Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Bachelor Thesis

Efficient Preprocessing of Short Sequence Segments for Efficient Filtering in Comparing Protein Sequences

Submitted by

Anh Viet Ta
Student number 6 747 004

in the study program

B.Sc. Computing in Science
at the Center for Bioinformatics Hamburg
and the Department of Informatics
of the MIN Faculty

November 24, 2023

1. Examiner: Stefan Kurtz
2. Examiner: Andrew Torda

Abstract

The software suite MMseqs2 and its predecessor MMseqs are the new standard of both sensitive biological sequence searching and sequence clustering. The prefiltering module in MMseqs2 is of particular importance, since it serves to rapidly reduce possibility space. This paper looks at the first q-gram generation stage of the module and proposes some possible optimization approaches.

This reference is only here to fill the bibliography: [1].

Contents

Chapter 1.

1 Recent developments of bioinformatic sequence searching and clustering

With the advent of high-throughput sequencing technologies, the cost of genome sequencing has come down sgnificantly. Sequence databases, such as UniProt have been growing by a factor of two every two years, which leads to a significant focus on developing searching and clustering methods that can handle large-scale datasets efficiently. Algorithms and software that can scale horizontally (across multiple machines) have gained importance. On sequence searching, new algorithms and heuristics have been developed to improve sensitivity without sacrificing speed. Tools like DIAMOND and MMseqs2 provide fast and sensitive sequence searching capabilities, especially in metagenomics and large-scale sequencing projects. HMM-based methods, such as the HMMER software, have been enhanced to improve their sensitivity and accuracy, making them invaluable for protein domain and family searching. The existing algorithms like BLAST have also seen massive improvement by using GPU parallelization and specific hardwares, namely FPGAs (Field-Programmable Gate Arrays) have been customized for accelerating sequence searching algorithms. Recent developments on metagenomics have also given ways to specialized databases and algorithms. These databases contain sequences from environmental samples and enable the identification of novel organisms and genes within complex microbial communities. On sequence clustering, myriads of methods have been developed. graph-based clustering methods, such as Markov Clustering (MCL) and Louvain algorithm, have gained popularity. These methods model sequences as nodes in a

graph and use edge weights to represent similarities, allowing for the detection of densely connected clusters within the graph. Density-based methods like DBSCAN (Density-Based Spatial Clustering of Applications with Noise) have been applied in bioinformatics. These algorithms group sequences based on the density of data points, enabling the discovery of clusters with varying shapes and sizes. Traditional distance-based clustering algorithms, such as hierarchical clustering and k-means, have been adapted and optimized for large biological datasets. Efficient distance metrics and clustering strategies have been a focus of research.

Chapter 2.

2 MMseqs2

MMseqs (Many-against-Many sequence searching) is a software suite for fast and deep clustering and searching of large datasets. It was published on January 6 2 016 by Maria Hauser, Martin Steinegger and Johannes Söding. It contains three core modules: a fast and sensitive prefiltering module that eliminates most non-homologous sequences, an local alignment module based on the striped Gotoh algorithm, and a clustering module based on similary graph. Due to the modularity of the structure, it allows user to create workflows tailored to specific clustering and searching needs. The default workflow utilizes the UniProt databases to create predefined parameters and resembles the average use case. The second, cascaded clustering, clusters the input database in multiple steps, increasing the sensitivity between each steps and therefore enables finding best-hit-searches 4-10 times faster. The third workflow simply compares databases using only prefiltering and aligning modules. The last workflow updates clustering between new and old clustered database by deleting deprecated sequences and appending new sequences to each cluster. In benchmark, MMseqs showed to be 4-30 times faster than UBLAST and RAPsearch2. MMseqs could cluster large databases down to 30% sequence identity 2 000 times the speed of BLASTclust. An improved, updated version MMseqs2 was published on June 7 2 017 . It boasts dramatic improvements in both efficiency and sensitivity by introducing several novel ideas. It revamped the prefiltering stage by introducing algorithm to find two consecutive, inexact q-gram matches and optimizing memory access. It also allowed the use of 7-gram due to the speedup of the algorithm. An in-depth look into the three core modules is presented below.

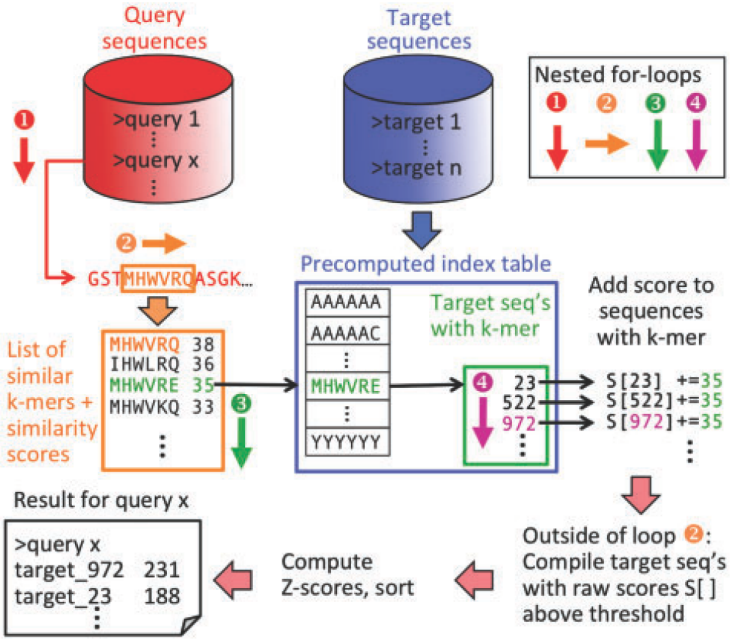

Figure 2.1.: Prefiltering stage in MMseqs

2.1. Fast q-gram prefiltering stage

In MMseqs, the prefiltering stage is the first step to process all sequences and serves to reduce the search space significantly, therefore it's imperative for the algorithm to be as efficient as possible. The process consists of 4 loops as outlined in Figure 2.1. In the first loop, query sequences are searched one by one against the target set. For each spaced seed window in the query sequence (loop 2), a q-gram is extracted and a list of all similar q-grams with a Blosum62 similarity above a threshold score is created using a linear-time branch-and-bound algorithm. Lower threshold scores result in higher average numbers of similar k-mers and thereby higher sensitivity and lower speed. Each generated q-gram (loop 3) then gets queried against a precomputed database of q-grams in target sequences. The sum of scores of the corresponding target sequences then gets collected in a vector based on the prefiltering score. In the last step the target score vector is sorted and the best matches get processed further. In MMseqs2, the prefiltering stage was reengineered (See Figure 2.2). After the query-target matches are collected, they get sorted and iterated over to detect double k-mer matches by comparing the current diagonal with the previously matched diagonal. If the previous and current diagonals agree, they both get stored as a double match for the ungapped alignment stage.

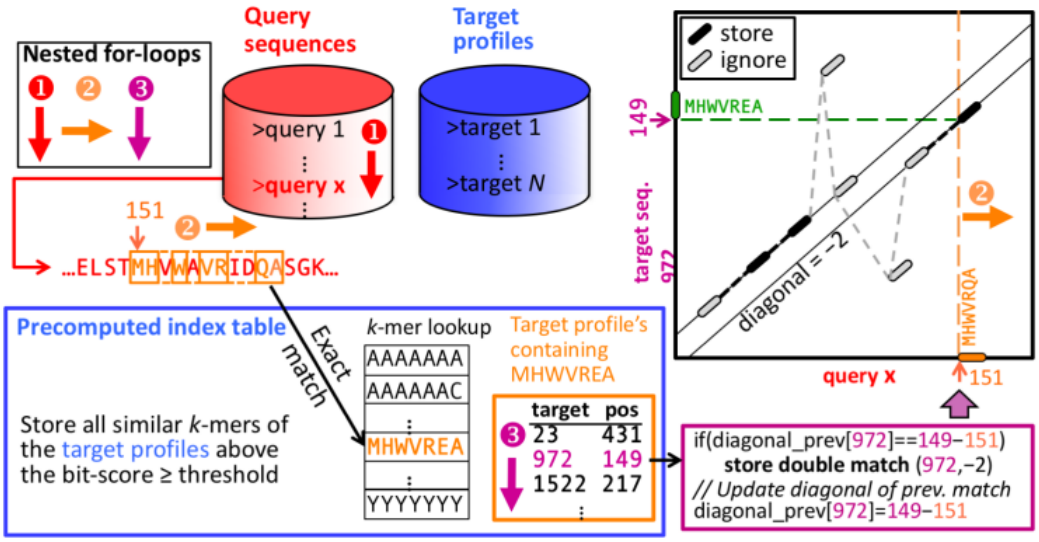

Figure 2.2.: Prefiltering stage in MMseqs2

2.2. Ungapped alignment and Smith-Waterman alignment stage

In MMseqs2, an additional ungapped alignment stage was introduced, where many target sequences are aligned at once using a vectorized approach. It also utilizes a linear memory access using a score matrix containing the p. Due to the extensive use of SIMD instructions, this stage achieves a linear time complexity, much more efficient compared to the Smith-Waterman alignment stage. After the ungapped alignment is completed, MMseqs2 computes for all filtered query-target sequence pairs an exact, unbounded Smith-Waterman alignments with affine gap penalties. This module utilizes a highly vectorized algorithm equipped with new SIMD instructions and adapted for sequence profiles.

