. (2006). WindowMasker: window-based masker for sequenced genomes. *Bioinformatics*, 22(2):134–141.

Morrison, D. R. (1968). PATRICIA - Practical Algorithm To Retrieve Information Coded in Alphanumeric. *J. ACM*, 15(4):514–534.

Murphy, L. R., Wallqvist, A., and Levy, R. M. (2000). Simplified amino acid alphabets for protein fold recognition and implications for folding. *Protein Eng.*, 13(3):149–52.

Needleman, S. B. and Wunsch, C. D. (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.

Okonechnikov, K., Golosova, O., and Fursov, M. (2012). Unipro UGENE: a unified bioinformatics toolkit. *Bioinformatics*, 28(8):1166–1167.

Patterson, D. A. and Hennessy, J. L. (1997). *Computer Organization & Design: The Hardware/Software Interface, Second Edition*. Morgan Kaufmann.

Pearson, W. R. and Lipman, D. J. (1988). Improved tools for biological sequence comparison. *Proceedings of the National Academy of Sciences*, 85(8):2444–2448.

Regan, L. and DeGrado, W. F. (1988). Characterization of a helical protein designed from first principles. *Science*, 241(4868):976–978.

Sander, C. and Schul, G. (1979). Degeneracy of the information contained in amino acid sequences: evidence from overlaid genes. *Journal of Molecular Evolution*, 13(3):245–252.

Sanger, F. and Coulson, A. R. (1975). A rapid method for determining sequences in DNA by primed synthesis with DNA polymerase. *J. Mol. Biol.*, 94(3):441–8.

Sanger, F., Nicklen, S., and Coulson, A. (1977). DNA sequencing with chain-terminating inhibitors. *Proceedings of the National Academy of Sciences of the United States of America (PNAS)*, 74:5463–5467.

Sellers, P. H. (1974). On the Theory and Computation of Evolutionary Distances. *SIAM Journal on Applied Mathematics*, 26(4):787–793.

Siragusa, E., Weese, D., and Reinert, K. (2012). Fast and sensitive read mapping with approximate seeds and multiple backtracking. *CoRR*, abs/1208.4238.

Smith, T. and Waterman, M. (1981). Identification of Common Molecular Subsequences. *Journal of Molecular Biology*, 147(1):195–197.

Tetu, S. G., Breakwell, K., Elbourne, L. D., Holmes, A. J., Gillings, M. R., and Paulsen, I. T. (2013). Life in the dark: metagenomic evidence that a microbial slime community is driven by inorganic nitrogen metabolism. *ISME J*, 7(6):1227–1236.

Ukkonen, E. (1993). Approximate String-Matching over Suffix Trees. In Apostolico, A., Crochemore, M., Galil, Z., and Manber, U., editors, *CPM*, volume 684 of *Lecture Notes in Computer Science*, pages 228–242. Springer.

Venter, J. C., Remington, K., Heidelberg, J. F., Halpern, A. L., Rusch, D., Eisen, J. A., Wu, D., Paulsen, I., Nelson, K. E., Nelson, W., Fouts, D. E., Levy, S., Knap, A. H., Lomas, M. W., Nealson, K., White, O., Peterson, J., Hoffman, J., Parsons, R., Baden-Tillson, H., Pfannkoch, C., Rogers, Y.-H., and Smith, H. O. (2004). Environmental Genome Shotgun Sequencing of the Sargasso Sea. *Science*, 304(5667):66–74.

Vingron, M. and Waterman, M. S. (1994). Sequence alignment and penalty choice. Review of concepts, case studies and implications. *J. Mol. Biol.*, 235(1):1–12.

Wang, J. and Wang, W. (1999). A computational approach to simplifying the protein folding alphabet. *Nature Structural Biology*, 6:1033–1038.

Waterman, M., Smith, T., and Beyer, W. (1976). Some biological sequence metrics. *Advances in Mathematics*, 20(3):367–387.

Weiner, P. (1973). Linear Pattern Matching Algorithms. In *SWAT (FOCS)*, pages 1–11. IEEE Computer Society.

Wommack, K. E., Bhavsar, J., and Ravel, J. (2008). Metagenomics: read length matters. *Appl. Environ. Microbiol.*, 74(5):1453–63.

Wootton, J. C. and Federhen, S. (1993). Statistics of Local Complexity in Amino Acid Sequences and Sequence Databases. *Computers & Chemistry*, 17(2):149–163.

