Filters and seeds approaches for fast homology searches in large datasets

Nadia Pisanti, Mathieu Giraud, Pierre Peterlongo

FILTERS AND SEEDS APPROACHES FOR FAST HOMOLOGY SEARCHES IN LARGE DATASETS

Nadia Pisanti, Mathieu Giraud, and Pierre Peterlongo

15.1 INTRODUCTION

15.1.1 Homologies and Large Datasets

Homologies inside large sequences or a large set of sequences are the key to several molecular biology studies. Similarities between genomic sequences are often traces of common ancestry, and the study of distances between species teaches us about the history of the evolution. Conserved elements between distant species are genes, transcription factors binding sites, transposable elements, or other functional elements.

Basically, homology-finding algorithms aim to detect in nucleic sequences more or less similar fragments, called simply *repeats*. Such fragments can be found within one sequence or in a set of several sequences. The selection pressure is not focused on the only nucleic sequences; for proteins, comparisons on the proteic sequences are often more relevant, and for RNA, the secondary structure can be more conserved than the nucleic sequence [14]. Similarities between sequences are often a first step to other more specific tools applied to the study of particular conserved elements.

On the other hand, the amount of data that biologists are dealing with are growing exponentially. Another recent reason for this relies on next generation

sequencers [56]; they enable faster sequencing of DNA and with lower costs (several orders of magnitude cheaper) than using the original Sanger et al. [54] method. These recent sequencers open the way to new horizons in molecular biology. As claimed in Mardis' review [40], "an astounding potential exists for these technologies to bring enormous change in genetic and biological research and to enhance our fundamental biological knowledge." Extracting pertinent knowledge from these floods of data requires to find similarities and patterns quickly and efficiently.

The exhaustive similarity search, implying dynamic programming [58], suffers from a time bottleneck. If no heuristic is used, then finding similarities between sequences requires a time proportional to the product of their lengths. Such approaches are difficult with large datasets. In practice, the exhaustive similarity search is reserved for small datasets and when the application requires very precise results. On larger datasets, similarities are found using preprocessing or heuristics through *filters*. Some filters have huge success; the most used tool in bioinformatics for sequences comparison in the last two decades, BLAST [2, 1], is based on a filtering heuristics.

15.1.2 Filter Preprocessing or Heuristics

Filters approaches are based on the following idea: occurrences of an approximately repeated fragment must share a certain number of short fragments that are exactly conserved. The search of repeats thus shall focus on regions of high enough concentration of these shared fragments. This idea can be used either for preprocessing data (removing as much as possible portions that can not contain occurrences of repeats), or as a heuristic (anchoring the search of the repeats using the so-called seeds).

In both cases, these approaches are powerful for quickly finding similarities with full sensibility (without missing any information) or high sensibility (possibly missing some information). Since the early 1990s, seed-based filters have been developed in two different directions. *Lossless filters* guarantee that no occurrence is missed (typically exhibiting a seed-based condition that is proven to be necessary and that it is easy to check), whereas *lossy seed filters* propose some seed models as a starting point (and then explore the properties of those models to assess the sensitivity performances of the filter).

Actually, lossless filters and lossy seed filters are highly related. The fundamental idea of both approaches is to focus directly on sequences fragments that are likely to provide a similarity, getting rid of useless computations that may be avoided. There is no precise border between these two concepts.

15.1.3 Contents

Within various contexts, the authors of this chapter developed methods based on lossless filters or on seed-based lossy filters. This chapter will present the lossless filters and the seed-based approaches by describing a brief state-of-the-art process for each method and by presenting the most recent works in these fields. The following section gives some common definitions and introduces basic concepts.

15.2 METHODS FRAMEWORK

15.2.1 Strings and Repeats

We introduce here the terminology used in the forthcoming sections. A *string* is a concatenation of zero or more symbols from an alphabet Σ. A string s of length n on Σ is represented also by $s[0]s[1]\ldots s[n-1]$, where $s[i] \in \Sigma$ for $0 \leq i < n$. The length of a string s is denoted by $|s|$. We denote $s[i, j]$ the *substring* $s[i]s[i+1]\ldots s[j]$ of s. In the following, we also use the notion of *q-gram*. A q-gram is a word (a short string) of length q. If a q-gram occurs in two strings ω and ω', then this q-gram is said to be *shared* by ω and ω'.

We recall that the *Hamming distance* between two words of the same length is the minimal number of substitutions needed to transform the first into the other, whereas the *edit distance* between two words (not necessarily of the same length) is the minimum number of substitutions, insertions and deletions to transform the first into the other. We denote by $d_H(\omega, \omega')$ (respectively, $d_E(\omega, \omega')$) the Hamming distance (respectively, the edit distance) between the two strings ω and ω'.

A set of words whose pairwise Hamming distance or edit distance is bounded by a given threshold is called an *approximate repeat*. To simplify the reading, in the following, the term "repeat" will design an approximate repeat. A *multiple repeat* is a repeat with at least three words. We focus on (L, r, d)-*Hrepeat* and (L, r, d)-*Erepeat*, defined as follows:

Definition 15.1 ((L, r, d)- Hrepeat) *Given a set $\mathcal{S}$ of one or more input strings, a length $L > 0$, an integer $r \geq 2$, and a Hamming distance $0 \leq d < L$, we call a (L, r, d)-Hrepeat a set $\{\omega_1, \ldots, \omega_r\}$ of r words of length L occurring in the sequences of $\mathcal{S}$ such that for all $i, j \in [1, r]$, $d_H(\omega_i, \omega_j) \leq d$.*

Definition 15.2 ((L, r, d)-Erepeat) *Given a set $\mathcal{S}$ of one or more input strings, a length $L > 0$, an integer $r \geq 2$, and an edition distance $0 \leq d < L$, we call a (L, r, d)-Erepeat a set $\{\omega_1, \ldots, \omega_r\}$ of r words having a length of at least $L - d$ occurring in the sequences of $\mathcal{S}$ such that for all $i, j \in [1, r]$, $d_E(\omega_i, \omega_j) \leq d$.*

Both definitions can be used to study repeats inside one sequence ($|\mathcal{S}| = 1$) or between several sequences ($|\mathcal{S}| > 1$). In the latter case, one also can enforce that the r words occur over r distinct sequences (and thus one needs $|\mathcal{S}| \geq r$).