3 Chapter 3. Theoretical background

Since for every alphabet $\mathcal{A}$, a transformer function $f : \mathcal{A} \to \mathcal{N}$ can be defined, mapping every character in $\mathcal{A}$ to a integer rank, the following sections will discuss the biological sequences not as sequences of residues, but as of transformed, coded integers. The alphabet will also be interpreted mostly as an integer subset $[0, |\mathcal{A}|) \cap \mathcal{N}$.

3.1. Optimizing target data processing

Since the target database can be very large, a precise and efficient evaluation must be prioritized. In MMseqs2, at every residue position, relevant characters based on the user-chosen spaced seed are individually extracted from the target sequence to create a q-grams and then process these q-grams as independent strings. This approach would incur a time complexity of $O(q)$ to compute a hash value for each q-gram and for the complete sequence of length n this would sum up to $O((n-q+1)q) = O(nq)$. But the q-grams are in fact not independent from each other. To create the next q-gram, one would only have to remove the first character of the previous q-gram and append the next character in the sequence. Assuming these operations are of constant time complexity, the whole process would require only $O(n+q)$ time. This family of hashing algorithms allows efficiently enumerate hash values of all q-grams of a sequence. In order to achieve independence between hash values, the characters are usually assigned a weight, which might be expressed as r^{q-1-i}, where r, the

radix, is a constant integer and i is the position of the character in the q-gram. The calculation of a hash value from a q-gram w could then be expressed as follows:

$$H(w) = \sum_{i=0}^{q-1} r^{q-1-i} w[i] \tag{3.1}$$

where w is a q-gram and $w[i]$ is the i-th character in w. As a base case of the recursive algorithm, the hash value for the first q-gram of the sequence is computed in $O(k)$ time by evaluating the sum defined in Equation (3.1). Provided we have calculated the hash value of the previous q-gram ax, where a is a character and x is a $q-1$-gram. Then the hash value of the next q-gram could be computed from $H(ax)$ and the next not yet processed character, say c. Firstly, the contribution of a character is subtracted from the hash value. The virtual window is then shifted right one character. This means that the weight of all characters in x increases by a factor or r. That is, we multiply by the radix. Lastly, the new character c is added to the hash. The hash value of xc is then calculated as follows:

$$H(xc) = (H(ax) - r^{n-1} \cdot a) \cdot r + c \tag{3.2}$$

Equations (3.1) and (3.2) provide the basic schema of recursive hashing. In implementation, the most basic form of the algorithm, called invertible integer encoding, is used, where $r = |\mathcal{A}|$. The pseudocode for this schema is outlined in Algorithm 1. In order to extend the schema to allow for spaced seeds, one could divide the seed into shorter blocks of q-grams, intertwined by blocks of "Don't Care". Each block can then be treated as individual q-gram with custom radix, and the complexity is then $O(nb)$, where b is the number of blocks. Since the number of blocks is in worst case the weight itself, and in best case 1, recursive hashing has a same or faster time complexity than normal hashing approachs.

3.2. Optimizing query data processing

In order to quantify the similarity between sequences, a scoring function is often used. They provide quantitative measures for comparing biological sequences, structures, and interactions, facilitating a deeper understanding of biological processes and aiding in drug discovery, evolutionary analysis, and functional genomics research. In sequence alignment context, these functions often compare pairwise the biological residues (often nucleotide bases or proteinogenic amino acids). The most common scoring function is the substitution matrix, often represented as a matrix of scores indicating the likelihood of one amino acid (or nucleotide) being replaced by another.

Algorithm 1 Invertible Integer Encoding

Input: Encoded sequence s of length n
alphabet size $r = |\mathcal{A}|$

```
HashValVec ← []
HashValue ← 0
for i from 1 to k do                          ▷ Compute 1st hash
    HashValue ← HashValue · r
    HashValue ← HashValue + s[i]
end for
HashValVec.append(HashValue)
for j from 1 to n − k do                      ▷ Compute other hash values
    HashValue ← HashValue − r^(n−1) · s[j]
    HashValue ← HashValue · r
    HashValue ← HashValue + s[j + k]
    HashValVec.append(HashValue)
end for
```

The BLOSUM (Blocks Substitution Matrix) and PAM (Point Accepted Mutation) matrices are examples of substitution matrices widely used in bioinformatics. The two families of score matrices can be summarized as a mathematical function:

$$\sigma : \mathcal{A} \times \mathcal{A} \to \mathcal{R} \tag{3.3}$$

A positive score between two residues often denotes similarity between them and a negative score indicates dissimilarity. The score of between two sequences $u, v \in \mathcal{A}^q$ can then be defined as follow:

$$\sigma(u,v) = \sum_{i=0}^{|u|-1} \sigma(u[i], v[i]) \tag{3.4}$$

In order to compute the k-environment for a q-gram u efficiently, MMseqs2 utilizes a branch-and-bound algorithm, where it decomposes u to groups of 2-grams and 3-grams u_i, prioritizing 3-gram. For $q \in \{2,3\}$, a score matrix $\mathsf{ST}_q[u] = [(v, \sigma(u,v)) \mid v \in \mathcal{A}^q]$ is computed and sorted after the subscores $\sigma(u,v))$. The full k-environment of a q-gram u can then be iterated as a subset of the cartesian product $\bigtimes \mathsf{ST}_{|u_i|}[u_i]$:

$$\mathrm{Env}_k(u) = \{(v, \sigma(u,v)) | (v, \sigma(u,v)) \in \bigtimes \mathsf{ST}_{|u_i|}[u_i], \sigma(u,v) \geq k\} \tag{3.5}$$

The iteration over score matrix row can be prematurely terminated, as soon as it's no longer feasible to reach the threshold k. The key to improve over the method outlined

in MMseqs2, is the following: Given a bijective mapping $\varphi : \{0,1,\ldots,q-1\} \rightarrow \{0,1,\ldots,q-1\}$. For any sequence of length q, we define a function

$$\varphi(u) = u[\varphi(0)]u[\varphi(1)]\ldots u[\varphi(q-1)] \tag{3.6}$$

in that, φ permutate the residue in u.

Lemma 1 Given a special permutation φ_u, so that it orders the characters in u:

$$u[\varphi_u(i)] \leq u[\varphi_u(i+1)] \forall i, 0 \leq i \leq q-2 \tag{3.7}$$

The resulted q-gram u_s from the permutation is called sorted q-gram and could be rearranged using the inverse function:

$$\varphi_u^{-1}(\varphi_u(i)) = i \forall i, 0 \leq i \leq q-1 \Leftrightarrow \varphi_u^{-1}(\varphi_u(u)) = u \tag{3.8}$$

Then the following property is valid: Given a q-gram u, its sorting permutation φ_u and the mapping $u[i] = (\varphi_u(u))[j]$:

$$\sigma(\varphi_u(u), \varphi_u(v)) = \sigma(u,v) \tag{3.9}$$

Proof:

$$\sigma(\varphi_u(u), \varphi_u(v)) = \sum_{j=0}^{q-1} \sigma((\varphi_u(u))[j], (\varphi_u(v))[j]) \tag{3.10}$$

$$= \sum_{i=0}^{q-1} \sigma(u[i], v[i]) \tag{3.11}$$

$$= \sigma(u,v) \tag{3.12}$$

The exchange of variable from Equation 3.10 to Equation 3.11 is allowed due to the bijective nature of function σ and the fact that its domain and codomain are identical.

The above proof shows that a permutation function applied on both sequences u and v permutates the characters in the same way and the score between the unpermutated sequences is equal to the score between the permutated. This means that the score matrix ST can be computed only for the sorted q-grams, and it would hold all information needed to compute the full k-environment. The advantage of this approach is the smaller number of sorted q-grams compared to unsorted, leading to a more efficient memory usage and a more compacted computation of the score matrix. Any overhead caused by the rearragement of the sorted q-grams could be reduced by SIMD (Single-Instruction Multiple-Data) instructions.