Xiong, J. (2006). *Essential bioinformatics.* Cambridge University Press.

Ye, Y., Choi, J.-H., and Tang, H. (2011). RAPSearch: a Fast Protein Similarity Search Tool for Short Reads. *BMC Bioinformatics*, 12:159.

Zhao, Y., Tang, H., and Ye, Y. (2012). RAPSearch2: a fast and memory-efficient protein similarity search tool for next-generation sequencing data. *Bioinformatics*, 28(1):125–126.

# A Additional materials on the implementation

## A.1 Lambda manual

```
lambda - BLAST compatible local aligner optimized for NGS and Metagenomics.
===========================================================================

SYNOPSIS
    lambda [OPTIONS] -q QUERY.fasta -d DATABASE.fasta [-o output.m8]

DESCRIPTION
    Lambda is a local aligner that is faster than BLAST, optimized for many query sequences.

    -h, --help
          Displays this help message.
    --version
          Display version information

  Input Options:
    -q, --query IN
          Query sequences (fasta). Valid filetypes are: fasta, fa, fna, faa, fastq, and fq.
    -d, --database IN
          Database sequences (fasta), with precomputed index (.sa). Valid filetypes are: fasta, fa, fna, and faa.

  Output Options:
    -o, --output OUT
          File to hold reports on hits (.m8 is blastall -m8 et cetera) Valid filetypes are: m0, m8, and m9. Default:
          output.m8.

  Program Options:
    -p, --program OUT
          Blast Operation Mode. One of blastn, blastp, blastx, tblastn, and tblastx. Default: blastx.
    -a, --alph NUM
          Alphabet Reduction for AminoAcid Alphabet (0 -> off; 2 -> Murphy10; 8,10,12 -> Lambda*). Default: 2.
    -qp, --querypart NUM
          Partition query into N blocks;defaults to number of CPUs; increase if you run out of memory to mulitple of
          NCPU Default: 0.

  Seeding / Filtration:
    -su, --ungappedseeds NUM
          Deacitvate EditDistance-Seeding. Default: 0.
    -sl, --seedlength NUM
          Length of the seeds (0 -> choose automatically). Default: 0.
    -sd, --seeddelta NUM
          maximum seed distance. Default: 1.
    -sg, --seedgravity NUM
          Seeds closer than this are joined(-1 -> choose automatically). Default: -1.
    -sm, --seedminlength NUM
          after postproc shorter seeds are discarded(0 -> choose automatically). Default: 0.
    -ss, --seedminscore NUM
          after postproc worse seeds are discarded. Default: 12.
    -sb, --seedminbits NUM
          after postproc worse seeds are discarded(-1 -> off). Default: -1.

  Extension:
    -x, --xdrop NUM
          Stop Banded extension if score x below the maximum seen(-1 means no xdrop). Default: 30.
    -b, --band NUM
          Size of the DP-band used in extension (-3 means log2 of query length; -2 means sqrt of query length; -1
          means full dp; n means band of size 2n+1) Default: -3.
    -e, --evalue NUM
          Minimum E-Value for Results. Default: 0.1.

ENVIRONMENT VARIABLES
    TMPDIR
          set this to a local directory with lots of space. If you can afford it use /dev/shm.
    OMP_NUM_THREADS
          number of threads to use.
```

***Figure A.1:*** *Output of the help-parameter*

## A.2 Alphabet reduction

| Target-Size | Clusters |
|---|---|
| 1 | ASXTRKQEZNDBHGPCLIVMFYW* |
| 2 | ASXTRKQEZNDBHGPCLIVMFYW * |
| 3 | ASXTRKQEZNDBHGP CLIVMFYW * |
| 4 | ASXTRKQEZNDBHGP CLIVM FYW * |
| 5 | ASXTRKQEZNDBHG P CLIVM FYW * |
| 6 | ASXT RKQEZNDBHG P CLIVM FYW * |
| 7 | ASXT RKQEZNDBH G P CLIVM FYW * |
| 8 | ASXT RKQEZNDBH G P C LIVM FYW * |
| 9 | ASXT RKQEZNDB H G P C LIVM FYW * |
| 10 | ASXT RKQEZ NDB H G P C LIVM FYW * |
| 11 | A SXT RKQEZ NDB H G P C LIVM FYW * |
| 12 | A SXT RKQEZ NDB H G P C LIVM FY W * |
| 13 | A SX T RKQEZ NDB H G P C LIVM FY W * |
| 14 | A SX T RK QEZ NDB H G P C LIVM FY W * |
| 15 | A S X T RK QEZ NDB H G P C LIVM FY W * |
| 16 | A S X T RK QEZ NDB H G P C LIV M FY W * |
| 17 | A S X T RK QEZ N DB H G P C LIV M FY W * |
| 18 | A S X T RK QEZ N DB H G P C LIV M F Y W * |
| 19 | A S X T R K QEZ N DB H G P C LIV M F Y W * |
| 20 | A S X T R K QEZ N DB H G P C L IV M F Y W * |
| 21 | A S X T R K Q EZ N DB H G P C L IV M F Y W * |
| 22 | A S X T R K Q EZ N DB H G P C L I V M F Y W * |
| 23 | A S X T R K Q EZ N D B H G P C L I V M F Y W * |
| 24 | A S X T R K Q E Z N D B H G P C L I V M F Y W * |