15.2.2 Filters—Fundamental Concepts

For finding multiple repeats, exhaustive methods based on dynamic programming suffer from a theoretical time complexity in $O(n^r)$, where n is the size of each sequence and r is the number of sequences (or the number of repeats searched in a unique sequence). Some optimizations based on string compression achieve a sub-$O(n^r)$ complexity [10], but applications on large datasets remain very difficult. The goal of filters thus is to reduce this factor n considerably by getting rid of almost

all the useless data that only would slow down the real search. Repeats then could be sought in a much reduced dataset. Ideally, this dataset would contain only the searched repeats. All filters start from the same observation that can be explained with the simple Example 15.1: two similar words share at least a certain number of q-grams.

■ **EXAMPLE 15.1 Shared q-Gram**

Words ATAGGAT and ATATGAT are two words of length 7 with Hamming distance equal to 1 (one substitution position 4). Occurring in a set of sequences $\mathcal{S}$, they are occurrences of a (7,2,1)-Hrepeat. These two words share the 3-gram ATA at position 0 and the 3-gram GAT at position 4.

Filters thus are meant as a preprocessing task to any algorithm that finds and localizes repeats, and hence, they can be employed as a preliminary step to any tool designed for finding repeats or an application using repeats.

With *lossless filters*, we refer to methods that filter the data ensuring that no fragments that may contain a repeat are removed; there are no false negatives. In general, however, there can be false positives; otherwise, there would be no difference between a filter (required to be fast) and an exhaustive search (inevitably slow). Besides the speed requirement, a filter is powerful if it is *selective*; that is, it leaves the least amount of false positives.

On the other hand, *lossy filters* may produce false negatives; preprocessing data with such filters may modify the final result. A good lossy filter must be as selective as possible but also as *sensitive* as possible; that is, it generates the least amount of false negatives, thus approaching the full sensitivity guaranteed by lossless filters.

Being lossless or lossy depends on the design of filter, but the same filter can be used for distinct *repeat models* (the kind of repeats searched, their required length, and their minimal frequency) and, hence, switching from lossless to lossy if the conditions happen to require more speed over precision. For example, as shown in Example 15.1, requiring to find at least two (possibly overlapping) 3-grams between words of length 7 leads to a lossless filter for (7,2,1)-Hrepeats. This means that the two members of any (7,2,1)-Hrepeat share at least two 3-grams. However, if the same condition (at least two 3-grams shared) is used for filtering for (7,2,2)-Hrepeats, then the filter become lossy. Its sensitivity can be measured; only 28.5% couples of words $\{\omega, \omega'\}$ with $|\omega| = |\omega'| = 7$ and $d_H(\omega, \omega') \leq 2$ share at least two 3-grams. The filter has thus a 28.5% sensibility.

Moreover, the computation of sensitivity and specificity also depends on the *background probability model*. The 28.5% sensibility of the previous filter was computed on a Bernoulli model consisting of independent and identically distributed nucleotides. Other more elaborated models better reflect the exact nature of biological sequences.

Although some filters can be used as a generic preprocessing step to any tools that finds repeats, specific filters thus are designed often for a particular repeat model and probability model.

Table 15.1 Methods discussed in this chapter

	Hamming distance (substitutions)	Edit distance (+ indels)
Lossless (sensitivity $= 1$)	NIMBUS	QUASAR, SWIFT, TUIUIU
Lossy (sensitivity ≤ 1)	spaced seeds, and their optimizations	

In the following, we expose some recent filtering methods as summed up in Table 15.1. Section 15.3 details lossless filters, and Section 15.4 details lossy seed filters.

15.3 LOSSLESS FILTERS

For large or complex problem instances, lossless filters help to get exact solutions that would have been otherwise prohibitive. They also speed up other heuristics. This section presents a brief history of lossless filters as well as a description of four of these filters. In all cases, the rationale first is to *detect and prove a necessary condition* for a fragment to be part of a repeat and then to *design a fast method* that checks this condition. To be as efficient as possible, the necessary condition should be designed *ad hoc* for the kind of sought repeat. For example, it does matter whether insertions and deletions are admitted; filters on Hamming or edit distance basically only share the rationale we just described.

15.3.1 History of Lossless Filters

Already in 1987, filtering has been suggested as a screening task to speed exact pattern matching algorithms; in [21], an efficient hash function was suggested for these purposes. The first screening method explicitly designed for finding repeats was suggested in [22] as an online algorithm that searches for "frequent elements" in stream data; also in the latter case, the elements to be searched were exact, that is, their occurrences all are required to be identical. The first time that a screening was devised that took into account approximation was in [17] where up to a certain number of mismatches are allowed (a bounded Hamming Distance is tolerated) but only with the purpose of finding (approximate) occurrences of a given pattern inside a given text.

Specifically aiming at computational biology applications, filtering has been employed successfully by several tools as a preprocessing to approximate pattern matching tasks. The QUASAR (Section 15.3.2, [7]) method first applies a necessary condition for two strings to be similar then finds all matches that admit a given limited number of edit operations. This necessary condition also has been used in SWIFT [52] to design a filter that is an evolution of QUASAR. The SWIFT algorithm adds the use of parallelograms in the necessary conditions (see Section 15.3.2). This approach leads to faster and more selective tools.

Both QUASAR and SWIFT filters search for fragments of the text that fulfill a condition that can be seen as a requirement for two words to belong to a $(L, 2, d)$-Erepeats. In this sense, they are both ancestors of NIMBUS (sections 15.3.3) and TUIUIU (Section 15.3.4), filters for finding *multiple* repeats.

15.3.2 QUASAR and SWIFT—Filtering Repeats with Edit Distance

The necessary condition of QUASAR [7] and SWIFT [52] tools can be embedded in any tool, exact or heuristic, that preprocesses the search of long repeats that consist of multiple occurrences with a limited number of insertions, deletions, and substitutions. By long repeats, we refer to repeats whose length of each occurrence is about 50 nucleotides and more.

With respect to QUASAR, SWIFT presents some improvements that lead to shorter execution time and to more specific filtering.

15.3.2.1 Necessary Condition. QUASAR and SWIFT apply the following condition to filter for $(L, 2, d)$-Erepeat. This condition was first introduced in [60].