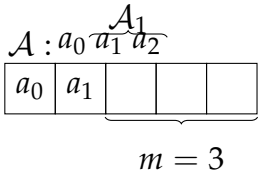

Figure 3.1.: Example of sorted q-gram enumerating

3.3. Working with sorted q-grams

Enumeration

In mathematics, a set is defined as a collection of items, where every element occurs exactly once. This definition can be expanded to multisets, where elements can occur more than once. Applying an order on these multisets, we can formalize a notion of sorted q-grams: Given an alphabet $\mathcal{A}$ and a natural number $q > 0$, a sorted q-gram over $\mathcal{A}$ is a multiset $M = a_0a_1...a_{q-1}$, $a_i \in \mathcal{A} \forall 0 \leq i < q$ and $a_i \leq a_j \forall 0 \leq i < j < q$. q can be computed as the binomial $\binom{q+|\mathcal{A}|-1}{q}$. The proof can be shown with induction: An index table of sorted q-grams can be computed recursively. Figure 3.1 showcases a model of the problem where $q = 5$ and $\mathcal{A} = \{a_0, a_1, a_2\}$. If the prefix of length 2 of u is fixed, say a_0a_1, the last 3 characters can be iterated as a sorted q-gram of size $5 - 2 = 3$ and alphabet $\mathcal{A}_1 = \{a_1, a_2\}$. This could be broken down further and generalized. Given a sorted q-gram u where the prefix of m characters is sorted and fixed, and $u[m-1]$ has a rank of a_i then the task is to enumerate every $q - m$-grams of alphabet size $(|\mathcal{A}| - a_i)$. The induction base case is then sorted unigrams over alphabet $\mathcal{A}_i = \{a_j | a_j \in \mathcal{A} \wedge j \geq i\}$, where there are $|\mathcal{A}| - i$ unigrams indexed from a_i to $|\mathcal{A}| - 1$.

Linear encoding

Using the index table, a scheme to encode any given q-gram over alphabet $\mathcal{A}$ to its table index in $O(q)$ can be devised.

Lemma 2 There exists an unique integer weight for each character a in position p so that given any sorted q-gram u over an alphabet $\mathcal{A}$,

$$c(u) = \sum_{p=0}^{q-1} w(p, u[p]) \tag{3.13}$$

Algorithm 2 MultisetEnumerateRecursion

Input: current multiset length q_i
current alphabet $\mathcal{A}_i$
current prefix u
current index table $\mathsf{IndexTable}_{q,\mathcal{A}}$
if $q_i = 0$ **then**
$\mathsf{IndexTable}_{q,\mathcal{A}}.\mathsf{append}(u)$
else
for $a_j \in \mathcal{A}_i$ **do**
$\mathsf{MultisetEnumerateRecursion}(q_i - 1, \mathcal{A}_i \backslash [a_i, a_j), u + a_j, \mathsf{IndexTable}_{q,\mathcal{A}}$
end for
end if

Algorithm 3 Enumerate Multiset $\mathsf{IndexTable}_{q,\mathcal{A}}$

Input: multiset length q
alphabet $\mathcal{A}$
$\mathsf{IndexTable}_{q,\mathcal{A}} \leftarrow []$
for $a_i \in \mathcal{A}$ **do**
$\mathsf{MultisetEnumerateRecursion}(q - 1, \mathcal{A} \backslash [a_0, a_i), a_i, \mathsf{IndexTable}_{q,\mathcal{A}})$
end for

encodes exactly u to its index in table IT.

Proof: Base case: In case of $q = 1$, $|\mathcal{A}|$ unigrams can be clearly encoded with $w(0,a) = a \forall a \in \mathcal{A}$. **Inductive step:** Given the above scheme is valid up until $q-1, q \in \mathcal{N}, q \geq 2$, it needs to be shown $w(q,a)$ is unique for all $a \in \mathcal{A}$. Assuming the weight isn't unique, therefore there exist two different weights $w_a \neq w'_a$ so that $c(au) = w_a + c(u)$ and $c(av) = w'_a + c(v)$ encode q-grams au and av respectively, $u, v \in \mathcal{A}^{q-1}, a \in \mathcal{A}$. The codes $c(au)$ and $c(av)$ must but differ exactly the code of their suffixes $c(u) - c(v)$, since they share the same prefix. One can then formulate:

$$c(au) - c(av) = (w_a + c(u)) - (w'_a + c(v)) \tag{3.14}$$
$$= (w_a - w'_a) + (c(u) - c(v)) \tag{3.15}$$
$$= c(u) - c(v) \tag{3.16}$$

which leads to $w_a - w'_a = 0$, or $w_a = w'_a$, which is a contradiction. Therefore $w(q,a)$ must be unique.

The recursive method to compute a $q \times |\mathcal{A}|$ table LE is then deduced based on the above proof. The base case can be directly given and based on the index tables

Algorithm 4 Creating linear encoding table $\mathsf{LinearEnc}_{q,\mathcal{A}}$

Input: sequence length q
alphabet $\mathcal{A}$
index tables $\mathrm{IT}_{q_i,\mathcal{A}} \forall q_i \in [1,q]$
$\mathsf{LinearEnc}_{q,\mathcal{A}} \leftarrow []$
$\mathsf{LinearEnc}_{q,\mathcal{A}}.\mathsf{append}([0,...,|\mathcal{A}|-1])$
for $q_i \in [2,q]$ **do**
  $\mathsf{CurrentWeight} \leftarrow [None \times |\mathcal{A}|]$
  for $i,u \in \mathrm{IT}_{q_i,\mathcal{A}}$ **do** ▷ Key-Value loop
    if $\mathsf{CurrentWeight}[u[0]] = None$ **then**
      $\mathsf{SuffixCode} \leftarrow 0$
      for $j \in [1,q_i]$ **do**
        $\mathsf{SuffixCode} += \mathsf{LinearEnc}_{q,\mathcal{A}}[j-1][u[j]]$
      end for
      $\mathsf{CurrentWeight}[u[0]] \leftarrow i - \mathsf{SuffixCode}$
    end if
  end for
  $\mathsf{LinearEnc}_{q,\mathcal{A}}.\mathsf{insert}(0,\mathsf{CurrentWeight})$ ▷ Place more significant weight at beginning
end for

$\mathrm{IT}_{q_i,\mathcal{A}}, 2 \leq q_i \leq q$, the q_i-grams are tracked for change in prefix and the weight of the prefix can be calculated with Equation 3.13. The pseudocode for the construction of LE is outlined in Algorithm 4.

3.4. Simplified approach in determining threshold

In MMseqs, in order to evaluate a proper threshold k for the environment, a z-score statistics was applied. For each query sequence, a calibration search through a subset of 100 000 randomly sampled target sequences is performed, where the number of prefiltering operations

$$\mathrm{sum}_L = \sum_{t=1}^{100\,000} (L_t - k + 1) \tag{3.17}$$

and its sum of scores

$$\mathrm{sum}_S = \sum_{t=1}^{100\,000} S_{qt} \tag{3.18}$$

are recorded. L_t here denotes the length of the target sequence t and S_{qt} the prefiltering score between query sequence s and target sequence t. The expected chance prefiltering score between them is then

$$S_0 = (L_t - k + 1)\frac{\text{sum}_S}{\text{sum}_L} \tag{3.19}$$

Assuming the number of q-gram matches is Poisson-distributed, the standard deviation of the scores σ_S can be computed through number of expected q-gram matches n_{match}:

$$n_{\text{match}} \approx \frac{S_0}{S_{\text{match}}} \tag{3.20}$$

$$\sigma_S = S_{\text{match}}\sqrt{n_{\text{match}}} \tag{3.21}$$

$$= \sqrt{(L_t - k + 1)\frac{\text{sum}_S}{\text{sum}_L}S_{\text{match}}} \tag{3.22}$$

where S_{match} is the expected score per chance q-gram match. The significant prefiltering score S_{qt} should then fulfill the condition

$$S_{qt} \geq Z_{thr}\sigma_S + S_0 \tag{3.23}$$

where Z_{thr} is the significant z-score. MMseqs2 would take in a sensitivity parameter s, labeled internally as the average length of q-gram list each sequence position, from the user and using a heuristic to compute for S_{qt}. This approach is shown to be lacking in control for s and therefore the goal is to streamline the process and to allow for a more fine-grained control of the length of q-gram list. **A context-free approach:** The average q-gram list l and the threshold k of the environment can be related through the right tail portion of a discrete distribution of unsorted-q-gram-vs-unsorted-q-gram score h_q. Figure 3.2 visualizes an exemplary score distribution between uniformly distributed 7-gram. The red portion in the right tail describes the probability $\mathcal{P}(\sigma \geq 0)$, which could be generally quantified to:

$$\mathcal{P}(\sigma \geq k) = \sum_{i=k}^{\infty} h_q(i) \tag{3.24}$$

The average q-gram list length l could be normalized to probability:

$$p = \frac{l}{|\mathcal{A}|^q} \tag{3.25}$$

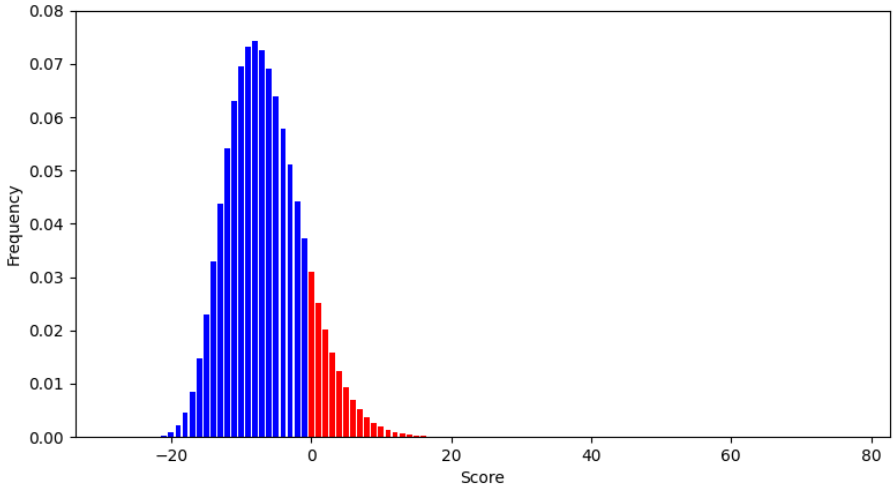

Figure 3.2.: Example of score distribution h_7 between uniformly distributed 7-gram. Blue denotes portion of the distribution where $\sigma < 0$. Red denotes portion of the distribution where $\sigma \geq 0$.