***Table A.1:*** *Full overview of Lambda's alphabet reductions*

## A.3 BLAST pairwise output

```
BLASTX I/O Module of SeqAn-1.5.0 (http://www.seqan.de)

Reference: Altschul, Stephen F., Thomas L. Madden, Alejandro A. Schaffer,
Jinghui Zhang, Zheng Zhang, Webb Miller, and David J. Lipman (1997),
"Gapped BLAST and PSI-BLAST: a new generation of protein database search
programs",  Nucleic Acids Res. 25:3389-3402.

Reference for SeqAn: Döring, A., D. Weese, T. Rausch, K. Reinert (2008): SeqAn --
An efficient, generic C++ library for sequence analysis. BMC Bioinformatics,
9(1), 11. BioMed Central Ltd. doi:10.1186/1471-2105-9-11



Database: /home/mi/h4nn3s/takifugu/ma_arbeit/svn/bench/sequence/db/uniprot_sprot.fasta
           538259 sequences; 191113170 total letters

[...]

Query= UBAFZ74TR  Sample 3 Mate UBAFZ74TF trimmed_to 84 845

Length=88


                                                                  Score     E
Sequences producing significant alignments:                      (Bits)  Value

sp|A8GPR9|HLYC_RICAH Possible hemolysin C OS=Rickettsia akari (...   86  2e-16
sp|A8GUH1|HLYC_RICB8 Possible hemolysin C OS=Rickettsia bellii ...   80  2e-14
sp|Q1RGX2|HLYC_RICBR Possible hemolysin C OS=Rickettsia bellii ...   80  2e-14
sp|A8EZU0|HLYC_RICCK Possible hemolysin C OS=Rickettsia canaden...   77  1e-13
sp|Q92GI2|HLYC_RICCN Hemolysin C homolog OS=Rickettsia conorii ...   84  8e-16
sp|Q4UK99|HLYC_RICFE Hemolysin C OS=Rickettsia felis (strain AT...   84  8e-16
sp|A8F2M1|HLYC_RICM5 Hemolysin C homolog OS=Rickettsia massilia...   81  9e-15
sp|O05961|HLYC_RICPR Hemolysin C OS=Rickettsia prowazekii (stra...   87  2e-16
sp|A8GTI4|HLYC_RICRS Hemolysin C homolog OS=Rickettsia ricketts...   84  8e-16
sp|Q68W10|HLYC_RICTY Hemolysin C OS=Rickettsia typhi (strain AT...   87  1e-16
sp|Q50592|Y1842_MYCTU UPF0053 protein Rv1842c/MT1890 OS=Mycobac...   95  4e-19
sp|O07585|YHDP_BACSU UPF0053 protein YhdP OS=Bacillus subtilis ...  101  6e-21
sp|O07589|YHDT_BACSU UPF0053 protein YhdT OS=Bacillus subtilis ...   98  7e-20
sp|P54505|YQHB_BACSU UPF0053 protein YqhB OS=Bacillus subtilis ...   93  1e-18
sp|P54428|YRKA_BACSU UPF0053 protein YrkA OS=Bacillus subtilis ...   96  3e-19
sp|O05241|YUGS_BACSU UPF0053 protein YugS OS=Bacillus subtilis ...  114  7e-25

ALIGNMENTS
> sp|A8GPR9|HLYC_RICAH Possible hemolysin C OS=Rickettsia akari (s
train Hartford) GN=tlyC PE=3 SV=1
Length=301

 Score =  86.7 bits (213), Expect =  2e-16
 Identities = 43/131 (32%), Positives = 83/131 (63%), Gaps = 2/131 (1%)
 Frame = -2

Query  760     DEAHLIGSIIELGDTIVREVMVPRPDMVTLEAHLTVSE-ALAVVVDTGFTRLPVVGDTVD  818
               DE +++ +++EL D  + ++MVPR D+V ++    ++E + ++ ++   TR  +   T+D
Sbjct  60      DERNILANLLELEDKTIEDIMVPRSDIVAIKLTANLAELSESIKLEVPHTRTLIYDGTLD  119

Query  819     DVIGLVLSKDLLAAQLRNPNDTDMRGLAREAVFVPESKRVVELMREMQSSKSHLAIVVDE  878
               +V+G +  KDL  A     N   ++ L R+ +    S ++++L+ +M+  ++H+AIVVDE
Sbjct  120     NVVGFIHIKDLFKALATKQNGR-LKKLIRKHIIAAPSMKLLDLLAKMRRERTHIAIVVDE  178

Query  879     YGGTAGLVSLE  889
               YGGT GLV++E
Sbjct  179     YGGTDGLVTIE  189

[...]
```