Theorem 15.1 *The minimum number of q-grams that words of an $(L, 2, d)$-Erepeat must share is*

$$p_2 = L - q + 1 - qd$$

Thus, while filtering for finding $(L, 2, d)$-Erepeats, all sequence fragments of length L that do not share at least p_2 q-grams with another fragment are filtered out. Note that q, user-defined, is thus one of the main parameter of such approaches.

15.3.2.2 QUASAR Implementation. The QUASAR tool slides a window w of length L, checking whether in the sequence there is another fragment of length at most $L - d$ that shares at least p_2 q-grams with w. For finding sets of shared q-grams, the sequences are partitioned into blocks of size $b \geq 2L$ occurring at every b position. Each such block overlaps by at least L characters with its predecessor. Such an approach ensures that any occurrence of a word of an $(L, 2, d)$-Erepeat is always totally contained in at least one such block. The q-grams of each block are indexed using a suffix array. The sliding window is retained if it shares at least p_2 q-grams with at least one such block.

15.3.2.3 SWIFT Implementation. SWIFT is an evolution of QUASAR. The main improvement relies on the use of restricted *parallelograms* that are a shaped area that limits the space search of the shared q-grams. Figure 15.1 shows an example of such a parallelogram. Should two strings w and w' have an edit distance no greater than d, then in the dynamic programming matrix used for computing their alignment within the edit distance, there would be an optimal alignment consisting in a path making at most d vertical or horizontal steps and, thus, involving at most d consecutive

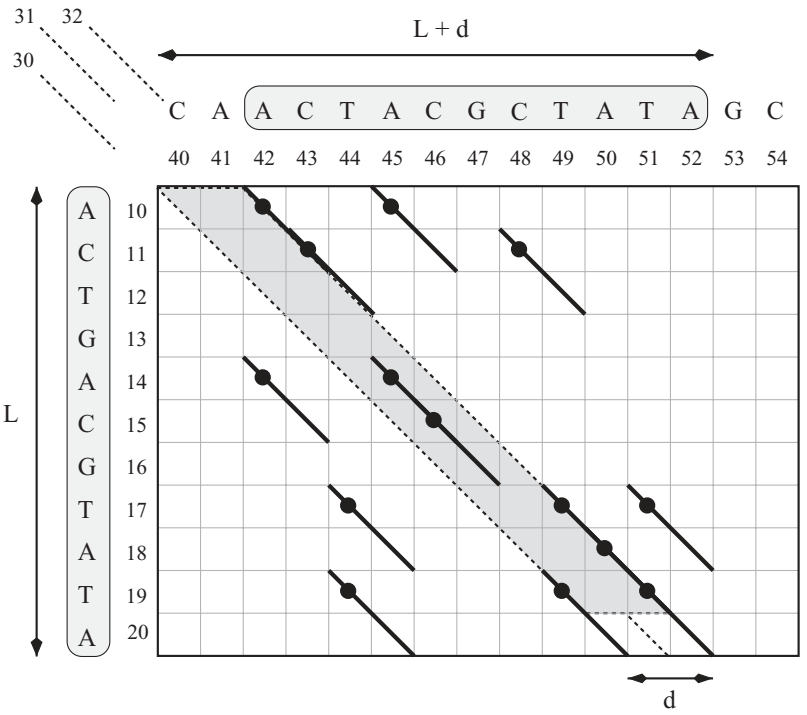

Figure 15.1 An example a $(11, 2, 2)$-repeat found using 2-grams. The shown parallelogram defines portions of sequences in which shared 2-grams must be searched for filtering for finding $(11, 2, 2)$-repeats.

diagonals. Because the matches caused by the mandatory q-grams shared by w and w' necessarily will belong to this path, they would be forced to be in an area restricted by two diagonals that are d positions apart—a parallelogram.

In practice, searching shared q-grams in the area restricted to the parallelograms is an elegant approach leading to major improvements. First, it enables increasing the speed of computations by limiting the working space. Second, it increases the specificity of the filter, avoiding false positives because of q-grams outside the parallelograms.

15.3.3 NIMBUS—Filtering Multiple Repeats with Hamming Distance

The NIMBUS tool is based on the Hamming distance. It has been the first filter designed directly for filtering while searching *multiple* repeats instead of repeats having only two occurrences. The tool NIMBUS [49, 50] can be employed as a preprocessing step to *any* tool that searches for long multiple repeats with mismatches or that performs alignments based on the detection of such local repeats. Being a lossless filter, this tool can be used as a preliminary step of both heuristics and exact methods.

15.3.3.1* NIMBUS *Necessary Condition. NIMBUS uses the following condition to filter for (L, r, d)-Hrepeats:

Theorem 15.2 *The minimum number of nonoverlapping q-grams that words of an (L, r, d)-Hrepeat must share is*

$$p_r = \left\lfloor \frac{L}{q} \right\rfloor - d - (r-2) \times \left\lfloor \frac{d}{2} \right\rfloor$$

NIMBUS uses Theorem 15.2 and hence detects and removes all sequences fragments of length in $[L-d, L+d]$ that do not share at least p_r q-grams with at least $r-1$ other fragments of length in $[L-d, L+d]$. For $r=2$, the formula of Theorem 15.2 coincides with Theorem 15.1.

We now give an insight of the proof. The full proof can be found in [49]. Let us consider the hypothetical alignment of r words of length L of a (L, r, d)-Hrepeat. Suppose $d=0$ (words are all identical), then they share exactly $\left\lfloor \frac{L}{q} \right\rfloor$ nonoverlapping q-grams. Now, if $d>0$, then the number of shared q-grams obviously decreases. In the worst case scenario, every pair of the r strings has Hamming distance d. For each position i in which there is a letter substitution between any pair of strings (positions represented by an x in Figure 15.2), no shared q-gram can include that position, meaning that up to q of them are excluded. Given that there are a total of $d \times r(r-1)/2$ positions in which there is a mismatch, at worst $q \times d \times r(r-1)/2$ q-grams are excluded, and thus, $p'_r = \left\lfloor \frac{L}{q} \right\rfloor - (q \times d \times r(r-1)/2)$ shared q-grams must be left. This would be a very weak necessary condition for a filter.