Therefore the approximation would be ideally in continuous distribution an exact value:

$$p = \mathcal{P}(\sigma \geq k) \tag{3.26}$$

or in discrete distribution, a value between two discrete values:

$$\mathcal{P}(\sigma \geq k) \leq p < \mathcal{P}(\sigma \geq k - 1) \tag{3.27}$$

The idea of the algorithm is to approximate a $q \times q$ score distribution. Then the distribution can be iterated from the right tail to calculate $\mathcal{P}(\sigma \geq k)$, where k is the current score threshold. The iteration can be stopped when Equation 3.27 is satisfied. The $q \times q$ score distribution can be approximated recursively through convolution:

$$h_q(x) = \sum_{y \in \mathcal{Z}} h_{q-1}(y) h_1(x - y) \tag{3.28}$$

It is important to note that h_1, the unigram score distribution, depends on both character distribution collected from target sequence and uniform character distribution, since the characters appeared in the environment are independent from each other.
Integrating context: The above strategy treats single character as independent and therefore loses its context in q-grams. Another approach could then be hashing the

target sequences to collect q-gram distribution, which is convenient since the work is done in the preprocessing of target dabase. This approach is but often complicated by large q, since a $q \times q$ score matrix must be computed, which incur an additional penalty of $O(|\mathcal{A}|^{2q})$, which is often not feasible when $q > 4$. A middle ground is to divide the q-grams into two or three sub-q-grams and collect their distribution separately. These distributions can then in the end be convoluted together to approximate the q-gram distribution. The extraction of threshold k then follows the same approach in the context-free variant.

Chapter 4.

4 Implementation and Design

4.1. Design overview

The program consists of seven main classes:

1. Seed readers convert seed into binary representation, computing its span and weight. To further process seed, two variants are created:

   a) Target reader divides seeds into blocks of shorter q-grams as input for recursive hashing function.

   b) Query reader divides seeds into blocks of short segments, sorts query sequences and uses sorted q-gram encoder to linearly encode the resulted sorted q-grams.

2. A recursive hash engine to collect every q-gram hashes from target sequences and package them.

3. A sorted q-gram index table to index every sorted q-gram of given scoring function and q.

4. Sorted q-gram encoder creates a table to help linearly encode sorted q-grams.

5. A class to approximate an appropriate threshold given target sequence data.

6. A score matrix to cache all scores between sorted q-grams and unsorted q-grams.

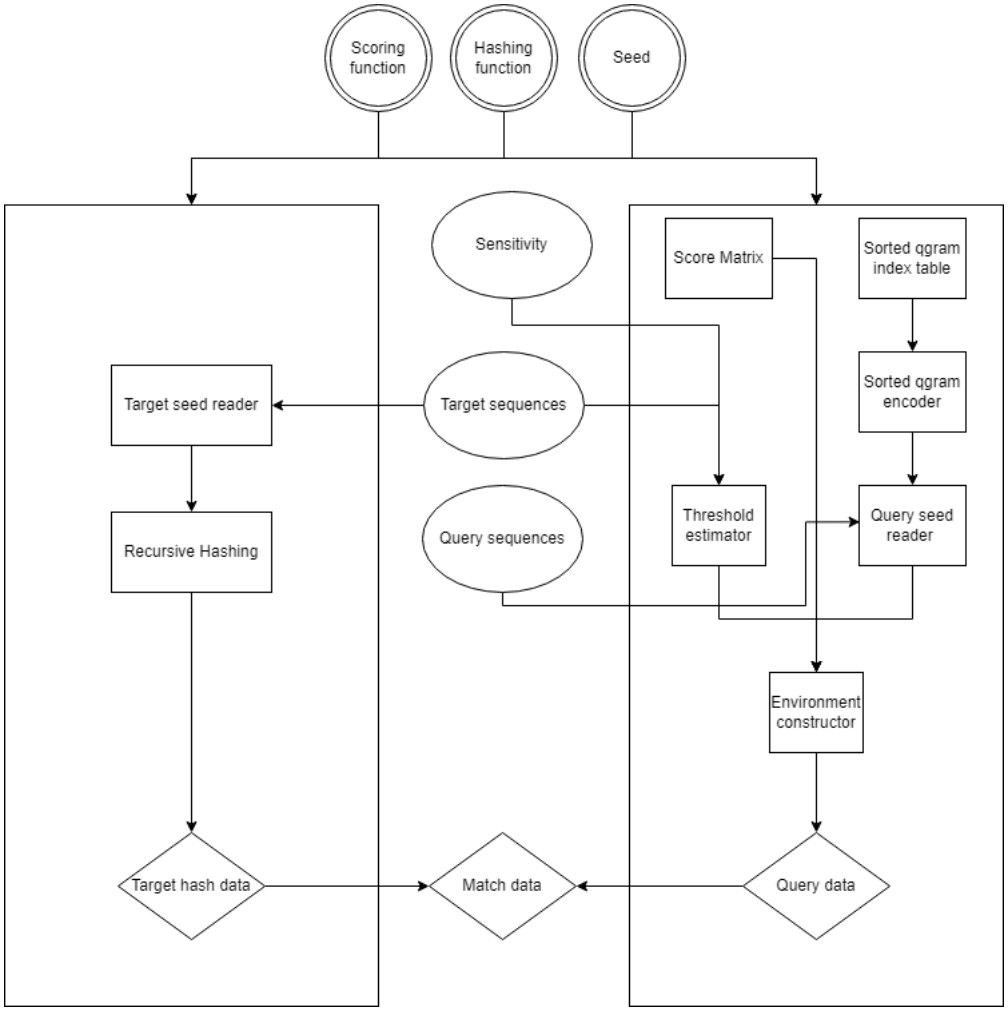

Figure 4.1.: Relationship between the main classes and inputs

7. An environment constructor gets sorted q-gram codes from the query seed reader, uses them to call individual scores from the score matrices and generates a Cartesian product from the environments.

In Figure 4.1, the relationship and dataflow between the classes are outlined. Since the implementation of the recursive hashing of target sequences is straightforward and doesn't diverge much from pseudocode, the section below focuses mainly on the processing of query sequences.

4.2. Input

At compile time, the program takes in a scoring function, which in the context of protein sequences could be a BLOSUM or PAM matrix, a spaced seed and a recursive hashing function. The scoring function should detail the relevant alphabet, the transformer function, which encodes every character in the alphabet to its corresponding rank, and a matrix of scores between every pairs of characters. Internally, the spaced seed is initially represented as a numeric constant, which during computation will be transcoded into a bitset. The span of the seed can then be computed according to the position of the first 1 in the bitset and also the seed weight can be calculated by counting 1s. At run time the program takes in two sequence databases in FASTA format. Any data error, for example empty file or wrong data format will automatically lead to termination of the program and an error message will be logged. Additionally, the sensitivity of the program can be adjusted. To further assist in expansion, some fixed parameters, e.g. maximum sub-q-gram length are defined as preprocessor constants.

4.3. Compile time computation

Figure 4.1 shows that some classes/tables can be evaluated at compile time. Each seed gets transcoded and its weight and span can be precomputed and therefore, the two seed readers and their schematics can be predetermined. The two index tables for sorted/unsorted digrams/trigrams take in only scoring function as parameter and therefore, can be wholely precomputed. Since the linear encoding table LE (see Algorithm 4) based solely on the sorted digrams/trigrams index table, it also can be evaluated at compilation.

4.4. Run time computation

At run time, firstly the target and query database gets encoded using the transformer function. Using the target seed reader, the target sequences can then be hashed in linear $O(nb)$ time and the hashes are packaged as byte units and saved in a contiguous container (i.e. array or vector). In the processing of query sequences, firstly the 2-gram and 3-gram score matrices ST can be evaluated using the index tables from sorted and unsorted q-grams according to Equation 3.4. The result then can then be stored as a pair of score and unsorted q-gram code and sorted after score value for further use. Thereafter, an appropriate threshold is evaluated using

the target sequences and an user-provided sensitivity. Using the score matrices and the threshold, the program enters the step of q-gram generation using the Cartesian product.

