



# Optimal neighborhood indexing for protein similarity search

Pierre Peterlongo, Laurent Noé, Dominique Lavenier, van Hoa Nguyen,  
Gregory Kucherov, Mathieu Giraud

► **To cite this version:**

Pierre Peterlongo, Laurent Noé, Dominique Lavenier, van Hoa Nguyen, Gregory Kucherov, et al.. Optimal neighborhood indexing for protein similarity search. BMC Bioinformatics, BioMed Central, 2008, 9 (534), 10.1186/1471-2105-9-534 . inria-00340510

**HAL Id: inria-00340510**

**<https://hal.inria.fr/inria-00340510>**

Submitted on 21 Nov 2008

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Optimal neighborhood indexing for protein similarity search

Pierre Peterlongo<sup>\*1</sup>, Laurent Noé<sup>2</sup>, Dominique Lavenier<sup>1</sup>, Van Hoa Nguyen<sup>1</sup>, Gregory Kucherov<sup>2</sup>, Mathieu Giraud<sup>2</sup>

<sup>1</sup> IRISA, INRIA, CNRS, Université Rennes 1

<sup>2</sup> LIFL, INRIA, CNRS, Université Lille 1

Email: Pierre Peterlongo\* - pierre.peterlongo@irisa.fr; Laurent Noé - laurent.noe@lifl.fr; Dominique Lavenier - dominique.lavenier@irisa.fr; Van Hoa Nguyen - vhnghuyen@irisa.fr; Gregory Kucherov - gregory.kucherov@lifl.fr; Mathieu Giraud - mathieu.giraud@lifl.fr;

\*Corresponding author

## Abstract

---

**Background:** Similarity inference, one of the main bioinformatics tasks, has to face an exponential growth of the biological data. A classical approach used to cope with this data flow involves heuristics with large seed indexes. In order to speed up this technique, the index can be enhanced by storing additional information to limit the number of random memory accesses. However, this improvement leads to a larger index that may become a bottleneck. In the case of protein similarity search, we propose to decrease the index size by reducing the amino acid alphabet.

**Results:** The paper presents two main contributions. First, we show that an optimal neighborhood indexing combining an alphabet reduction and a longer neighborhood leads to a reduction of 35% of memory involved into the process, without sacrificing the quality of results nor the computational time. Second, our approach led us to develop a new kind of substitution score matrices and their associated *e*-value parameters. In contrast to usual matrices, these matrices are rectangular since they compare amino acid groups from different alphabets. We

describe the method used for computing those matrices and we provide some typical examples that can be used in such comparisons. Supplementary data can be found on the website <http://bioinfo.lifl.fr/reblosum>.

**Conclusions:** We propose a practical index size reduction of the neighborhood data, that does not negatively affect the performance of large-scale search in protein sequences. Such an index can be used in any study involving large protein data. Moreover, rectangular substitution score matrices and their associated statistical parameters can have applications in any study involving an alphabet reduction.

---

## Background

One fundamental task in bioinformatics concerns large scale comparisons between proteins or families of proteins. It often constitutes the first step before further investigations. A typical comparison, for example, is to query a database with a newly discovered sequence. Observed similarities witness a putative common biological function and direct further studies.

In this paper, we focus on massive protein sequence comparisons: a large database is iteratively compared with relatively short queries (such as newly sequenced data). A possible approach is to use the exact dynamic programming method [1]. For a given similarity model, this method provides optimal alignments within a quadratic computation time. Some optimizations achieve a sub-quadratic complexity [2], but the computation time remains prohibitive for large scale comparisons. Thus, in practice, the full dynamic programming approach is applied to comparison of short sequences.

A successful family of similarity search methods is provided by *seed-based heuristics*, starting with **Fasta** [3] and **Blast** [4] and including specific methods for protein similarities such as **Blastp** [5]. Seed-based heuristics were recently enhanced by advanced seeding tools like the *spaced seeds* used in **PatternHunter** [6] or **Yass** [7] (see [8] for a recent survey). Authors of this paper also worked on the alliance between advanced seeds techniques and

reconfigurable architectures [9].

The main idea of seed-based heuristics is to anchor the detection of similarities using matching short words or short subsequences occurring in both compared sequences. The form of these words or subsequences is provided by a pattern called a *seed*. A word that respects the seed is called a *key*. For instance, *MVK* is one of  $20^3$  possible keys for the seed of three consecutive characters on the protein alphabet. Detection of similarities between two strings is done in three stages, as presented in Figure 1:

- **Stage 1:** search for keys that occur in both strings,
- **Stage 2:** extension of these matching keys with an ungapped alignment, keeping only the alignments with a score greater than a given threshold  $\mathcal{T}$ ,
- **Stage 3:** full dynamic programming algorithm, applied only to successfully extended matching keys.

In this work, we consider comparisons between a set of protein queries against a large protein database of  $N$  amino acids. A common usage of **Blast** is to index the queries, and then to scan the full database at the runtime. If the size of the query and the database allow it, a full indexation of both leads to advantageous results [10]. In our work, we applied approach used e.g. in **Blat** [11] where the database is indexed once and each query is successively processed.

To be efficient, the database positions are indexed by seed keys. The usual indexing scheme is shown Figure 2: for each key, a list of all its occurrences is stored. At **Stage 1**, each query position corresponds to a seed key (or, for the **Blastp** approach, a set of seed keys that are similar to the query seed key). An index access provides the list of key occurrences in the database, enabling **Stage 2**. We call such an approach the **offset indexing** approach. In this case, for each seed position, an offset of  $\lceil \log_2 N \rceil$  bits is stored. The index size is thus equal to  $S_{\text{offset}} = N \times \lceil \log_2 N \rceil$  bits.

For each query position, each execution of **Stage 2** needs to access all the occurrences of the corresponding key. This leads to numerous random memory accesses that are time

consuming: memory accesses at random positions are not efficiently cached and require high latencies [12]. A way to reduce the computation time is thus to avoid as far as possible such random memory accesses. For that purpose, it is possible to additionally store, for each key occurrence, its left and right neighborhoods in the sequence, as illustrated in Figure 3. Thus, given a position in the query and its corresponding key, all neighborhoods of this key occurrences in the database are obtained through a single random memory access. For each database position, two neighborhoods are additionally stored. We call this indexing approach the **neighborhood indexing** approach. The overall index size is then equal to  $S_{\text{neighborhood}} = N \times (\lceil \log_2 N \rceil + 2\alpha L)$  bits, where

- $\alpha$  is the number of bits for coding a character (amino acid), and
- $L$  is the length of each neighborhood.