Observing that the positions of the mismatches between pairs of strings must necessarily overlap, the stronger bound of Theorem 15.2 is obtained. To see why the positions of mismatches must overlap, let us consider the simple case of $r=3$ strings s_1, s_2, and s_3, and $d=1$. Let a be the position of the unique mismatch between s_1 and s_2, let b be that of the mismatch between s_2 and s_3, and let c be that between s_1 and s_3. Because $s_1[a] \neq s_2[a]$, then $s_3[a]$ cannot match with both, and thus, either s_3 mismatches with s_1 at position a (and hence $a=c$), or it mismatches with s_2 there, and therefore, $a=b$; summing up, column a either coincides with b or with c. Moreover (and symmetrically), at position c, it must be that s_1 has a mismatch

Figure 15.2 Words belonging to a (13, 3, 2)-Hrepeat share at least p_3 nonoverlapping 3-grams with $p_3 = \left\lfloor \frac{13}{3} \right\rfloor - 2 - (3-2) \times \left\lfloor \frac{2}{2} \right\rfloor = 1$. Indeed, a 3-gram GTA, for instance, is shared by the three sequences. In this example, mismatches were spread every three characters. This repartition is the most limiting case for the number of shared 3-grams.

with s_2 or with s_3, and hence, actually when $d = 1$, it must be that $a = b = c$; there is a unique column that no q-gram can cross. Clearly, in the general case of $r > 3$ and $d > 1$, there are more than one position of mismatches, but the principles of the mandatory concentration of mismatches positions into columns can be generalized. Roughly speaking, for any d and $r > 2$, each new word contributes by around $d/2$ new columns containing a mismatch.

15.3.3.2 Multiple Sequences Versus a Single Sequence. It is worth noticing that Theorem 15.2 does not depend on the repartition of repeat occurrences over sequences. Thus, it applies either in the case of searching repeats occurring in a single sequence or distributed over at least r sequences.

15.3.3.3 NIMBUS Implementation: Bifactor Array. Sets of shared q-grams are searched efficiently thanks to *bifactors*. Given a sequence, a bifactor is simply two substrings separated with a gap. Example 15.2 shows a bifactor. Each couple of shared q-gram is a shared bifactor. The set of $p_r > 1$ shared q-grams are detected by finding a couple of shared q-grams $A - B$ as shared bifactors. Then a second shared bifactor $B - C$ (starting by B) is searched. In case of success, the group of three shared q-grams $A - B - C$ is found and so on. To speed the detection of shared bifactors, they are indexed in a specialized data structure called the *bifactor array* described in [50].

■ **EXAMPLE 15.2 Bifactor**

On the sequence TATATAGTAC, at position 1, occurs the bifactor ATA GTA, with two substrings of length three separated by a gap of length two. Bifactors are similar to spaced seeds that will be discussed in Section 15.4.2.

15.3.3.4 QUASAR and NIMBUS Implementations Differences. The algorithm of NIMBUS deals with repeats having possibly more than two occurrences, whereas QUASAR was designed for filtering for repeats having only two occurrences. Moreover, QUASAR does not check the order of the shared q-grams. As the edit and Hamming distances do not allow inversions, the shared q-grams must have the same repartition in each word of (L, r, d)-Hrepeats. Thus, the NIMBUS tool, in addition to allowing the finding of repeats with more than two occurrences, applies in practice a method with higher specificity.

15.3.3.5 Performance. Preprocessing with NIMBUS a dataset in which one wants to find functional elements using a multiple local alignment tool such as GLAM [16], the overall execution time could be reduced from 7.5 hours (directly aligning with GLAM only) to less than two minutes (filtering with NIMBUS and then aligning with GLAM) [49].

15.3.4 TUIUIU—Filtering Multiple Repeats with Edit Distance

The TUIUIU [51] tool proposes to extend the filtration for (L, r, d)-Erepeat with $r \geq 2$. The approach uses a filtration condition framework based on the p_2 number of shared q-grams (Theorem 15.1).

With $r \geq 2$, a necessary condition involving the number of q-grams that must be shared by *all* r occurrences of the repeats would result as too weak here because, when indels are allowed, the property of mismatch columns that concentrate does not hold anymore. Therefore, the choice made in TUIUIU actually was to design a very strong necessary condition for two strings to be at most edit distance d and to insert this checking in a suitable framework that detects fragments of the input data that fulfill the requirement with respect to at least $r - 1$ other fragments belonging to distinct input strings. Therefore, the contribution of the algorithm introduced in TUIUIU is twofold; first, a new necessary condition for $(L, 2, d)$-Erepeat is introduced, which results in being stronger that previous ones; second, the framework that extends the necessary condition to multiple repeats, which actually can be employed with any $(L, 2, d)$-Erepeat condition inside.

The necessary condition checked by TUIUIU is actually a series of three possible levels of selectivity, resulting in as many versions of the filter. The first condition (already introduced in [52]) requires that Theorem 15.2 holds with $r = 2$, and also that in the alignment matrix of the two strings, these $p_2 = \lfloor L/q \rfloor - d$ q-grams result in matches that lay in a parallelogram-shaped area (see Figure 15.1). The second (further) condition that TUIUIU imposes is very simple; for w and w' to be a $(L, 2, d)$-Erepeat, the q-grams that they must share have to occur in w at distinct positions (that is, at least p_2 of them). This apparently trivial condition actually resulted in giving a substantial contribution to the strength of the filter in that TUIUIU can check it in negligible constant time, and it does increase the selectivity. For this reason, the performances of the filter version that use this condition clearly outperform those of the filter that uses the first condition only. The third and most stringent condition additionally imposes that there is a set of p_2 shared q-grams that occur in w and w' in the same order. This third condition, involving longest common subsequence (LCS) computations, checks the conservation in the order of the shared q-grams. It requires some extra time to be checked, but experiments showed evidence of the fact that because this is done only for pairs of strings w and w' that already survived the previous conditions, the delay is limited. In practice, this most restrictive constraints resulted in being worth using in many interesting applications, as, for example, while using values of d larger than 10% of L.

The first is the fastest and less sensible and is, in practice, the same as that of SWIFT (the use of a parallelogram), except that it is used within the multiple repeats filtering task. The second option introduces an extra requirement that leads to a more selective tool while being still as fast as before (or actually sometimes even faster). This condition actually could be itself a good filter for finding approximate matches of a pattern into a text. The third choice results in the most selective filter but at a time cost that experiments show to be often worth paying when the target task is finding multiple repeats.