***Figure A.2:*** *Example output in pairwise format*

# B Additional materials on the benchmark

## B.1 Details on pre- and post-processing

**Common post-processing** Post-processing that it is performed for all programs is (1) stripping the output file of comments and empty lines, (2) filtering out results that remain below the minimum bit-score, and (3) sorting the results alphabetically. Then the best (by bit-score) result is extracted for every query sequence and for every query and subject pair and each are saved to an extra file. Results per file are counted and for the best-per-query file the median bit-score and the median percent-identity are computed, as well as the "overlap" of the results with the BLAST+-reference result file.

**BLAST** Benchmark contains no program-specific pre- or post-processing for this module.

**BLAST+** Benchmark contains no program-specific pre- or post-processing for this module.

**PAUDA** PAUDA prints output in the pairwise BLAST format (`blastall -m 0`), so in a post-processing step this output needs to be converted to tabular format. The benchmark contains a routine for this, however not all information is converted, see App. B.3.

**RAPSearch2** Since RAPSearch2 works with the logarithm of the e-value instead of the e-value, initially the program parameter is transformed and after program execution the e-value in the output file is re-transformed into non-logarithmic space.

**UBlast** UBlast prints the entire query-id in tabular format, all other programs truncate at the first whitespace in the id. For comparability, query-ids in UBlast's output are also truncated.

**Lambda** Benchmark contains no program-specific pre- or post-processing for this module.

## B.2 Full program command-lines

### BLAST

Indexing command:

```
seg db -x > db_seg && formatdb -i db_seg
```

Search command:

```
blastall -p blastx -e 0.1 -d db_seg -i query -a 12 -b 1000 -C 0 -F F -m 8 -o output
```

### BLAST+

Indexing command:

```
segmasker -infmt fasta -in db -outfmt maskinfo_asn1_bin -out db.asnb && makeblastdb -mask_data
    db.asnb -dbtype prot -in db
```

Search command:

```
blastx -evalue 0.1 -db db -query query -outfmt 6 -comp_based_stats 0 -seg no -num_threads 12
    -max_target_seqs 1000 -out output
```

### PAUDA$_{fast}$

Indexing command:

```
seg /run/shm/uniprot_sprot.fasta -x > db_seg.fasta && pauda-build db_seg.fasta db
```

Search command:

```
pauda-run --fast 100k_long_reads.fna output
```

### PAUDA$_{slow}$

Indexing command:

```
seg /run/shm/uniprot_sprot.fasta -x > db_seg.fasta && pauda-build db_seg.fasta db
```

Search command:

```
pauda-run --slow 100k_long_reads.fna output
```

### RAPSearch2

Indexing command:

```
prerapsearch -d /run/shm/uniprot_sprot.fasta -n ./db
```

Search command:

```
rapsearch -q /run/shm/100k_long_reads.fna -d db -o output -z 12 -e -1 -v 1000 -b 0
```

### UBlast6$_{\text{default}}$

Indexing command:

```
usearch -makeudb_ublast uniprot_sprot.fasta -output db.udb -dbmask seg
```

Search command:

```
usearch -ublast 100k_long_reads.fna -db db.udb -evalue 0.1 -qmask none -threads 12 -blast6out
    output
```