As seen in Figure 4, the main advantage of the neighborhood indexing is that it speeds up the execution time by a factor ranging between 1.5 and 2 over the offset indexing. The actual speed gain depends on the database length and on many implementation and architecture parameters (such as memory and cache sizes, cache strategies and access times) that will not be discussed here. An obvious drawback of the neighborhood indexing is the additional memory it requires to store neighborhoods. Comparing the two indexing schemes, the ratio  $r$  between the overall index sizes of the neighborhood indexing and the offset indexing is

$$r = \frac{S_{\text{neighborhood}}}{S_{\text{offset}}} = 1 + \frac{2\alpha L}{\lceil \log_2 N \rceil}.$$

In common experiments,  $\lceil \log_2 N \rceil$  is between 20 and 40,  $\alpha L$  is between 20 and 200, hence  $r$  is between 2 and 21. It is worth mentioning that the  $\lceil \log_2 N \rceil$  value is often raised to a more practical 32 or 64 bits, reducing the ratio  $r$  even more. Storing neighborhoods becomes then relevant with the reduction of memory prices. For instance, the modern technology brings the possibility to get gigabytes of Flash memory in a personal computer for some hundred dollars. It is thus interesting to exploit this storage space as much as possible. It can be used for treating larger databases, but also, as in this work, for speeding

up widely used applications.

However, the index size still remains the main limitation. In this paper, we study how the size of a large neighborhood index can be reduced while preserving the result quality. For this purpose, we worked on reducing as much as possible the ratio  $r$ . A way for doing this is to reduce the factor  $\alpha L$ . We propose to simultaneously increase the neighborhood length ( $L$ ) and reduce the alphabet size ( $2^\alpha$ ). We limit the alphabet size by partitioning amino acids into groups. This reduces  $\alpha$  by encoding neighborhood characters in less than 5 bits required for coding 20 amino acids. Partitioning the amino acids into 16 groups enables to encode each group using 4 bits, and partitioning into 8, 4 or 2 groups enables to encode each group by 3, 2, and 1 bits respectively. All these reduced alphabets are tested in this paper.

Grouping amino acids was studied in several papers [13–16]. Groups can rely on amino acid physical-chemical properties or on a statistical analysis of alignments. For example, the authors of [13] computed correlation coefficients between pairs of amino acids based on the BLOSUM50 matrix and used a greedy algorithm to merge them. A branch-and-bound algorithm for partitioning the amino acids was proposed in [14]. Those papers mainly deal with the construction of reduced alphabets, but none of them studies how the alphabet reduction affects the sensitivity of similarity search, or undertakes a quantitative analysis of the trade-off between search sensitivity and index size for those alphabets. This raises the following problem that is solved in this paper: *Can reduced alphabets allow one to decrease the factor  $\alpha L$  while preserving the quality of similarity search results?*

## Results and Discussion

The main result of our work is an effective reduction of the index size without deteriorating the quality of the results of similarity search. Moreover, we provide substitution score matrices and  $e$ -value parameters to be used with reduced alphabets. Our results are based on the alphabets defined by the amino acids groups proposed by Li and al. (Table 3 of [15]), but our method can be applied to any other amino acids partitions. The website

<http://bioinfo.lifl.fr/reblosum> provides data for all the alphabets reported in [16].

In the rest of the paper, the original alphabet of 20 amino acids is denoted by  $\Sigma_{20}$ , where each character is encoded by 5 bits. Reduced alphabets  $\Sigma_{16}$ ,  $\Sigma_8$ ,  $\Sigma_4$  and  $\Sigma_2$ , respectively of size 16, 8, 4 and 2, have each character encoded by 4, 3, 2 and 1 bits respectively. We used the following alphabets, defined in [15]:

$$\begin{aligned}\Sigma_{16} &= \{[C], [FY], [W], [ML], [IV], [G], [P], [A], [T], [S], [N], [H], [QE], [D], [R], [K]\}, \\ \Sigma_8 &= \{[CFYW], [MLIV], [G], [P], [ATS], [NH], [QED], [RK]\}, \\ \Sigma_4 &= \{[CFYW], [MLIV], [GPATS], [NHQEDRK]\}, \text{ and} \\ \Sigma_2 &= \{[CFYWMLIV], [GPATSNHQEDRK]\}.\end{aligned}$$

The main idea is to represent the neighborhoods of keys stored in the index (see Figure 3) over a reduced alphabet. Consequently, at **Stage 2** of the similarity search, amino acid sequences are compared with sequences over the reduced alphabet. By an *alignment over*  $\Sigma \times \Sigma'$ , we understand an alignment between a sequence over  $\Sigma$  and a sequence over  $\Sigma'$ . Thus, in this paper we will consider alignments over  $\Sigma_{20} \times \Sigma_{20}$ ,  $\Sigma_{20} \times \Sigma_{16}$ ,  $\Sigma_{20} \times \Sigma_8$ ,  $\Sigma_{20} \times \Sigma_4$  and  $\Sigma_{20} \times \Sigma_2$ .

In the next sections, we describe how to evaluate the quality of **Stage 2** and how a substantial index size reduction can be obtained by using longer neighborhoods on reduced alphabets. As presented in Figure 5, using a reduced alphabet involves several parameters that we study in the following sections. In section *Rectangular substitution score matrices*, we present substitution score matrices used for alignments over  $\Sigma_{20} \times \Sigma_8$  and  $\Sigma_{20} \times \Sigma_{16}$ . We then present the computation of *e-value* to estimate the significance of alignments over reduced alphabets. The last section, *Experimental validation*, describes a practical application of reduced alphabets to real biological data.

## Stage 2 algorithm and quality

A detailed description of **Stage 2** is given in Algorithm 1. Query and database neighborhoods of a matching key (detected during **Stage 1**) are compared character by

character over  $L$  positions. During this comparison that uses substitution score matrices (lines 5 and 9), the highest scores for the left and right neighborhoods are kept (lines 6 and 10). If the sum of the highest scores exceeds a threshold  $\mathcal{T}$ , the alignment is kept for **Stage 3** (line 12), otherwise it is rejected (line 13). Note that in the offset indexing case, a random memory access is performed in order to retrieve neighborhoods  $\mathbf{left}_{db}$  and  $\mathbf{right}_{db}$  (line 1). This is not the case for the neighborhood indexing, as the neighborhoods are stored directly in the index.

---

**Algorithm 1 Stage 2**

---

**Ensure:** reports if a matching key occurrence potentially belongs to an alignment

**Require:** query neighborhoods ( $\mathbf{left}_{query}$  and  $\mathbf{right}_{query}$ )

```

1: get database neighborhoods  $\mathbf{left}_{db}$  and  $\mathbf{right}_{db}$ 
2:  $result_{left} \leftarrow 0$ ;  $highest_{left} \leftarrow 0$ 
3:  $result_{right} \leftarrow 0$ ;  $highest_{right} \leftarrow 0$ 
4: for  $i$  from 1 to  $L$  do
5:    $result_{left} \leftarrow result_{left} + subst\_score(\mathbf{left}_{db}[i], \mathbf{left}_{query}[i])$ 
6:   if  $result_{left} > highest_{left}$  then  $highest_{left} \leftarrow result_{left}$  endif
7: end for
8: for  $i$  from 1 to  $L$  do
9:    $result_{right} \leftarrow result_{right} + subst\_score(\mathbf{right}_{db}[i], \mathbf{right}_{query}[i])$ 
10:  if  $result_{right} > highest_{right}$  then  $highest_{right} \leftarrow result_{right}$  endif
11: end for
12: if  $highest_{left} + highest_{right} \geq \text{threshold } \mathcal{T}$  then return true endif
13: return false

```

---

The quality of **Stage 2** is measured by a trade-off between its sensitivity (ability to extend true alignments) and selectivity (ability to filter out spurious seed hits). Computation of those values is described page 11.