To require that the repeat occurs in r distinct sequences, TUIUIU slides a window over all input sequences. At each moment, it considers the window itself w and all remaining sequences virtually divided into blocks that are candidate to contain w'. For the first position of the window, it builds an index of all its q-grams and stores how many of them belong to each block. For every new position of the window, updating this information is very simple as w simply drops a q-gram and acquires a new one. It is thus also easy to check, for each block, whether it has enough shared q-grams. If for w, there are enough blocks that satisfy the retained, then w is conserved; otherwise, w is filtered out.

Like NIMBUS, TUIUIU also supports a query in which there is a single input sequence, and the r occurrences of the repeat only are required to be distinct as they all must belong to the same sequence.

15.3.4.1 Complexity and Parameters. In general (with p_2 *large enough*, see [51]), the average complexity is in

$$O\left(\frac{b+d}{b} n^2 |\Sigma|^{-q}\right)$$

where n is the sum of the sequence lengths and b the thickness of parallelograms. It is worth noting that q, the length of the q-grams, strongly influences the results and the computation time. With a small q, computation is slow (a lot of shared randomly q-grams), but the filter is very specific. On the other hand, a large value for q increases the theoretical speed both because of the $|\Sigma|^{-q}$ term in the complexity but also because only few large q-grams are shared randomly between sequences. However, with a larger q value, the filter becomes less selective. This is a result of the decrease of the strength of the necessary condition on p_2. In practice, as presented in [51], applying TUIUIU on biological sequences with $q = 6$ presents a good balance between specificity and execution time.

15.3.4.2 Success Story. TUIUIU was applied as a preprocessing step of a multiple alignment application (GLAM2 [15], an evolution of GLAM that allows indels and is thus more suitable with edit distance), leading to an overall execution time (filter plus alignment) on average of 63 and at best 530 times smaller than before (direct alignment).

15.4 LOSSY SEED-BASED FILTERS

When dealing with weaker similarities, lossless filters become too difficult to be implemented efficiently. As an example, when we search for $(40, 2, 10)-$Hrepeats (that means a $\geq 75\%$ similarity), Theorem 15.2 cannot ensure anything for q-grams with $q \geq 4$. In some homology studies between "distant" species, one frequently has to look at similarity levels between 30% and 60%; for those kind of similarities, tools

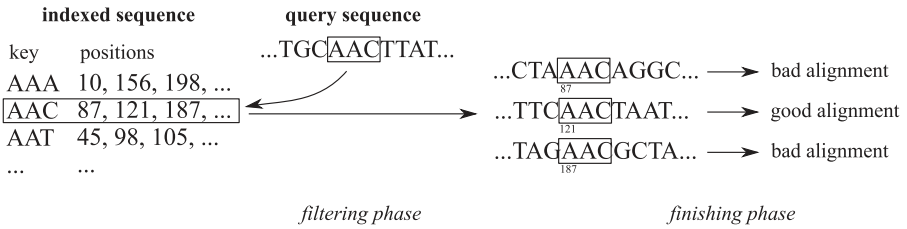

Figure 15.3 During the filtering phase, a scan of the query sequence uses the index (here with the key AAC) to identify all matching seeds. Here the first $r = 3$ seeds are shown. Then the finishing phase get neighborhoods of the indexed sequence from the memory (one memory access per position) and builds local alignments between this neighborhood and the one of the query sequence. Here only the position 121 in the indexed sequence leads to a "good" local alignment.

other than lossless filters must be designed. In this section, we explain seed-based heuristics and detail some of their implementations.

15.4.1 Seed-Based Heuristics

As in lossless filters, the 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 the 20^3 possible keys for the seed of three consecutive characters on the protein alphabet.

We now summarize the discussion made by the excellent survey of Brown ([5], page 122). Seed-based alignment proceeds in three phases (Figure 15.3):

- *Indexing phase:* "indexing one or more of the sequences"—typically in $O(n)$ time, or at most $O(kn)$ time, where n is the size of the indexed sequence and k is the size of keys. For each key, all positions of the occurrences in the database are stored.
- *Filtering phase:* "using the index to identify possible alignment seeds"—typically, each position in the query is processed in $O(1 + r)$ time, where r is the number of matching seeds. Thus, this phase takes $O(m + R)$ time, where m is the size of the query and R is the total number of matching seeds.
- *Finishing phase:* "building local alignments from the seeds"—$O(R)$, assuming that most seeds are bad seeds (that are not extended into good alignments). To be efficient, the finishing step usually begins by a fast alignment between the query and the database on a small *neighborhood* around the seed match, then proceeds to a full alignment if the fast alignment was above a given threshold.

In fact, if we leave out the constant terms and the $O(m + n)$ time required to read the sequences, a seed-based alignment algorithm runs in additional $O(R)$ time;

the specificity of the filtering phase is crucial. Traditional BLAST seeds, noted as ###########, are contiguous 11-grams on nucleotides. Assuming the DNA sequences are random, R is approximately $mn/4^{11}$. With large genomes, this quantity becomes too large.

15.4.2 Advanced Seeds

For the same number of detected good alignments (sensibility), how can it be possible to have less hits and thus to decrease R ? Instead of contiguous k-words, it is more advantageous to use so-called *spaced seeds* that correspond to matches "with gaps" between sequences and thus gapped diagonals in the dynamic programming matrix. Hits of a spaced seed are less related than the hits of a contiguous word; for a same specificity (number of hits), the sensitivity is better.

■ **EXAMPLE 15.3 Spaced Seeds**

With the spaced seed ##-## of weight 4, the nucleic key AA-CT matches the four strings AAACT, AACCT, AAGCT and AATCT. This seed is lossless on $(40, 2, 10)-$Hrepeats: that means that all $(40, 2, 10)-$Hrepeats share at least a gapped 4-gram shaped by ##-##. This is better than the contiguous 4-gram #### that misses some (rare) alignments.

On $(40, 2, 20)-$Hrepeats, both seeds are lossy. On a Bernoulli model, the spaced seed ##-## has now a sensibility of 86,8%, whereas the seed #### only achieves a 79,8% sensibility. Sensibilities are computed with Iedera [27].