Local composition bias correction

In the implementation of MMseqs2, a mechanism to evaluate the surrounding of current position while iterating the query sequences is introduced. The idea of this mechanism is to adjust the threshold of the q-gram environment in response to regions where local composition varies considerably from the background distribution. Without adjusting, these low regions can lead to biases in the prefiltering result. This correction of the threshold can be summarized as follow:

$$\Delta\sigma_i(u[i]) = \sum_{a=1}^{|\mathcal{A}|} f(a)\sigma(a, u[i]) - \frac{1}{2d} \sum_{j=i-d, j\neq i}^{i+d} \sigma(u[i], u[j]) \tag{4.1}$$

where u is the query sequence, i the current residue on the sequence and $f(a)$ the background frequency of residue a. The minuend is a representation of the expected score resulting from the background distribution, which can be precomputed when the target data is read. The subtrahend involves the current region in the query sequence and introduces a parameter d for the radius of the region, defined in MMseqs2 as a constant 20. The corrected score between a query q-gram u and its generated q-gram v can then be computed as the sum of the pairwise amino acid score and the score correction.

$$\sigma_c(u, v) = \sigma(u, v) + \sum_{i=0}^{q-1} \sigma_i(u[i]) \tag{4.2}$$

In the implementation, this correction is instead subtracted from the threshold at the beginning of the computation.

$$\sigma_c(u, v) = \sigma(u, v) + \sum_{i=0}^{q-1} \sigma_i(u[i]) \geq k \Leftrightarrow \sigma(u, v) \geq k - \sum_{i=0}^{q-1} \sigma_i(u[i]) = k_c \tag{4.3}$$

Turning Wheel Mechanism

An issue in enumerating the Cartesian product is the possible difference in number of loops (e.g. 2 loops for seed weight 4-6, 3 loops for seed weight 7). To resolve this problem and allow easy expansion to greater seed weight, a flexible loop structure is

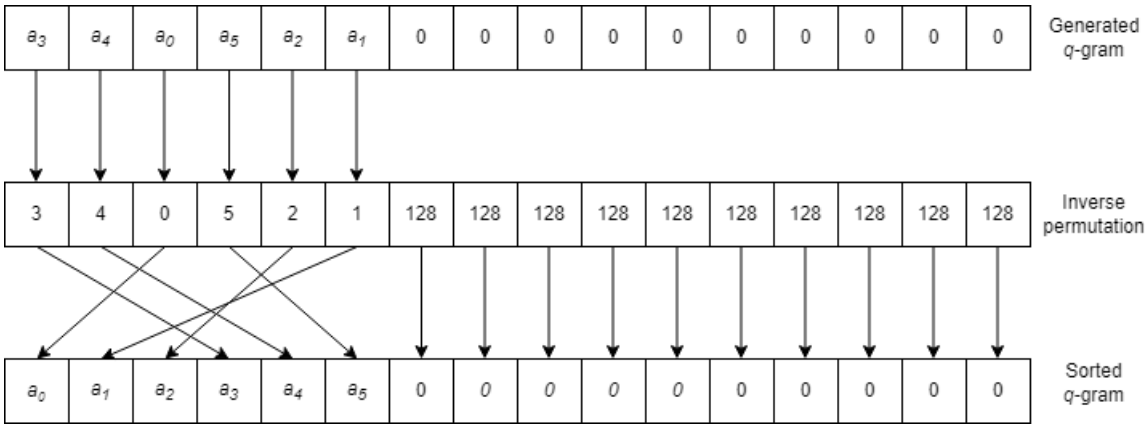

Figure 4.2.: Schematics of SIMD instruction __mm_shuffle_epi8

designed, where an array containing the loop indexes is created. The earliest entry of the array holds the outermost loop index and the later an entry is, the more inner the loop index the entry holds. By iterating through only the innermost loop and only adjusting the outer loop indexes as needed, the scheme can account for any number of loops. The formulation of the loop structure, is outlined in the pseudocode 5.

SIMD

SIMD (Single-Instruction Multiple-Data) is a group of instructions allowing vectorized, parallel data processing. Instead of processing data sequentially, they are capable of defining wide registers, commonly 128-bit or 256-bit, packaging data in those registers and process data in parallel. The key to the shuffle using SIMD is the instruction `__mm_shuffle_epi8`, which is available on platforms supporting SSSE3 instruction sets or the equivalent `vqtbl1q_u8` on ARM platform. Both instructions take in two 128-bit (equivalent to 16-byte) vectors. In order to comply with the instructions, both the q-gram and the inverse permutation get padded internally to 16-byte. First vector holds the data, in this case the q-gram, and the second the information on how the data get shuffled, in this case the inverse permutation. A schematic is visualized in Figure 4.2. The right padded section of the inverse permutation is filled with integer 128 to signal the instruction to not shuffle and to simply pad the resulted vector with 0.

4.5. Merging target and query data

After being collected, the hashed data from the target and query sequences are packaged as byte units, prioritizing hash values. They then get sorted individually using radix sort (see GTTL), resulting in two data vectors sorted by hash values which would then be merged value by value, skipping through unmatching blocks. The merging process results in a vector of matches, represented again as byte units, containing respectively the sequence number of the match on the target, on the query, the diagonal number (difference between the target and query sequence position), and the query sequence position. These quartets are again sorted with radix sort to prepare for ungapped alignment stage.

Algorithm 5 Turning Wheel Mechanism

Input: number of loops n
loop data vector v_0, v_1, ..., v_{n-1}
threshold k
function createQgram
function getScore
function getCode

```
loopid ← n − 1
wheel ← [0] × n
score ← getScore(wheel)
if score < k then                    ▷ Current data vectors not viable. Early Termination.
    return
else                                 ▷ First loop iteration
    code ← getCode(wheel)
    createQgram(code)
end if
while True do
    wheel[loopid] ← wheel[loopid] + 1
    if wheel[loopid] ≥ len(v_loopid) then                    ▷ End of loop
        wheel[loopid] ← 0
        if loopid = 0 then                    ▷ No longer viable in outermost loop
            break
        end if
        loopid ← loopid − 1
    else
        score ← getScore(wheel)
        if score ≥ k then                    ▷ Viable q-gram found
            code ← getCode(wheel)
            createQgram(code)
        else                    ▷ No viable q-gram found, end current loop
            wheel[loopid] ← len(v_loopid)
        end if
        if loopid ≠ n − 1 then                    ▷ Reset to innermost loop
            loopid ← n − 1
        end if
    end if
end while
```

Algorithm 6 Merging query and target data

```
Input: query data Query
       target data Target
       function CreateMatch
  TargetIdx ← 0
  QueryIdx ← 0
  while TargetIdx ≠ Target.end ∧ QueryIdx ≠ Query.end do
    HashValue_t ← Target[TargetIdx].HashValue
    HashValue_q ← Query[QueryIdx].HashValue
    if HashValue_t < HashValue_q then                ▷ Skipping on target vector
      while Target[TargetIdx].HashValue = HashValue_t do
        TargetIdx ← TargetIdx + 1
      end while
    else
      if HashValue_t > HashValue_q then              ▷ Skipping on query vector
        while Query[QueryIdx].HashValue = HashValue_q do
          QueryIdx ← QueryIdx + 1
        end while
      else                                           ▷ Match found
        TargetBlockEnd = TargetIdx
        while Target[TargetBlockEnd] = HashValue_t do
          TargetBlockEnd ← TargetBlockEnd + 1
        end while
        QueryBlockEnd = QueryIdx
        while Query[QueryBlockEnd] = HashValue_q do
          QueryBlockEnd ← QueryBlockEnd + 1
        end while
        CreateMatch([TargetIdx, TargetBlockEnd), [QueryIdx, QueryBlockEnd))
        TargetIdx ← TargetBlockEnd
        QueryIdx ← QueryBlockEnd
      end if
    end if
  end while
```

5 Chapter 5. Benchmark and Performance

Separate modular benchmarks were carried out for the processing of target sequences, threshold estimation and similar q-gram generation. All measurements were performed using C++ on an Intel CPU i5-10 300H 2.5 GHz with 8 GB DDR4 running Ubuntu on a SSD. The program was compiled with g++ 11.3.0. The test dataset composed of a target database from MMseqs2 and a query database created as a difference set from MMseqs2 queryset against the target database. Some statistics on these datasets are summarized below in Table A.1. Table A.2 outlines the spaced seeds used in testing, which are taken from MMseqs2.

5.1. Processing target sequences

Testing from Figure 5.1 showed that the direct extraction of q-grams generally has a better run time. Recursive hashing only presented better performance when the spaced seeds have no block or with 1 block with high weight. The reason could be repeated lookup of character to remove/add causing cache-miss, slowing the computation.