The performance of **Stage 2** gets better for higher sensitivity and lower selectivity. Increasing the threshold  $\mathcal{T}$  or decreasing the neighborhood length  $L$  makes **Stage 2** more selective but less sensitive (faster execution at the price of worse quality results) while decreasing  $\mathcal{T}$  or increasing  $L$  increases the sensitivity and decreases the selectivity (better quality results at the price of a slower execution).



### Reducing the index size by 35% without loss of quality

As shown in Figure 6, the sensitivity/selectivity trade-off follows a convex curve. We propose here to achieve an equivalent trade-off with a reduction of the index size.

Clearly, for a fixed neighborhood length  $L$  (in Figure 6, 16 amino acids), the sensitivity/selectivity trade-off is always better when using the full amino acid alphabets than a reduced alphabet. This is easily explained by the fact that reducing the alphabet size decreases the alignment accuracy. In order to keep up with the sensitivity/selectivity ratio, the neighborhood length  $L$  should be increased. In Figure 7, all reduced alphabets, used with increased neighborhood lengths, now perform equivalently (or slightly better) than the full alphabet.

Figure 8 shows the dependency, for different reduced alphabets, between the number of bits needed to store both neighborhoods (X axis) and the selectivity (Y axis), for an equivalent quality (fixed sensitivity). Those results are obtained with the use of special substitution score matrices, adapted to reduced alphabets, that are presented in the next section. Our main result is that for any given selectivity, using any of the reduced alphabets for storing neighborhoods leads to a smaller  $\alpha L$  factor than for the  $\Sigma_{20}$  alphabet. Therefore, for a fixed memory usage, the sensitivity/selectivity trade-off is always better with a reduced alphabet than with the full  $\Sigma_{20}$  alphabet.

In practice, this result enables a reduction of the index size without any sacrifice in running time or in result quality. Table 1 shows the memory requirements for different alphabets. We obtain a practical reduction of 42% of the factor  $\alpha L$  using the reduced alphabet  $\Sigma_2$  instead of  $\Sigma_{20}$ . The ratio  $r$  on the overall index size is then reduced by 35%.

### Rectangular substitution score matrices

We designed a method for computing substitution score matrices for any pair of possibly reduced amino acid alphabets. As this method is based on the original programs of [17], we call such matrices REBLOSUM for Rectangular BLOSUM matrices. The REBLOSUM matrices for alphabets  $\Sigma_{20} \times \Sigma_{20}$  are the original BLOSUM matrices. Tables 2 and 3 present REBLOSUM matrices for alignments over alphabets  $\Sigma_{20} \times \Sigma_{16}$  and  $\Sigma_{20} \times \Sigma_8$  respectively.

Such matrices can be applied in any method reducing the amino acid alphabets by residue grouping. As one may be interested in using any other pair of alphabets, we additionally propose a web interface available at <http://bioinfo.lifl.fr/reblosum>. The web interface computes REBLOSUM matrices for other amino acids alphabets, in particular for  $\Sigma_{20} \times \Sigma_4$  and  $\Sigma_{20} \times \Sigma_2$  that are not presented in Tables 2 and 3, as well as for other alphabets listed in [16] and for custom alphabets provided by the user.

### Parameters for *e*-value computation

The *e*-value, or *expected value*, provides the expected number of alignments with a given score, when comparing a text  $T$  and a query  $Q$  of length  $|T|$  and  $|Q|$  respectively. Local alignment methods like **Blast** sort results by increasing *e*-value, thus reflecting their decreasing significance. In the **Blast** algorithm, the *e*-value of an alignment is obtained by

$$e\text{-value} = K \cdot |Q| \cdot |T| \cdot e^{-\lambda s},$$

where  $s$  is the score of the alignment obtained with substitution matrices. Parameters  $\lambda$  and  $K$  are two constants that fit the Gumbel law, computed using methods described in [18]. Table 4 provides those parameters for several REBLOSUM substitution matrices.

### Experimental validation

In a model where the **Stage 2** alignments are ungapped, using reduced alphabets and alignments on longer neighborhoods can affect the result quality. Indeed, the longer the neighborhoods are, the bigger the chance to meet a gap in the sequences. More generally, the probabilities distributions used in theoretical sensitivity and specificity computations do not truly reflect the nature of the biological sequences. We thus validated our approach with large-scale tests on biological sequences. We set a database to be the hard-masked human chromosome 21 (UCSC Release hg18) translated according to the six possible reading frames. The query set was a set of seven archea and bacteria proteomes derived from a study of mitochondrial diseases. This set was selected for its interest toward the detection of potential insertions of mitochondrial genes in the human genome. Moreover, testing out our approach comparing such distant species represents one of the hardest

application case. Indeed more typical homology searches on closer sequences is easier. Tests on such homology searches could have hidden potential issue on our approach.

The database contained 12 700 507 amino acids whereas the query was composed by 5 321 439 amino acids. Using the `ssearch` method [19], 650 alignments were obtained between the database and the query (maximal  $e$ -value:  $10^{-3}$ ). This set of exhaustive *optimum* alignments was sufficient to validate our method in comparison with results obtained using different alphabets. The seed used in **Stage 1** was a subset seed (see [20]), as in [9]. For the neighborhood indexing, we indexed the database using each of the alphabets  $\Sigma_{20}$ ,  $\Sigma_{16}$ ,  $\Sigma_8$ ,  $\Sigma_4$  and  $\Sigma_2$ . We selected the neighborhood length to have a theoretical sensitivity close to 0.95 and a theoretical selectivity close to 0.01. Theoretical sensitivity and selectivity are defined according distributions presented on page 11.

This leads to indexing  $2 \times 11$  characters for  $\Sigma_{20}$ ,  $2 \times 12$  characters for  $\Sigma_{16}$ ,  $2 \times 14$  characters on  $\Sigma_8$ ,  $2 \times 19$  characters for  $\Sigma_4$ , and  $2 \times 32$  characters for  $\Sigma_2$  (Figure 7). The database index sizes are reported in Table 6. Using alphabet  $\Sigma_2$  instead of  $\Sigma_{20}$  reduces the overall index size: the ratio  $r$  goes from  $r_{20} = 5.58$  to only  $r_2 = 3.67$ , that is a 35% reduction. The initial assumption of ungapped alignments in the **Stage 2** can be wrong with a neighborhood length of  $2 \times 32$ . Thus one could prefer to use the alphabet  $\Sigma_4$  with  $2 \times 19$  characters, giving a 25% reduction of the overall index size ( $r_4 = 4.17$ ).

As shown in Table 5, each of the reduced alphabets yields a practical full sensitivity, as all the 650 alignments are found in each test. Moreover, the practical selectivity, close to  $10^{-3}$ , is here better than the theoretical one (0.01).

## Conclusions

We proposed a method for reducing the index size when storing neighborhoods of seed keys in protein databases. This approach is based on reducing the alphabet of indexed data while using a longer neighborhood. We save 35% of the index size without any modification on the result quality assuming an ungapped alignment model. We provided

optimal lengths for selected alphabets.