### UBlast6$_{\text{slow}}$

Indexing command:

```
usearch -makeudb_ublast uniprot_sprot.fasta -output db.udb -dbmask seg
```

Search command:

```
usearch -ublast 100k_long_reads.fna -db db.udb -evalue 0.1 -qmask none -accel 1 -threads 12
    -blast6out output
```

### UBlast7$_{\text{default}}$

Indexing command:

```
usearch -makeudb_ublast uniprot_sprot.fasta -output db.udb -dbmask seg
```

Search command:

```
usearch -ublast 100k_long_reads.fna -db db.udb -evalue 0.1 -qmask none -threads 12 -blast6out
    output
```

### UBlast7$_{\text{slow}}$

Indexing command:

```
usearch -makeudb_ublast uniprot_sprot.fasta -output db.udb -dbmask seg
```

Search command:

```
usearch -ublast 100k_long_reads.fna -db db.udb -evalue 0.1 -qmask none -accel 1 -threads 12
    -blast6out output
```

### Lambda$_{\text{fast}}$

Indexing command:

```
segmasker -infmt fasta -in db.fasta -outfmt interval -out db.seg && lambda_indexer -i db.fasta
    -s db.seg -p blastx -a 00
```

Search command:

```
lambda -q query.fasta -d db.fasta -e 0.1 -sl 08 -su 1 -sd 0 -sg 8 -sm 0 -ss 26 -sb 0 -p blastx
    -a 00 -qp 12 -x 30 -o output.m8
```

### Lambda$_{\text{default}}$

Indexing command:

```
segmasker -infmt fasta -in db.fasta -outfmt interval -out db.seg && ~/lambda_indexer -i db.fasta
    -s db.seg -p blastx -a 02
```

Search command:

```
lambda -q query.fasta -d db.fasta -e 0.1 -sl 10 -su 1 -sd 1 -sg 10 -sm 0 -ss 32 -sb 0 -p blastx
    -a 02 -qp 12 -x 30 -o output.m8
```

### Lambda$_{\text{slow}}$

Indexing command:

```
segmasker -infmt fasta -in db.fasta -outfmt interval -out db.seg && ~/lambda_indexer -i db.fasta
    -s db.seg -p blastx -a 10
```

Search command:

```
lambda -q query.fasta -d db.fasta -e 0.1 -sl 10 -su 1 -sd 1 -sg 10 -sm 0 -ss 32 -sb 0 -p blastx
    -a 10 -qp 12 -x 30 -o output.m8
```

## B.3 On comparability of outputs

| Program | %-Id | len | misM | gapOp | qS | qE | sS | sE | eValue | bitScore |
|---|---|---|---|---|---|---|---|---|---|---|
| BLAST | 100.00 | 33 | 0 | 0 | 2 | 100 | 91 | 123 | 5e-17 | 74.3 |
| BLAST+ | 100.00 | 33 | 0 | 0 | 2 | 100 | 91 | 123 | 2e-16 | 74.3 |
| PAUDA | 100 | -1 | -1 | -1 | 2 | 100 | 91 | 123 | 3e-13 | 74.3 |
| RAPSearch2 | 100 | 33 | 0 | 0 | 2 | 100 | 90 | 122 | 2.18776e-13 | 74.3294 |
| Lambda | 100 | 33 | 0 | 0 | 2 | 100 | 91 | 123 | 2.0131e-13 | 74.3294 |

***Table B.1:*** *Example output of all tools compared – %-Id is the percentage of identical positions in the alignment; len is the length of the alignment; misM is the number of mismatches; gapOp is the number of gap-openings; qS,qE, sS, sE are the query and subject, start and end positions, respectively.*

Table B.1 contains the exact tabular output of the tools, for the best hit of `A.Ac.2.1.414748_211733` against `sp|P03641|F_BPPHX`. The first two columns (query and subject identifier) where removed. PAUDA's output contains some columns with value −1, because it doesn't support writing tabular format and its output had to be converted from BLAST's record file format; not all information is converted by the benchmark, as it is non-trivial to extract and the given information suffices to compare the results.

UBlast did not have comparable results, so it could not be included. RAPSearch2 has a bug: its subject positions are always set off by one (this is the case for all results, however the alignments seem to be correct). The bit-scores are identical for all applications, the e-values are close between all BLAST-competitors, but differ from BLAST and BLAST+. This indicates that BLAST's documentation is lacking, since all reimplementations converge on a different value.