The idea of using spaced seeds for biological sequence comparisons first was proposed in 2002 by Ma *et al.* [37] in the PatternHunter software. Following this article, theoretical design and usage of better seeds became an active field of research [6, 11, 32, 38], with extensions on vector seeds [3], protein seeds [4, 24], and subset seeds [27, 53]. The most complete and recent survey of this domain is [5].

In all those seed models, one *designs* appropriate seeds according to sensitivity/selectivity criteria and the class of target alignments. Moreover, instead of using a single seed, one can use several seeds simultaneously (so-called *multiple seeds*) to improve further the sensitivity/selectivity trade-off.

15.4.3 Latencies and Neighborhood Indexing

15.4.3.1 Latencies for the Finishing Step. What exactly is stored during the indexing phase? For each key, we want to remember the list of all its occurrences in the database. In the usual *offset indexing* approach, depicted on Figure 15.3, an offset of $\log N$ bits is stored for each seed position (where N is the size of the database). The index size thus is equal to $N \times \log N$ bits.

For each query position, each hit returned by the filtering phase leads to an iteration of the finishing phase. This iteration accesses some neighborhood of the positions. These memory accesses are *random*, that is, unpredictable and noncontiguous.

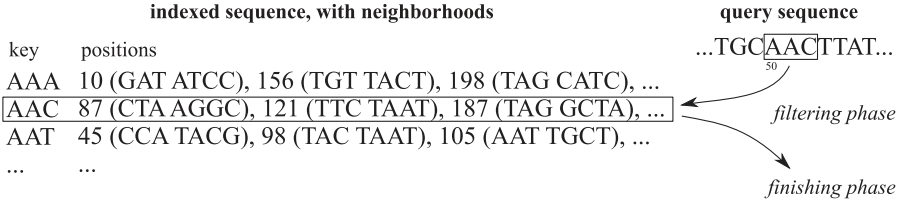

Figure 15.4 In the neighborhood indexing approach, for each key, a small neighborhood of each key occurrence is stored redundantly in the index. Here $L = 7$ nucleotides are stored with each position. The filtering phase needs one memory access (per key). The finishing phase does not need any more additional access.

Such accesses are not cached efficiently and require high latencies [19]. This is especially true when using multiple seeds, for example, in the BlastP approach (on average, 26 index look-ups for the Blosum-62 background distribution of amino acids [48]).

15.4.3.2 Neighborhood Indexing. A way to reduce the computation time thus is to avoid as far as possible such random memory accesses. In [48], a *neighborhood indexing* approach was proposed. The idea is to store additionally, for each key occurrence, its left and right neighborhoods in the sequence (Figure 15.4). 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 memory access. There is no need for further memory accesses to random positions in the original sequence. The overall index size then is equal to $N \times (\log N + \alpha L)$ bits, where α is the number of bits for coding a character (nucleotide or amino acid), and L is the total length of the neighborhoods.

The main advantage of the neighborhood indexing is that it speeds the execution time by a factor ranging between 1.5 and 2 over the offset indexing [48]. 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 between the overall index sizes of the neighborhood indexing and the offset indexing is $1 + \alpha L / \log N$. Usual values for $\log N$ are between 20 and 40, and usual values for L are between between 2×20 and 2×200; hence, this ratio is between 2 and 21.

15.4.3.3 Implementation Detail of the Neighborhood Indexing: Alphabet Reduction. In [48], a reduction of this ratio for the proteic case is proposed. The idea is is to use a reduced amino acid alphabet, and thus to reduce α. Grouping amino acids was studied in several papers [8, 13, 33, 42]. Groups can rely on amino acid physical-chemical properties or on a statistical analysis of alignments. For example, the authors of [42] 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 [8]. An extreme application of the grouping gives alphabets with only two symbols. Li *et al.* proposed in [33] the alphabet $\Sigma_2 = \{\text{CFYWMLIV}, \text{GPATSNHQEDRK}\}$. Using this alphabet for amino acids divides the storage requirement per 5 but is, of course, far less precise.

In [48], we showed that we could retrieve the original sensitivity by using longer neighborhoods and keeping the original alphabet for the query string. We computed *rectangular* BLOSUM matrices (ReBlosum, Example 15.4) that compare amino acid groups (from the indexing alphabet) with actual amino acids (for the query string). Our method applies on any amino acid partition. The best results were obtained with Li's Σ_2 alphabet; with the same sensibility, 35% less memory is neeeded for a neighborhood length of 2×32 instead of 2×11 amino acids when $\log N = 24$.

■ **EXAMPLE 15.4 Alphabet Reduction**

	C	F	Y	W	M	L	I	V	G	P
CFYWMLIV	4	4	3	4	3	4	4	3	−6	−6
GPATSNHQEDRK	−4	−5	−4	−6	−3	−5	−5	−4	2	2
	A	T	S	N	H	Q	E	D	R	K
CFYWMLIV	−2	−2	−4	−6	−4	−4	−6	−7	−5	−5
GPATSNHQEDRK	1	1	2	2	1	2	2	2	2	2

The ReBlosum matrix for comparison of Li's alphabet Σ_2 (indexing alphabet) with the usual alphabet Σ_{20} (querying alphabet) computed on alignments with 62% identity. On a Bernoulli model, comparing a query of length 32 (on Σ_{20}) with an indexed neighborhood of length 32 (on Σ_2) is as sensible as comparing a query of length 11 (on Σ_{20}) with a usual neighborhood on length 11 on Σ_{20} but with a 35% memory reduction.

15.4.4 Seed-Based Heuristics Implementations

15.4.4.1 CPU Implementations. The first widely used implementations of seed-based heuristics were Fasta [46] then Blast [1]. The Blast 2 implementation [2] uses a double hit, allowing a greater sensibility.

PatternHunter [37], followed by PatternHunter 2 [31], are the reference papers for the spaced seeds and the multiple seeds ideas. Another widely used tool, Blat [23], allows one mismatch in a contiguous seed and, thus, is equivalent to a collection of several spaced seeds. Yass [45, 44] is a comparison tool between nucleic sequences that implements several optimisation to seeds, including multiple "transition constrained" seeds, favoring transition mutations between purines and between pyrimidines. Yass come with the Iedera tool [27] that designs and evaluates various types of seeds.