Furthermore, the proposed method requires unusual substitutions score matrices that are called REBLOSUM, for rectangular BLOSUM matrices. These matrices provide substitution scores between letters from different alphabets. We extended the computation of traditional BLOSUM matrices in order to compute REBLOSUM matrices, and adapted the computation of  $\lambda$  and  $K$  parameters for  $e$ -value estimation to reduced alphabets. We provided REBLOSUM matrices and their corresponding  $\lambda$  and  $K$  values for selected alphabets. Other matrices and parameters can be obtained from the website <http://bioinfo.lifl.fr/reblosum>.

## Methods

In this section, we describe the methods we used to compute the sensitivity and selectivity of similarity search on reduced alphabets as well as the neighborhood length. We further describe the computation of REBLOSUM substitution score matrices and of the  $e$ -value parameter. Moreover, we explain how the threshold  $\mathcal{T}$  is computed at **Stage 2** depending on the  $e$ -value specified by the user. Finally, we describe how we estimated the time gain of the the neighborhood indexing over the offset indexing.

### Selectivity and sensitivity computation

The sensitivity of **Stage 2** is defined by the ratio of retained “true alignments” (a “true alignment” is an alignment known to be relevant, according to a model or to a reference set like the BLOCKS database) :

$$\text{sensitivity} = \frac{\# \text{ successfully extended true alignments}}{\# \text{ true alignments}},$$

The selectivity is defined as the ratio of retained “random alignments” (a “random alignment” means an alignment of randomly chosen amino acid pairs drawn according to an appropriate probability distribution) :

$$\text{selectivity} = \frac{\# \text{ successfully extended random alignments}}{\# \text{ random alignments}}.$$

Note that here we focus on the behavior of **Stage 2** and do not take into consideration the sensitivity/selectivity of **Stage 1**. In particular, in the above fractions we consider only alignments that extend a hit presumably reported at **Stage 1**.

The sensitivity and the selectivity of **Stage 2** rely on three parameters: the alphabet choice, the neighborhood length, and the score threshold  $\mathcal{T}$ . Given these three parameters, we applied a dynamic programming algorithm to compute the probability for the filter to retain an alignment drawn according to a given *amino acid pair distribution*. Applied to distributions of “true” and “random” alignments (foreground and background distributions, respectively), the algorithm gives a theoretical estimation of the sensitivity and the selectivity of the filter. The two distributions were the Bernoulli models (namely the expected and the observed probabilities, see below), obtained with the BLOSUM programs on the BLOCKS protein database when processing the BLOSUM-62 matrix.

In our Algorithm 1, two neighborhoods (left and right) are processed. We thus consider the sum of two maximal scores, reached in the left and right neighborhoods. The probability that this sum reaches a given threshold  $\mathcal{T}$  at least once is computed as follows. First, we compute the probability for each neighborhood independently to reach *any* given maximal score  $s$  ( $s \geq 0$ ) within the neighborhood length. Then, these two independent discrete distributions are combined to compute the  $\mathcal{T}$  threshold requirement.

For our experiments, we calibrated the neighborhoods lengths to have a sensibility close to 0.95 and a selectivity close to 0.01, and computed related thresholds values (available of the REBLOSUM website).

### Computing REBLOSUM matrices

There are several substitution score matrices for the regular  $\Sigma_{20} \times \Sigma_{20}$  alphabet, and the most common of them are matrices from the BLOSUM family [21] (BLOCKS SUBstitution Matrix). They are built from the BLOCKS database of ungapped multiple alignments [22]. For a given identity level  $X$  and two amino acids  $i$  and  $j$ , the BLOSUM $X$  score  $B_{i,j}$  are log-likelihoods of *amino acid pair frequencies*:

$$B_{i,j} = \log \left( \frac{q_{i,j}}{p_i \cdot p_j} \right),$$

where  $p_i \cdot p_j$  is the *expected* probability of aligning  $i$  against  $j$ , and  $q_{i,j}$  is the *observed* probability of the same event in a subset of alignments of the BLOCKS database that have

at least X percent of identity. (Note that the computation of  $q_{i,j}$  takes into account different contributions provided by alignments with different identity levels.)

In our case, sequences over different alphabets are compared and we then have to adapt the matrix computation to compute appropriate rectangular matrices. For this purpose, the original data file (BLOCKS database version 5) was downloaded and the original programs of [17]<sup>1</sup> were modified in order to take into account the reduced alphabet on “one side” of the matrix and compute new log-likelihood scores. Given two alphabets  $\Sigma$  and  $\Sigma'$ , we compute such matrices for several identity levels X, using the log-likelihood of *groups of amino acid pair frequencies*:

$$B_{I,J} = \log \left( \frac{q_{I,J}}{p_I \cdot p_J} \right),$$

where  $p_I \cdot p_J$  is the *expected* frequency of aligning any amino acids from group  $I \subseteq \Sigma$  against any other amino acid from group  $J \subseteq \Sigma'$ , and  $q_{I,J}$  is the *observed* frequency of the same event in a subset of alignments of the BLOCKS database that have at least X percent of identity. The recent paper [23] discovered flaws in the original BLOSUM implementation, but shows that a corrected program does not improve (and even in some cases decreases) the results quality. Therefore, we did not take the proposed modifications into account.

The website <http://bioinfo.lifl.fr/reblosum> proposes a selection of REBLOSUM matrices for several alphabets, as well as an interface to compute REBLOSUM matrices for any alphabet and identity level specified by the user.

### **Prototype for estimating the time gain of offset indexing over neighborhood indexing**

For comparing the execution time between offset indexing and neighborhood indexing, a C prototype was created. In the case of the offset indexing, the index stores positions of all seeds in an unique integer array. For each seed key, a pointer provides the first occurrence in this array. In the case of the neighborhood indexing, the index uses a (unique) structure array instead of an integer array. For each key occurrence, the structure contains the key

---

<sup>1</sup>downloaded from <http://sci.cnb.uam.es/Services/ftp/databases/blocks/unix/blosum/>

position together with two neighborhoods.

Tests reported in Figure 4 were run on a 2 GHz PC with an AMD Opteron processor. The database size was selected so that the index fits into the main memory (4 GB) but not into the L1/L2 cache (1 MB). In those tests, the neighborhood indexing performs almost twice as fast as the offset indexing.

## Authors contributions

All authors conceived the study. VHN created and tested the prototype estimating the time gain using a neighborhood index. LN and MG computed REBLOSUM matrices and optimal neighborhood lengths using different alphabets and created the web interface. PP performed tests on biological sequences and drafted the manuscript. DL and GK proposed the setup of the work and participated in its coordination. All authors read and approved the final manuscript.

## Acknowledgments

This work was supported by the INRIA ARC grant “Flash” (*Seed Optimization and Indexing of Genomic Databases*).