5.2. Determining threshold

In order to test the accuracy of the threshold estimation, measurements were firstly carried out on MMseqs2 with default seed. The test shows that MMseqs2 often

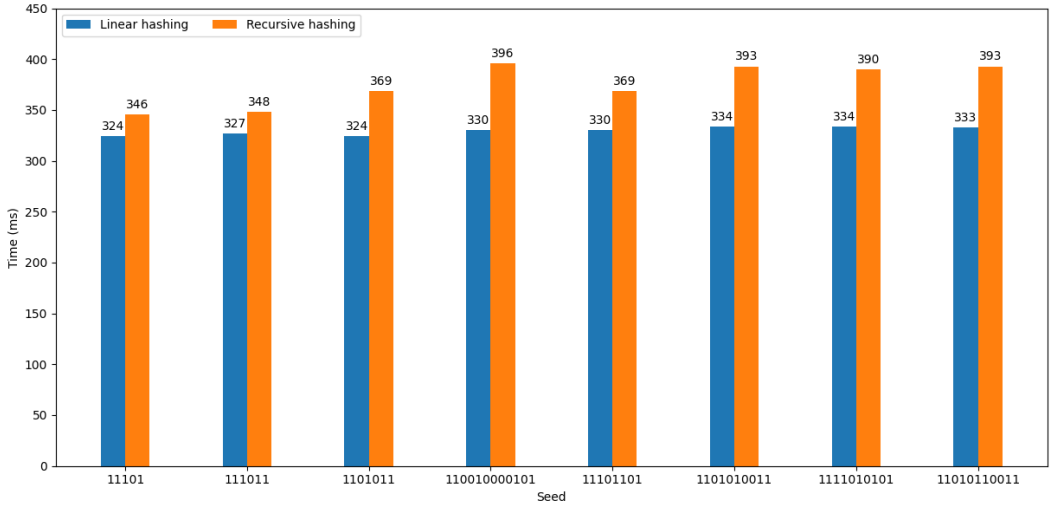

Figure 5.1.: Performance of two hashing functions

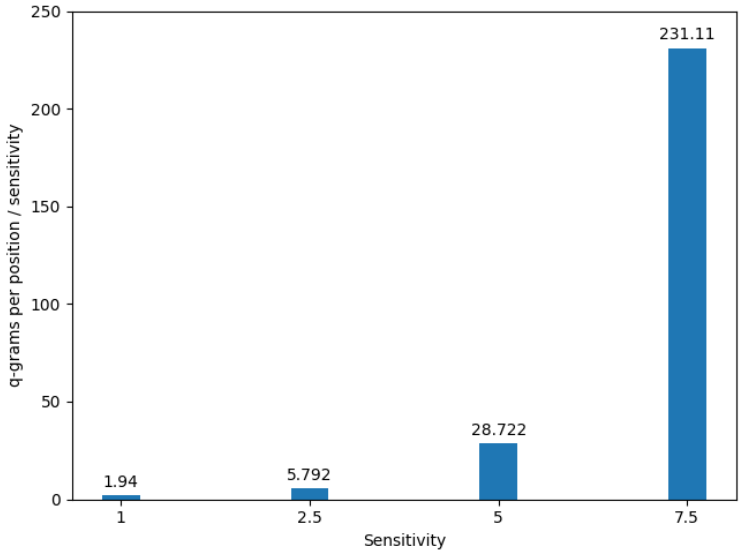

Figure 5.2.: Ratio of number of generated q-grams using MMseqs2 approach vs. expected

overgenerated q-grams, with the worst case creating 231.11 % more than expected. Both of the approaches, context-free and context-sensitive, were tested with the dividing scheme in context-sensitive approach reused from the q-gram generation algorithm. For each test, the total number of q-grams generated is averaged and then compared to the expected list length per position and MMseqs2 result. The benchmarks of context-free (Figure 5.3) and context-sensitive approach (Figure 5.4) both show better control compared to MMseqs2 and the resulted list lengths are

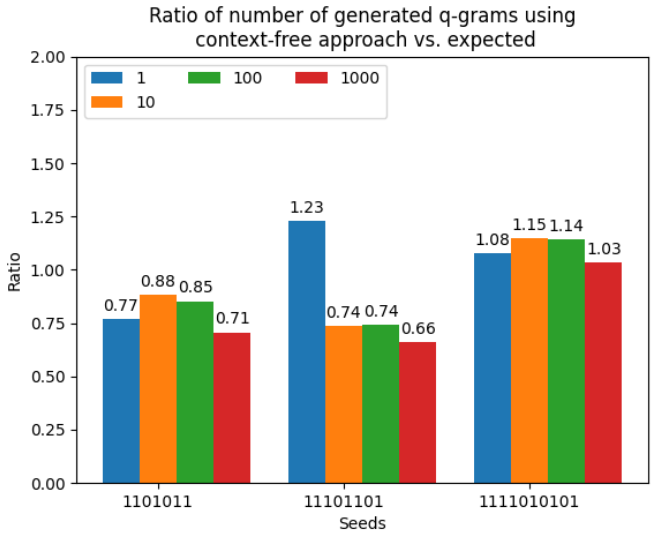

Figure 5.3.: Ratio of number of generated q-grams using context-free approach vs. expected

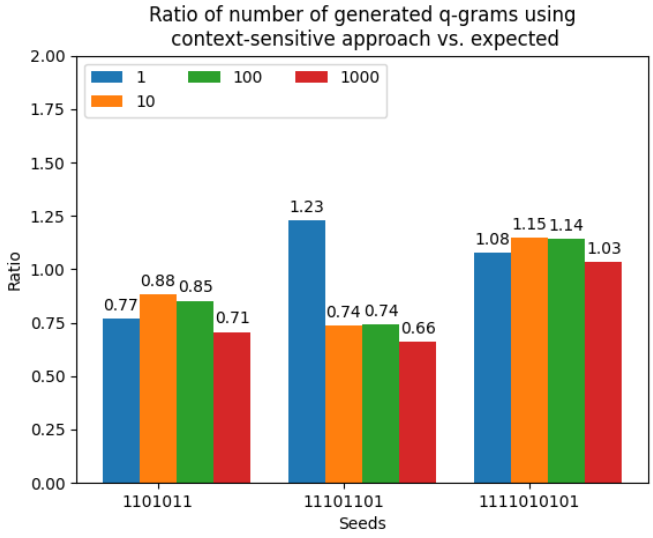

Figure 5.4.: Ratio of number of generated q-grams using context-free approach vs. expected

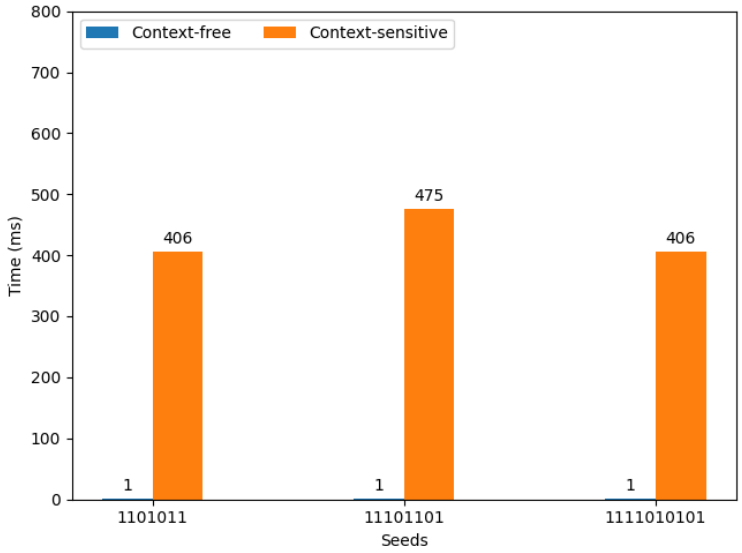

Figure 5.5.: Run time comparison of context-free vs. context-sensitive approach

relatively close to the expected value. The thresholds created by this approach show to be a good upper bounds for the k-environment in lower q, where only sensitivity of 1 in the seed weight 6 crosses the expected ammount. In higher seed weight, both approaches serve well as an approximation, where only a maximum of 15 % more q-grams were generated compared to expected value. In order to discern the usage of each approach,the run time of each method is investigated further in Figure 5.5:

AVT: #1: I'm looking to make a stacked bar chart with annotation on each bar as legend for each smaller step, e.g. for context-sensitive it would be extracting frequency of sub-q-gram and creating histogram.