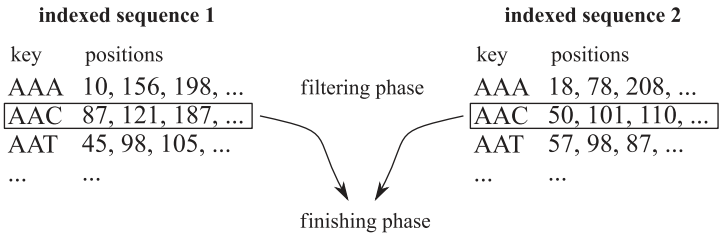

Figure 15.5 ORIS and PLAST seed-based indexing. As the two sequences are indexed, the filtering phase is a simultaneous scan of both indexes. If the neighborhoods also are included in the index, then here there is almost no more random access during both filtering and finishing phases.

A really nice idea appeared in 2008 in ORIS and then PLAST [28, 43]. The idea here is to build two indexes, one for each bank and then to scan simultaneously the both indexes (Figure 15.5). The advantage of such an approach is that there is virtually no cache miss for seeds that are not extended further; all latencies are removed even during the filtering step.

Seed-based heuristics are found in wider applications, for example, in ZOOM, a read mapper for high-throughput sequencers [18].

15.4.4.2 Parallel Implementations. Instead of a serial implementation of a seed-based heuristics, one can parallelize some parts of the computation. Fine-grained parallelism can be realized either through vector single instruction multiple data (SIMD) instructions or on specialized architectures (custom processors or reconfigurable FPGA processors). Moreover, threads on a multicore architecture, or clusters of any of the previous solutions can provide additional coarse-grained parallelism.

- *Blast-like seeds.* The first hardware implementation of a seed-based heuristic was done in 1993 on a custom processor with the BioSCAN architecture [57]. Several implementations on reconfigurable field-programmable gate array (FPGA) processors have been developed independently since 2003 ([9, 20, 26, 41], see [29] for a review). A cluster-enabled version of Blast was proposed on message possing interface (MPI) [12, 59].
- *Other heuristics.* DASH [25] is an algorithm implemented on FPGA with a better sensibility than BLAST. In 2006, [30] proposed an architecture designed to proteic comparisons using seeds of three amino acids. We proposed the implementation on subset seeds [27] on a FPGA architecture coupled with large Flash memories [47]. The PLAST algorithm was designed especially for easy parallelization, and several parallel versions are available [43].

Finally, the new many-cores architectures like graphic processing units (GPUs) also can offer both levels of parallelism. Recently, numerous parallel GPU implementations of regular Smith–Waterman dynamic programming were proposed

[34, 35, 36, 39, 55]. Seed-based heuristics also could take benefit from those architectures.

15.5 CONCLUSION

Sequence homologies are the key to molecular biology studies. Modeled by approximate repeats and approximate multiple repeats, the discovery of homologies remains a difficult task suffering from time bottlenecks. Moreover, molecular biology studies have to cope with datasets whose size grows exponentially. Today biologists often must restrict the area of their research while looking for similarities into sequences or between sequences. Some computations, in particular concerning the research of similarities not limited to two occurrences, are simply unfeasible.

Given a similarity model, filters are methods permitting quick focus on sequences fragments that may contain some repeats occurrences. Thus, after a filtering step, programs designed for the similarity research may be applied to much smaller datasets and find faster repeat occurrences. This chapter exposed two kind of filters. First, lossless filters that ensure that no occurrences of repeats may be filtered out. Second, lossy seed filters that are heuristics that do not assure that no repeat occurrences are missed after filtration.

Used as first steps of similarity research, *lossless filters* present the large advantage that they do not modify the results quality. Although applied for the research of multiple repeats, they can reduce computation time by several orders of magnitude. However this kind of filter generally is limited to the research of well-conserved similarities. On the other hand, *lossy seed filters* are methods widely used that provide even faster results. This kind of filter may be used for finding similarities with a high divergence rate. However, even if some approaches estimate the sensitivity of the results, they cannot ensure finding 100% of the searched results.

Today, new sequencing techniques increase even the exponential rate of the flood of data. New methods must be designed to improve further the similarity search still working for increasing the sensitivity close to the full sensitivity.

15.6 ACKNOWLEDGMENTS

We are grateful to Julien Allali (Université Bordeaux 1) and Jérémie Bourdon (Université de Nantes) for their comments on this chapter.

REFERENCES

1. S.F. Altschul, W. Gish, W. Miller, E.W. Myers, and D.J. Lipman. Basic local alignment search tool. *J Mol Biol*, 215(3):403–410, 1990.
2. S.F. Altschul, T.L. Madden, A.A. Schäffer, J. Zhang, Z. Zhang, W. Miller, and D.J. Lipman. Gapped BLAST and PSI-BLAST: A new generation of protein database search programs. *Nucleic Acids Res*, 25(17):3389–3402, 1997.