## References

1. Smith T, Waterman M: **Identification of common molecular subsequences**. *Journal of Molecular Biology* 1981, **147**(195–197).
2. Crochemore M, Landau G, Ziv-Ukelson M: **A Sub-quadratic Sequence Alignment Algorithm for Unrestricted Cost Matrices**. In *Symposium On Discrete Algorithms (SODA 02)* 2002:679–688.
3. Lipman D, Pearson W: **Improved tools for biological sequence comparison**. *Proc. Natl. Acad. Sci. USA* 1988, **85**:2444–2448.
4. Altschul S, Gish W, Miller W, Myers W, Lipman D: **Basic local alignment search tool**. *Journal of Molecular Biology* 1990, **215**(3):403–410.
5. Altschul S, Madden T, Schäffer A, Zhang J, Zhang Z, Miller W, Lipman D: **Gapped BLAST and PSI-BLAST: a new generation of protein database search programs**. *Nucleic Acids Research* 1997, **25**(17):3389–3402.

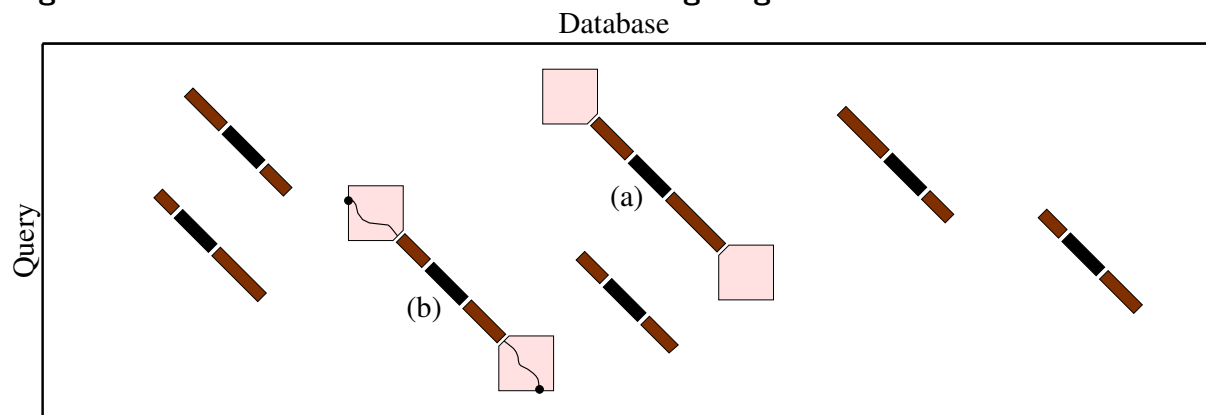
6. Li M, Ma B, Kisman D, Tromp J: **PatternHunter II: Highly Sensitive and Fast Homology Search**. *Journal of Bioinformatics and Computational Biology* 2004, **2**(3):417–439. [(early version in GIW 2003)].
7. Noé L, Kucherov G: **YASS: enhancing the sensitivity of DNA similarity search**. *Nucleic Acids Research* 2005, **33**:W540–W543.
8. Brown DG: *Bioinformatics Algorithms: Techniques and Applications*, Wiley-Interscience (I. Mandoiu, A. Zelikovsky) 2008 chap. A survey of seeding for sequence alignment, :126–152.
9. Peterlongo P, Noé L, Lavenier D, Georges G, Jacques J, Kucherov G, Giraud M: **Protein similarity search with subset seeds on a dedicated reconfigurable hardware**. In *Parallel Biocomputing Conference (PBC 07), Volume 4967 of Lecture Notes in Computer Science (LNCS)* 2007.
10. Nguyen VH, Lavenier D: **Speeding up Subset Seed Algorithm for Intensive Protein Sequence Comparison**. In *6th International Conference on research, innovation and vision for the future* 2008.
11. Kent WJ: **BLAT—the BLAST-like alignment tool**. *Genome Res* 2002, **12**(4):656–664, [<http://dx.doi.org/10.1101/gr.229202>. Article published online before March 2002].
12. Hennessy JL, Patterson DA: *Computer Architecture, A Quantitative Approach*. Morgan Kaufmann 2006.
13. Murphy L, Wallqvist A, Ronald L: **Simplified amino acid alphabets for protein fold recognition and implications for folding**. *Protein Engineering* 2000, **13**(3):149–152.
14. Cannata N, Toppo S, Romualdi C, Valle G: **Simplifying amino acid alphabets by means of a branch and algorithm and substitution matrices**. *Bioinformatic* 2002, **18**(8):1102–1108.
15. Li T, Fan K, Wang J, Wang W: **Reduction of protein sequence complexity by residue grouping**. *Protein Engineering* 2003, **16** (5):323–330.
16. Edgar R: **Local homology recognition and distance measures in linear time using compressed amino acid alphabets**. *Nucleic Acids Research* 2004, **32**:380–385.
17. Henikoff J, Henikoff S: **Amino Acid Substitution Matrices form Protein Blocks**. *Proc. Natl. Acad. Sci. USA* 1992, **89**:10915–10919.
18. Karlin S, Altschul SF: **Methods for assessing the statistical significance of molecular sequence features by using general scoring schemes**. *Proc Natl Acad Sci U S A* 1990, **87**(6):2264–2268.
19. Lipman D, Pearson W: **Rapid and Sensitive Protein Similarity Searches**. *Science* 1985, **227**:1435–1441.



20. Roytberg M, Gambin A, Noé L, Lasota S, Furletova E, Szczurek E, Kucherov G: **Efficient seeding techniques for protein similarity search**. In *Bioinformatics Research and Development, Proceedings of the 2nd International Conference BIRD 2008, Vienna (Austria), July 7-9, 2008, Volume 13 of Communications in Computer and Information Science*, Springer Verlag 2008:466–478.
21. Henikoff S, Henikoff J: **Amino acid substitution matrices from protein blocks**. *Proc. Natl. Acad. Sci. USA* 1992, **89**:10915–10919.
22. Henikoff S, Henikoff J: **Automated assembly of protein blocks for database searching**. *Nucleic Acids Res.* 1991, **19**(23):6565–6572.
23. Styczynski MP, Jensen KL, Rigoutsos I, Stephanopoulos G: **BLOSUM62 miscalculations improve search performance**. *Nat Biotech* 2008, **26**(3):274–275, [<http://dx.doi.org/10.1038/nbt0308-274>].

## Figures

**Figure 1 - Schematic view of a Blast-like 3-stage algorithm**



Representation of the three stages of comparison of a query (vertical) against a database (horizontal): **Stage 1**: identify seeds, i.e. small patterns occurring in both the query and the database (black diagonals). **Stage 2**: compute seed extensions and keep only those for which the score verifies at threshold  $\mathcal{T}$  (brown diagonals). On the Figure, seeds (a) and (b) are successfully extended. **Stage 3**: perform a full dynamic programming computation (white squares) on remaining seeds. In this example, only seed (b) leads to a significant alignment.