The context-free approach shows to take very little time, almost instanteneous. Coupled with the very good estimation of the threshold it proves to be an universal on-the-fly solution. The context-sensitive approach takes a lot more time, average around 0.5 s to create a threshold (compensated for creation of unsorted-unsorted q-gram score matrices, see Section 6.3),since it incurs a $O(|\mathcal{A}|^{2q})$ time to build histogram. Therefore this approach should be used only on high sensitivity, where long computation time is expected, or cached in the preprocessing of target sequences. It is important to note that since the approximation of the threshold k doesn't take local score correction into account, all of the above measurements are without local score correction. In Figure 5.6 are some benchmarks with correction of varying degree, showcasing how the average list length is impacted. The result shows that the average q-gram list can vary strongly and unpredictably under usage of local composition bias score correction, where if the scale of the correction is from 0.75 - 1, the resulted

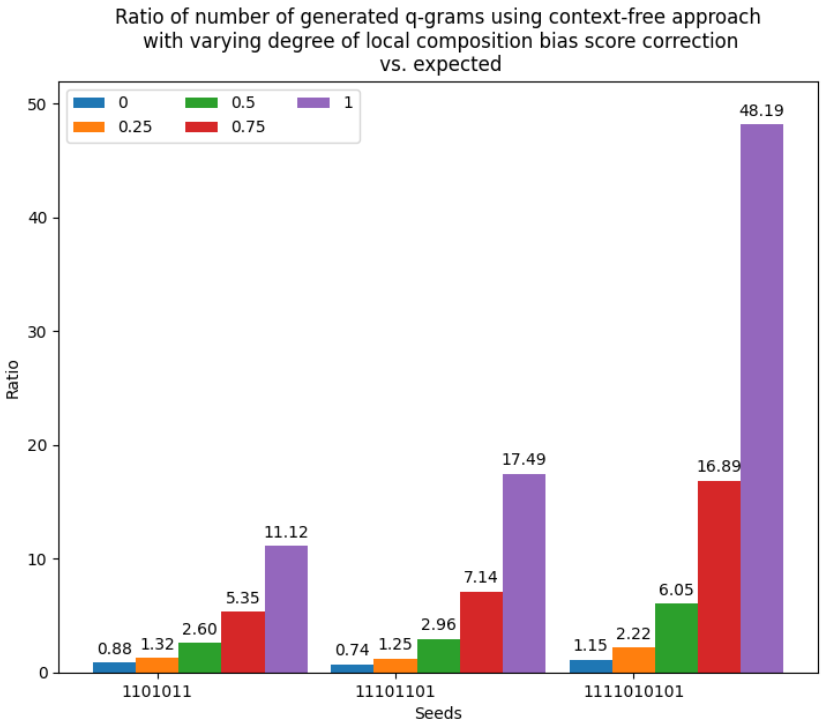

Figure 5.6.: Influence of local composition bias score correction on average q-gram list length. The original estimation on q-gram list length was 10.

number can be as few as 5 times or as many as 48 times more than expected. The proper scaling of the factor remains to be investigated further.

5.3. Generating q-grams

In order to test the efficiency of the q-gram list generation, the original workflow in MMseqs2 was replicated and tested against the new algorithm. This workflow, along with both variants of the new algorithm, with and without SIMD implementation, was measured in Figure 5.7, 5.8 and 5.11 for different seed weights. The tests showed that the new algorithm has generally a 10 % to 15 % performance loss compared to the MMseqs2 method. This is the overhead caused by the sorting and reshuffled of q-gram. The tradeoff is the low initial memory consumption, where only 36 MB of memory is consumed for the score matrices instead of 200 MB. The SIMD function unfortunately shows to have an negative impact on performance, which could be due to the SIMD instruction not optimized for the large padded section. On average,

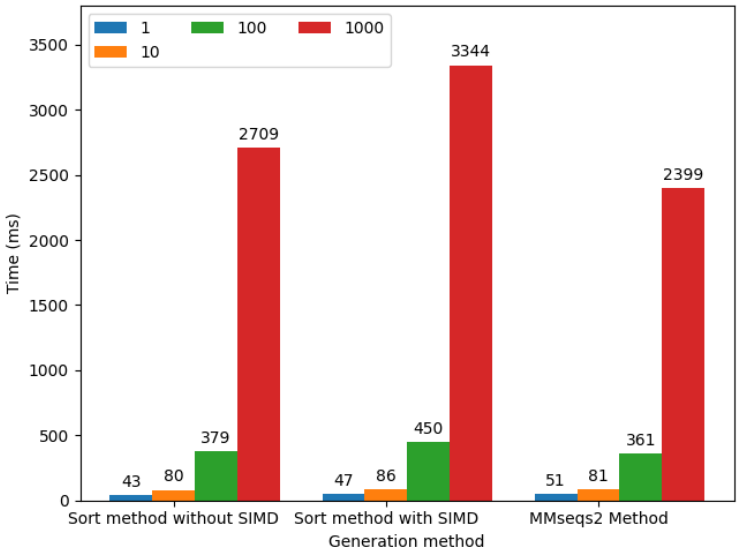

Figure 5.7.: Time measured in miliseconds to generate q-grams with different sensitivities using seed 1 101 011

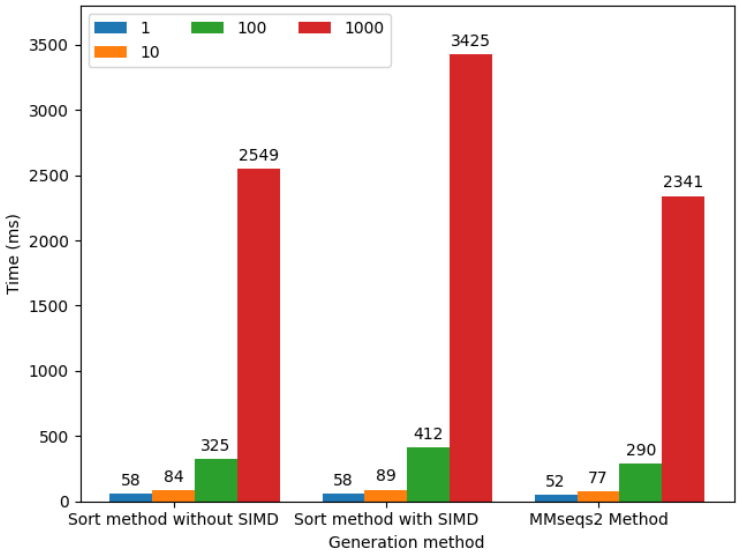

Figure 5.8.: Time measured in miliseconds to generate q-grams with different sensitivities using seed 11 101 101

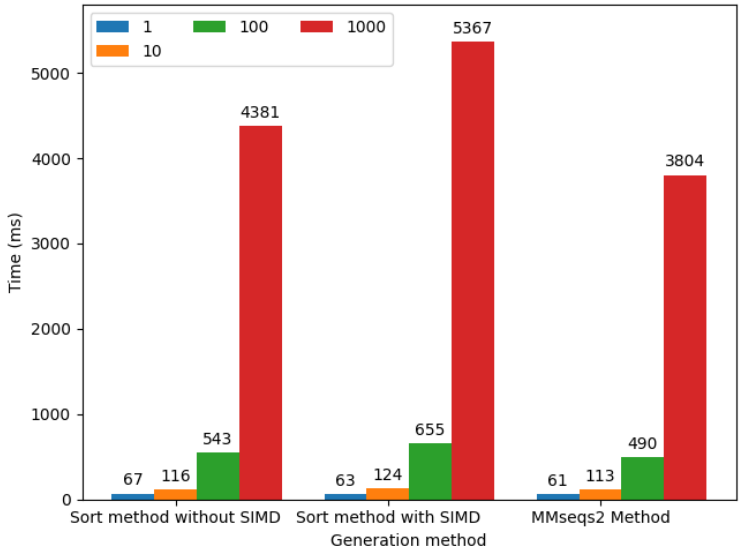

Figure 5.9.: Time measured in miliseconds to generate q-grams with different sensitivities using seed 1 111 010 101

the program spends 18.2 ns to generate a 5-gram, 18.3 ns to generate a 6-gram and 20.1 ns to generate a 7-gram. This shows that the q-gram generation scales generally well with q, where an additional loop in creation of 7-gram causing an increase of 10 %. It could be reasoned that the generation of 8-gram and 9-gram may follow the same trend, which coupled with the new approach in estimating threshold could lead to an expansion of seed weight (see Section 6.2).

5.4. General Performance

In this section, the overall run time of the whole program is profiled to find hotspots and to figure the scalability of the system. The program will be run with the most optimized settings, meaning context-free threshold approximation and q-gram generation using sorted method. In Figure, the recorded time in each step of the program is visualized.