## B.4 Additional figures

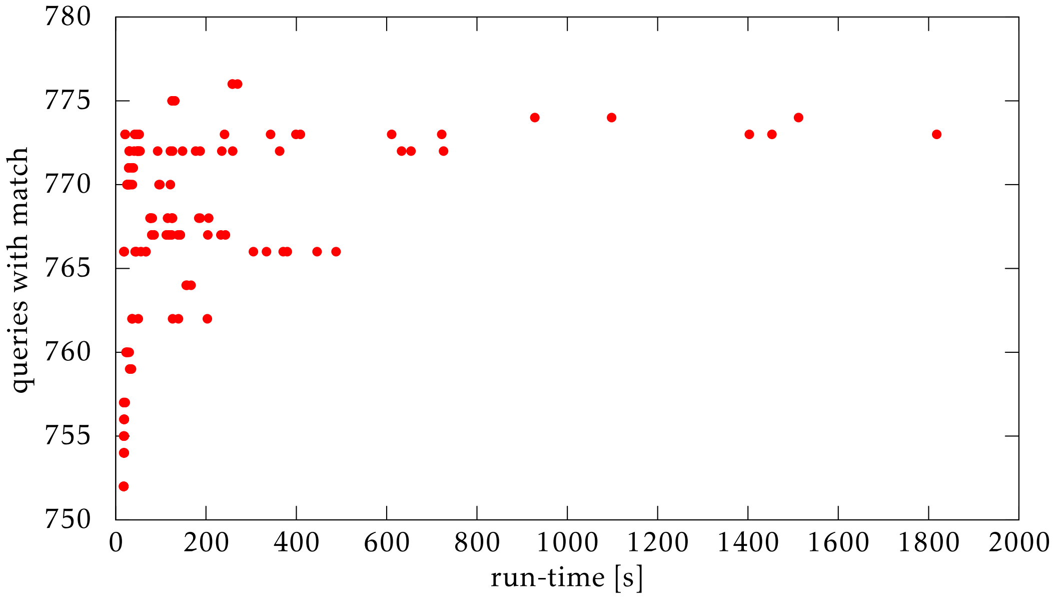


***Figure B.1:*** *Matched queries vs. run-time (dataset I) [uncropped]*

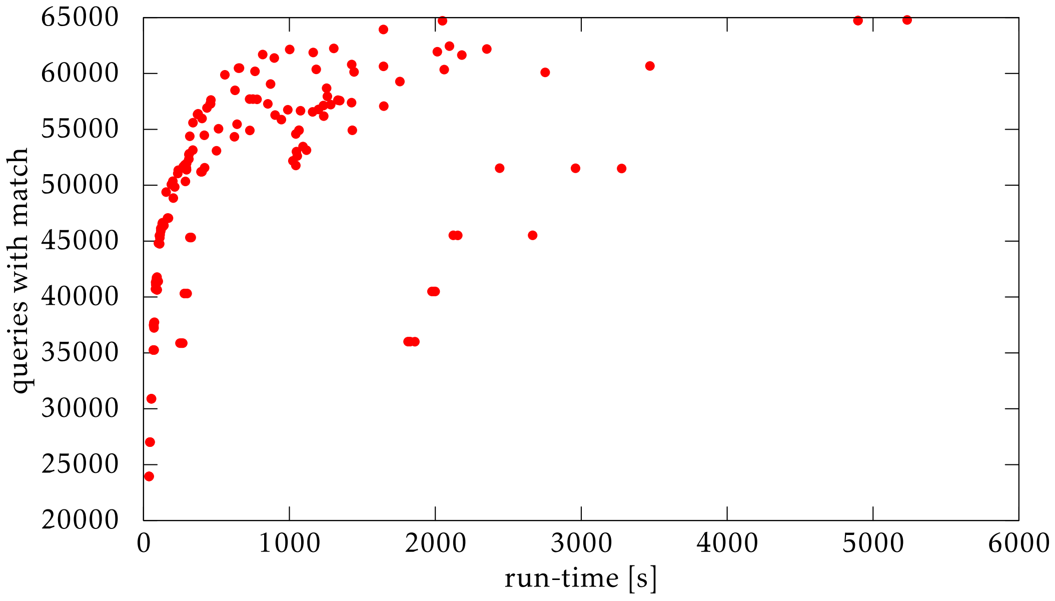


***Figure B.2:*** *Matched queries vs. run-time (dataset II) [uncropped]*