**Figure 2 - Offset indexing**

seed	database pos.
HRT	80743 1403483 1565421
HRV	12131
HRW	1987432 6465455 7654321

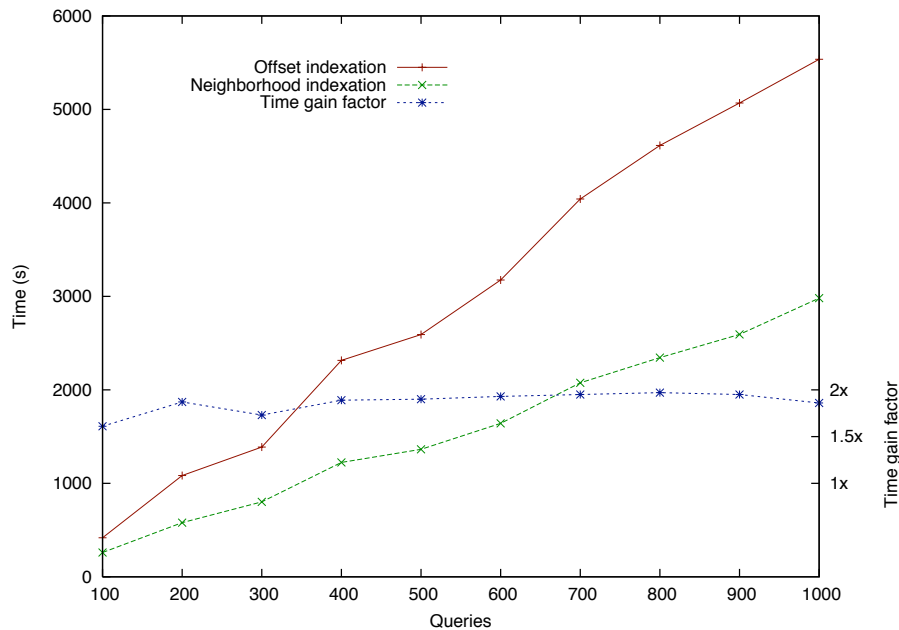
Fragment of an offset index. For each seed key (here composed of three letters), the list of its occurrence positions is stored.

**Figure 3 - Neighborhood indexing**

seed	database pos.	left neighbor	right neighbor
HRT	80743 1403483 1565421	WGN..IGPG DSG..HYRW ERS..LDLQ	QERN..ITM KHEL..ITI HWLD..IAK
HRV	12131	IKS..GASS	AKIE..KLE
HRW	1987432 6465455 7654321	VVK..FTGQ HYV..IGGD VTF..KDEV	AWLE..KIW RNPH..GMH KARE..QPL

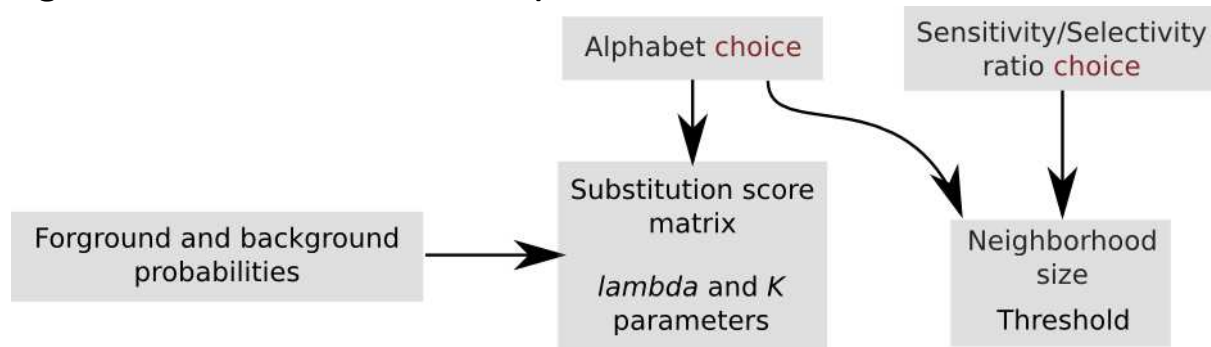
Fragment of a neighborhood index. For each seed key, the list of its occurrence positions is stored. For each occurrence, its right and left neighborhoods are additionally stored.

**Figure 4 - Time saved by neighborhood indexing compared to offset indexing**



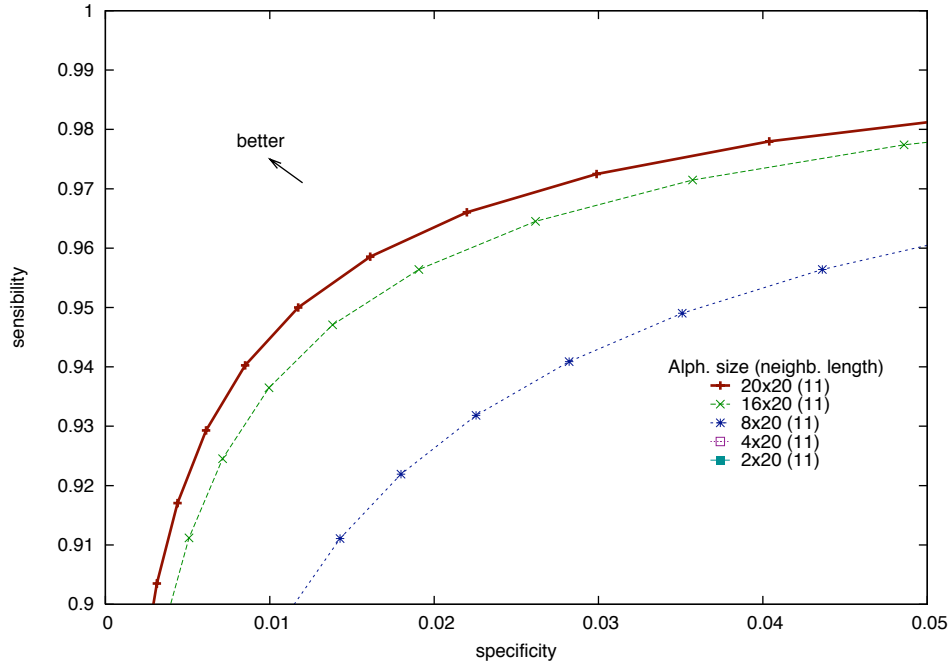
Execution time using the offset indexing and the neighborhood indexing for comparing a growing set of queries against a bank of  $70 \cdot 10^3$  proteins.

**Figure 5 - Parameters involved in alphabet reduction**



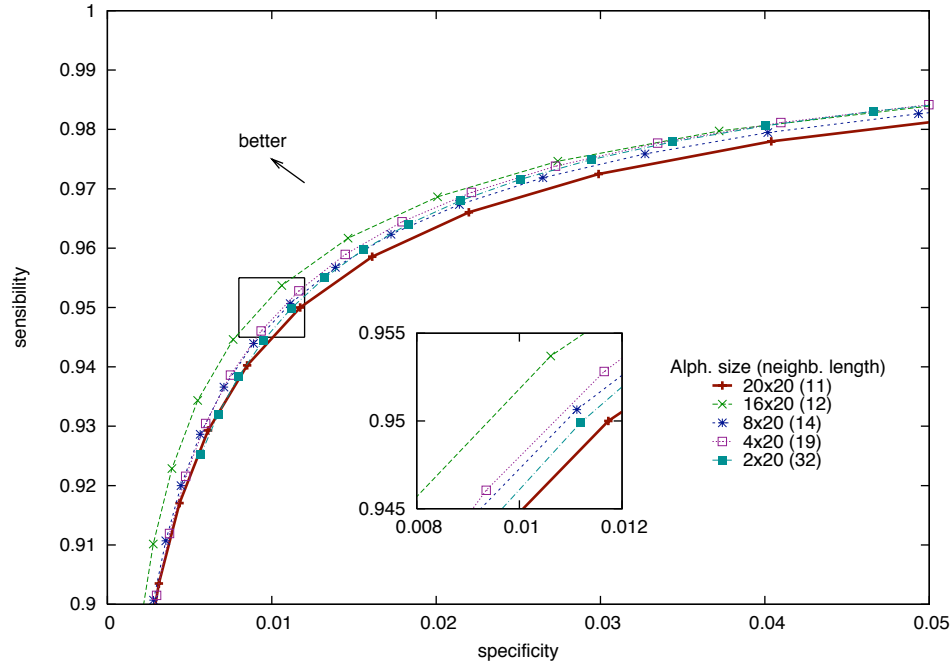
Once an alphabet and a sensitivity/selectivity ratio are chosen, several parameters are computed. Substitution score matrix and *e*-value parameters depend only on the alphabet and the model probabilities, whereas the optimal neighborhood size and the threshold depends also on the sensitivity/selectivity level.