AVT: #2: Same as last comment

The test showed that the run time of processing target data is generally negligible. The processing of match data could be more significant, reaching 0.4 s in testing but still scales much better than q-gram generation phase. This means that the hotspot of the program lies in the q-gram generation phase, specifically in the Cartesian product

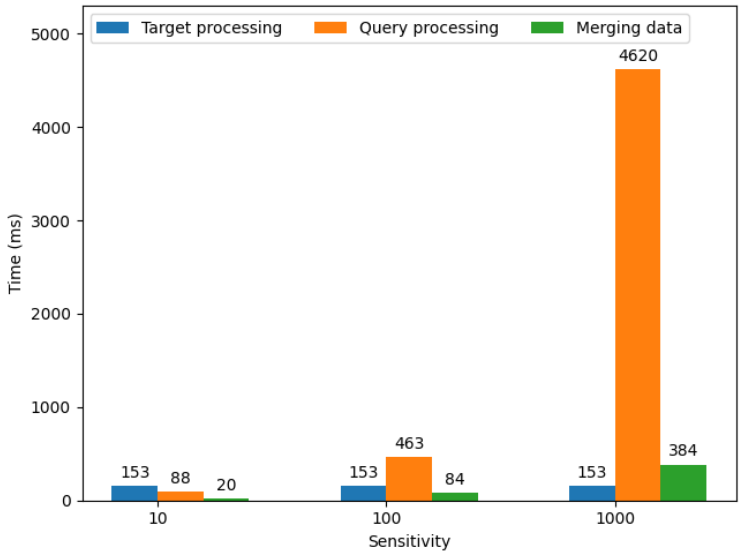

Figure 5.10.: Run time of different phases in the program, measured in miliseconds, using seed 11 101 101, sensitivity 10, 100 and 1 000

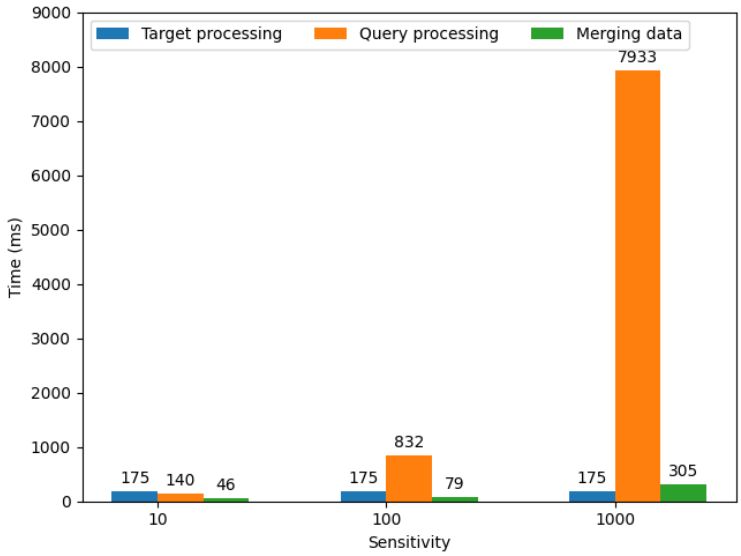

Figure 5.11.: Run time of different phases in the program, measured in miliseconds, using seed 1 111 010 101, sensitivity 10, 100 and 1 000

formation and in lesser manner, the radix sort of q-gram data. In the most heavy measurement, the q-gram generation phase still takes twice as long as the sorting after.

Chapter 6.

6 Discussion & Future improvements

6.1. Features

The new workflow was created with expandability in mind, therefore a lot of the calculation allows for greater seed size and even greater sub-q-gram size. A few of MMseqs2 features were also recreated, namely the ability to scale local composition bias correction and sensitivity manually. The design of the program also prioritizes flexibility, in that every calculation option, ranging from MMseqs2 mode vs new sorted q-gram method mode and both context-free and context-sensitive approach can be changed on-the-fly without recompilation. The program was intended to be a testing environment only, therefore custom workflows in MMseqs2 were not recreated, namely the stepwise sensitive search.

6.2. Possibility of expanding seed

Earlier it was shown that the run time of q-gram generation scales almost constantly to seed, therefore some testing was created to test for seeds of weight 8 and 9:

AVT: #3: Waiting for BytesUnit fix

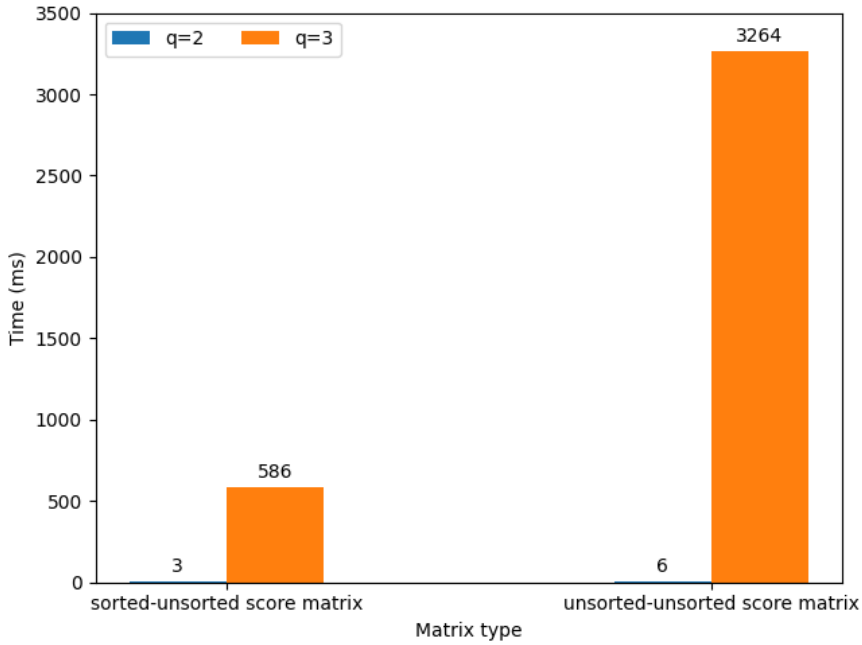

Figure 6.1.: Time measured in miliseconds to generate $q \times q$ score matrix

6.3. Compile-time calculation of score matrices

Even though the class diagram in Figure 4.1 shows that it's possible to compute the score matrix in compile time, since it only depends on the two index tables of sorted/unsorted q-grams, the implementation decides to create the table in run time instead. The reason largely is due to the high memory stack size requirements and lack of compile time sorting algorithm. A manual compile time sorting implementation (i.e. quicksort) would again demand more compile time instances. A separate benchmark in Figure 6.1 showed that this could incur a 0.6 s loss in creating the sorted-unsorted 3-gram score matrix or 3.3 s in creating the unsorted-unsorted 3-gram score matrix at the start of the computation.

6.4. Performance

Even though the new workflow shows general improvements on performance, the implementation still falls short compared to original MMseqs2 program, especially in large q-gram list length.This shows that there is still possible future optimization, particularly in loops. Another important further improvent would be the ability to process the data in multithreads.

7 Chapter 7. Conclusion

In this paper, the prefiltering module of the MMseqs2 software suite was examined and alternative optimization methods were tried and tested. A possible improvement in processing target sequences through recursive hashing, although has a better time complexity in theory, was shown to not be a major improvement, other than in edge cases of low block numbers. A new method of estimating threshold for better estimation of number of generated q-grams was implemented and showed to be a better estimation approach, both in low and high sensitivity. Finally, an optimized branch-and-bound algorithm was introduced and presented both a better run time and lower memory consumption, allowing future expansion into larger seed size.

AVT: #4: Missing references. Will be added over the weekend.

Bibliography

[1] Huey Duck, Dewey Duck, and Louie Duck. Getting more badges. *The Junior Woodchuck Journal*, 2:53–86, 2004.

Appendix A.

A Some additional method descriptions

Dataset	Number of sequences	Maximum Sequence Length	Minimum Sequence Length
QUERY_DIFF	426	4 291	8
TARGET	20 000	8 081	7

Table A.1.: Datasets used in testing the modules

Nr	Seed	Span	Weight	Number of Blocks
1	11 101	5	4	2
2	111 011	6	5	2
3	1 101 011	7	5	3
4	110 010 000 101	12	5	4
5	11 101 101	8	6	3
6	1 101 010 011	10	6	4
7	1 111 010 101	10	7	5
8	11 010 110 011	11	7	4

Table A.2.: Seeds used in testing

Appendix B.

B Some Additional Data

Morbi luctus, wisi viverra faucibus pretium, nibh est placerat odio, nec commodo wisi enim eget quam. Quisque libero justo, consectetuer a, feugiat vitae, porttitor eu, libero. Suspendisse sed mauris vitae elit sollicitudin malesuada. Maecenas ultricies eros sit amet ante. Ut venenatis velit. Maecenas sed mi eget dui varius euismod. Phasellus aliquet volutpat odio. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Pellentesque sit amet pede ac sem eleifend consectetuer. Nullam elementum, urna vel imperdiet sodales, elit ipsum pharetra ligula, ac pretium ante justo a nulla. Curabitur tristique arcu eu metus. Vestibulum lectus. Proin mauris. Proin eu nunc eu urna hendrerit faucibus. Aliquam auctor, pede consequat laoreet varius, eros tellus scelerisque quam, pellentesque hendrerit ipsum dolor sed augue. Nulla nec lacus.

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich vorliegende Bachelorarbeit im Studiengang Computing in Science selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Hamburg, den 24. November 2023

Anh Viet Ta

Ich bin mit einer Einstellung der Bachelorarbeit in den Bestand der Bibliothek des Departments Informatik einverstanden.

Hamburg, den 24. November 2023

Anh Viet Ta