**Figure 6 - Sensitivity/selectivity trade-off using different alphabets with a constant neighborhood length**



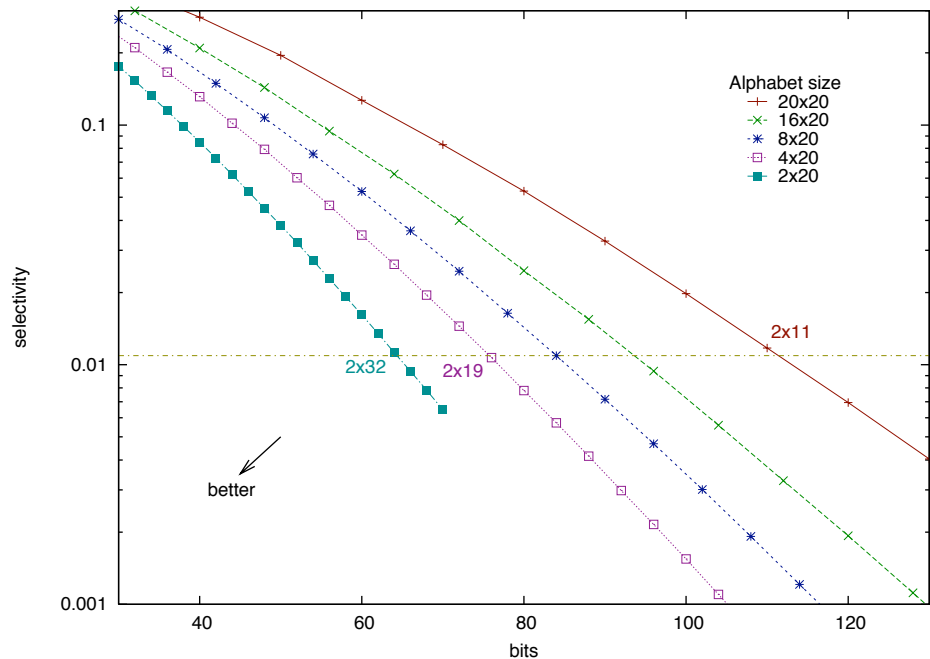
Sensitivity/selectivity trade-off for two neighborhoods of length 11 (other lengths give similar results). When the length is fixed, reduced alphabets provide worse results than the  $\Sigma_{20} \times \Sigma_{20}$  alphabet. The curves for alphabets  $\Sigma_{20} \times \Sigma_4$  and  $\Sigma_{20} \times \Sigma_2$ , not shown, are even worse.

**Figure 7 - Sensitivity/selectivity trade-off using different alphabets with adapted neighborhood lengths**



Sensitivity/selectivity trade-off for two neighborhoods with the adapted lengths of Table 1. Now all reduced alphabets are equivalent (or slightly better, due to integer rounding of the neighborhood lengths) than the original alphabet  $\Sigma_{20} \times \Sigma_{20}$ .

**Figure 8 - Memory for neighborhood storage for different alphabets at a fixed sensitivity**



Memory space needed to achieve a sensitivity close to 0.95. The same quality can be achieved with 64 bits (2 neighborhoods of 32 amino acids encoded in 1 bit, sensitivity of 0.9499, selectivity of 0.0112) instead of 110 bits (2 neighborhoods of 11 amino acids encoded in 5 bits, sensitivity of 0.9500, selectivity of 0.0111). All reduced rectangular alphabets lead to smaller index sizes than the regular  $\Sigma_{20} \times \Sigma_{20}$  alphabet.

## Tables

**Table 1 - Memory for neighborhood storage for different alphabets with adapted neighborhood lengths**

Memory needed for storing neighborhoods for sensitivity/selectivity trade-offs shown on Figure 7.

indexed neighborhood alphabet	bits per character ( $\alpha$ )	neighborhoods length ( $L$ )	total per index line ( $2\alpha L$ )	relative gain compared to $\Sigma_{20}$ ( $1 - 2\alpha L/110$ )
$\Sigma_{20}$	5	11	110	0%
$\Sigma_{16}$	4	12	96	13%
$\Sigma_8$	3	14	84	24%
$\Sigma_4$	2	19	76	31%
$\Sigma_2$	1	32	64	42%

**Table 2 - REBLOSUM matrix for alphabet  $\Sigma_{20} \times \Sigma_{16}$** 

Matrix REBLOSUM62 for alphabet  $\Sigma_{20} \times \Sigma_{16}$ . Scores located on the “diagonal” are shown in bold.

	<i>C</i>	<i>F</i>	<i>Y</i>	<i>W</i>	<i>M</i>	<i>L</i>	<i>I</i>	<i>V</i>	<i>G</i>	<i>P</i>	<i>A</i>	<i>T</i>	<i>S</i>	<i>N</i>	<i>H</i>	<i>Q</i>	<i>E</i>	<i>D</i>	<i>R</i>	<i>K</i>
[ <i>C</i> ]	<b>9</b>	-2	-2	-2	-1	-1	-1	-1	-3	-3	0	-1	-1	-3	-3	-3	-4	-3	-3	-3
[ <i>FY</i> ]	-2	<b>5</b>	<b>5</b>	1	0	0	-1	-1	-3	-3	-2	-2	-2	-3	0	-2	-3	-3	-2	-3
[ <i>W</i> ]	-2	1	2	<b>11</b>	-1	-2	-3	-3	-2	-4	-3	-2	-3	-4	-2	-2	-3	-4	-3	-3
[ <i>ML</i> ]	-1	0	-1	-2	<b>3</b>	<b>4</b>	1	1	-3	-3	-1	-1	-2	-3	-2	-2	-3	-3	-2	-2
[ <i>IV</i> ]	-1	0	-1	-3	1	1	<b>3</b>	<b>3</b>	-3	-3	-1	0	-2	-3	-3	-2	-3	-3	-3	-2
[ <i>G</i> ]	-3	-3	-3	-2	-3	-4	-4	-3	<b>6</b>	-2	0	-2	0	0	-2	-2	-2	-1	-2	-2
[ <i>P</i> ]	-3	-4	-3	-4	-2	-3	-3	-2	-2	<b>7</b>	-1	-1	-1	-2	-2	-1	-1	-1	-2	-1
[ <i>A</i> ]	0	-2	-2	-3	-1	-1	-1	0	0	-1	<b>4</b>	0	1	-2	-2	-1	-1	-2	-1	-1
[ <i>T</i> ]	-1	-2	-2	-2	-1	-1	-1	0	-2	-1	0	<b>5</b>	1	0	-2	-1	-1	-1	-1	-1
[ <i>S</i> ]	-1	-2	-2	-3	-1	-2	-2	-2	0	-1	1	1	<b>4</b>	1	-1	0	0	0	-1	0
[ <i>N</i> ]	-3	-3	-2	-4	-2	-3	-3	-3	0	-2	-2	0	1	<b>6</b>	1	0	0	1	0	0
[ <i>H</i> ]	-3	-1	2	-2	-2	-3	-3	-3	-2	-2	-2	-2	-1	1	<b>8</b>	0	0	-1	0	-1
[ <i>QE</i> ]	-3	-3	-2	-2	-1	-3	-3	-2	-2	-1	-1	-1	0	0	0	<b>4</b>	<b>4</b>	1	0	1
[ <i>D</i> ]	-3	-3	-3	-4	-3	-4	-3	-3	-1	-1	-2	-1	0	1	-1	0	2	<b>6</b>	-2	-1
[ <i>R</i> ]	-3	-3	-2	-3	-1	-2	-3	-3	-2	-2	-1	-1	-1	0	0	1	0	-2	<b>5</b>	2
[ <i>K</i> ]	-3	-3	-2	-3	-1	-2	-3	-2	-2	-1	-1	-1	0	0	-1	1	1	-1	2	<b>5</b>

**Table 3 - REBLOSUM matrix for alphabet  $\Sigma_{20} \times \Sigma_8$** 

Matrix REBLOSUM62 for alphabet  $\Sigma_{20} \times \Sigma_{16}$ . Scores located on the “diagonal” are shown in bold.

	<i>C</i>	<i>F</i>	<i>Y</i>	<i>W</i>	<i>M</i>	<i>L</i>	<i>I</i>	<i>V</i>	<i>G</i>	<i>P</i>	<i>A</i>	<i>T</i>	<i>S</i>	<i>N</i>	<i>H</i>	<i>Q</i>	<i>E</i>	<i>D</i>	<i>R</i>	<i>K</i>
[ <i>CFYW</i> ]	<b>4</b>	<b>4</b>	<b>4</b>	<b>5</b>	-1	0	-1	-1	-3	-3	-2	-2	-2	-3	0	-2	-3	-3	-3	-3
[ <i>MLIV</i> ]	-1	0	-1	-2	<b>2</b>	<b>3</b>	<b>3</b>	<b>2</b>	-3	-3	-1	-1	-2	-3	-3	-2	-3	-3	-2	-2
[ <i>G</i> ]	-3	-3	-3	-2	-3	-4	-4	-3	<b>6</b>	-2	0	-2	0	0	-2	-2	-2	-1	-2	-2
[ <i>P</i> ]	-3	-4	-3	-4	-2	-3	-3	-2	-2	<b>7</b>	-1	-1	-1	-2	-2	-1	-1	-1	-2	-1
[ <i>ATS</i> ]	-1	-2	-2	-3	-1	-2	-1	-1	0	-1	<b>2</b>	<b>2</b>	<b>2</b>	0	-1	-1	-1	-1	-1	-1
[ <i>NH</i> ]	-3	-2	0	-3	-2	-3	-3	-3	-1	-2	-2	-1	0	<b>5</b>	<b>5</b>	0	0	1	0	0
[ <i>QED</i> ]	-3	-3	-2	-3	-2	-3	-3	-3	-2	-1	-1	-1	0	0	0	<b>3</b>	<b>3</b>	<b>4</b>	0	0
[ <i>RK</i> ]	-3	-3	-2	-3	-1	-2	-3	-2	-2	-1	-1	-1	0	0	0	1	0	-1	<b>4</b>	<b>4</b>

**Table 4 - Parameters  $\lambda$  and  $K$  for different alphabets**

Parameters  $\lambda$  and  $K$  of the Gumbel law for different alphabets, obtained with the corresponding REBLOSUM score matrices.

alphabets	$\lambda$	$K$
$\Sigma_{20} \times \Sigma_{20}$	0.320	0.139
$\Sigma_{20} \times \Sigma_{16}$	0.333	0.143
$\Sigma_{20} \times \Sigma_8$	0.223	0.142
$\Sigma_{20} \times \Sigma_4$	0.212	0.128
$\Sigma_{20} \times \Sigma_2$	0.161	0.101

**Table 5 - Practical results for different alphabets – Quality estimations**

Similarity search results obtained on reduced alphabets. The number of positions tested (validating **Stage 1** only and independent from the chosen alphabet) is  $1.59 * 10^9$ . The practical selectivity is computed dividing the number of positions validating both **Stage 1** and **Stage 2** by the number of positions tested.

alphabets	number of positions validating <b>Stage 1</b> and <b>Stage 2</b>	practical selectivity	number of detected alignments	practical sensitivity
$\Sigma_{20} \times \Sigma_{20}$	$2.14 * 10^6$	$1.35 * 10^{-3}$	650 (all)	1
$\Sigma_{20} \times \Sigma_{16}$	$1.39 * 10^6$	$0.88 * 10^{-3}$	650 (all)	1
$\Sigma_{20} \times \Sigma_{16}$	$0.98 * 10^6$	$0.62 * 10^{-3}$	650 (all)	1
$\Sigma_{20} \times \Sigma_8$	$0.62 * 10^6$	$0.39 * 10^{-3}$	650 (all)	1
$\Sigma_{20} \times \Sigma_4$	$3.14 * 10^6$	$1.98 * 10^{-3}$	650 (all)	1
$\Sigma_{20} \times \Sigma_2$	$2.93 * 10^6$	$1.85 * 10^{-3}$	650 (all)	1

**Table 6 - Practical results for different alphabets – Memory requirements**

Database index size for neighborhood indexing on different alphabets. The first three columns are the same as in Table 1, the other two columns refer to the experience described in section “Practical results”. The index size is equal to  $N \times (\lceil \log_2 N \rceil + 2\alpha L)$ , as explained in the beginning of the paper. Here  $N = 12\,700\,507$  and  $\lceil \log_2 N \rceil = 24$ . The ratio  $r$  is against the size of the index for offset indexing, which is here

$$S_{\text{offset}} = N \times \lceil \log_2 N \rceil = 0.30 * 10^9 \text{ bits} = 38 \text{ MBytes.}$$

alphabet	$\alpha$	$L$	$S_{\text{neighborhood}}$	$r$
$\Sigma_{20}$	5	11	$1.70 * 10^9$ bits = 212 MBytes	5.58
$\Sigma_{16}$	4	12	$1.52 * 10^9$ bits = 190 MBytes	5.00
$\Sigma_8$	3	14	$1.37 * 10^9$ bits = 171 MBytes	4.50
$\Sigma_4$	2	19	$1.27 * 10^9$ bits = 159 MBytes	4.17
$\Sigma_2$	1	32	$1.12 * 10^9$ bits = 140 MBytes	3.67