3. B. Brejová, D. Brown, and T. Vinar. Vector seeds: An extension to spaced seeds. *J Comput Syst Sci*, 70(3):364–380, 2005.
4. D. Brown. Optimizing multiple seeds for protein homology search. *IEEE Trans Comput Biol Bioinformatics*, 2(1):29–38, 2005.
5. D.G. Brown. A survey of seeding for sequence alignment. In I. Mandoiu and A. Zelikovsky, editor, *Bioinformatics Algorithms: Techniques and Applications*. Wiley-Interscience, New York, 2008, pp. 126–152.
6. J. Buhler, U Keich, and Y. Sun. Designing seeds for similarity search in genomic DNA. *J Comput Syst Sci*, 70(3):342–363, 2005.
7. S. Burkhardt, A. Crauser, P. Ferragina, H.-P. Lenhof, E. Rivals, and M. Vingron. *q*-gram based database searching using a suffix array (QUASAR). *Annual Conference on Research in Computational Molecular Biology (RECOMB 99)*, Lyon, France, 1999, pp. 77–83.
8. N. Cannata, S. Toppo, C. Romualdi, and G. Valle. Simplifying amino acid alphabets by means of a branch and algorithm and substitution matrices. *Bioinformatics*, 18(8):1102–1108, 2002.
9. C. Chang. BLAST implementation on BEE2. University of California at Berkeley, 2004.
10. M. Crochemore, G.M. Landau, and M. Ziv-Ukelson. A subquadratic sequence alignment algorithm for unrestricted scoring matrices. *SIAM J Comput*, 32(6):1654–1673, 2003.
11. M. Csürös and B. Ma. Rapid homology search with two-stage extension and daughter seeds. *International Computing and Combinatorics Conference (COCOON 05)*, Kunming, China, 2005, pp. 104–114.
12. A. Darling, L. Carey, and W. Feng. The design, implementation, and evaluation of mpiBLAST. *ClusterWorld Conference and Expo (CWCE 2003)*, SanJose, CA, 2003.
13. R.C. Edgar. Local homology recognition and distance measures in linear time using compressed amino acid alphabets. *Nucleic Acids Res*, 32(1):380–385, 2004.
14. A. Fontaine, A. de Monte, and H. Touzet. MAGNOLIA: multiple alignment of protein-coding and structural RNA sequences. *Nucleic Acids Res*, 36(S2):W14–W18, 2008.
15. M.C. Frith, N.F.W. Saunders, B. Kobe, and T.L. Bailey. Discovering sequence motifs with arbitrary insertions and deletions. *PLoS Comput Biol*, 4(5):e1000071, 2008.
16. M.C. Frith, U. Hansen, J.L. Spouge, and Z. Weng. Finding functionnal sequence elements by multiple local alignement. *Nucleic Acid Res*, 32(1):189–200, 2004.
17. R. Grossi and F. Luccio. Simple and efficient string matching with k mismatches. *Inf Proces Lett*, 33(3):113–120, 1989.
18. L. Hao, Z. Zefeng, Q.Z. Michael, M. Bin, and L. Ming. ZOOM! Zillions of oligos mapped. *Bioinformatics*, 24(21):2431–2437, 2008.
19. J.L. Hennessy and D.A. Patterson. *Computer Architecture, A Quantitative Approach*. Morgan Kaufmann, Burlington, MA, 2006.
20. A. Jacob, J. Lancaster, J. Buhler, and R. Chamberlain. FPGA-accelerated seed generation in Mercury BLASTP. *Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 07)*, Napa Valley, CA, 2007, pp. 95–106.
21. R.M. Karp and M.O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J Res Dev*, 31(2):249–260, 1987.
22. R.M. Karp, S. Shenker, and C.H. Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *ACM Trans Database Syst*, 28:51–55, 2003.

23. W. James Kent. BLAST - the BLAST-like alignment tool. *Genome Res*, 12:656–664, 2002.
24. D. Kisman, M. Li, B. Ma, and W. Li. tPatternHunter: gapped, fast and sensitive translated homology search. *Bioinformatics*, 21(4):542–544, 2005.
25. G. Knowles and P. Gardner-Stephen. A new hardware architecture for genomic and proteomic sequence alignment. *IEEE Computational Systems Bioinformatics Conference (CSBC 04)*, Stanford, CA, 2004.
26. P. Krishnamurthy, J. Buhler, R. Chamberlain, M. Franklin, K. Gyang, and J. Lancaster. Biosequence similarity search on the Mercury system. *IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP 04)*, Galvaston, TX, 2004.
27. G. Kucherov, L. Noé, and M. Roytberg. A unifying framework for seed sensitivity and its application to subset seeds. *J Bioinformatics Comput Biol*, 4(2):553–569, 2006.
28. D. Lavenier. Ordered index seed algorithm for intensive dna sequence comparison. *IEEE International Workshop on High Performance Computational Biology (HiCOMB 08)*, Miami, FL, 2008.
29. D. Lavenier and M. Giraud. Reconfigurable Computing. *Bioinformatics Applications*. Springer, New York, 2005.
30. D. Lavenier, L. Xinchun, and G. Georges. Seed-based genomic sequence comparison using a FPGA/FLASH accelerator. *Field Programmable Technology (FPT 2006)*, Bangkok, Thailand, 2006, pp. 41–48.
31. M. Li, B. Ma, D. Kisman, and J. Tromp. PaternHunter II: Highly sensitive and fast homology search. *Genome Inform*, 14:164–175, 2003.
32. M. Li, M. Ma, and L Zhang. Superiority and complexity of the spaced seeds. *Symposium on Discrete Algorithms (SODA 06)*, Miami, FL, 2006, pp. 444–453.
33. T.P. Li, K. Fan, J. Wang, and W. Wang. Reduction of protein sequence complexity by residue grouping. *Protein Eng*, 16(5):323–330, 2003.
34. L. Ligowski and W. Rudnicki. An efficient implementation of Smith-Waterman algorithm on GPU using CUDA, for massively parallel scanning of sequence databases. *IEEE International Workshop on High Performance Computational Biology (HiCOMB 09)*, Rome, Italy, 2009.
35. W. Liu, B. Schmidt, G. Voss, and W. Müller-Wittig. GPU-ClustalW: Using graphics hardware to accelerate multiple sequence alignment. *IEEE International Conference on High Performance Computing (HiPC 06)*, volume 4297 of *Lecture Notes in Computer Science (LNCS)*, 2006, pp. 363–374.
36. Y. Liu, D. Maskell, and B. Schmidt. CUDASW++: optimizing Smith-Waterman sequence database searches for CUDA-enabled graphics processing units. *BMC Res Notes*, 2(1):73, 2009.
37. B. Ma, J. Tromp, and M. Li. PatternHunter: faster and more sensitive homology search. *Bioinformatics*, 18(3):440–445, 2002.
38. D. Mak, Y. Gelfand, and G. Benson. Indel seeds for homology search. *Bioinformatics*, 22(14):e341–e349, 2006.
39. S.A Manavski and G. Valle. CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. *BMC Bioinformatics*, 9 Suppl 2:S10, 2008.

40. E.R. Mardis. Next-generation DNA sequencing methods. *Annu Rev Genom Hum Genet*, 9(1):387–402, 2008.
41. K. Muriki, K. Underwood, and R. Sass. RC-BLAST: Towards an open source hardware implementation. *IEEE International Workshop on High Performance Computational Biology (HiCOMB 05)*, Denver, CO, 2005.
42. L.R. Murphy, A. Wallqvist, and L. Ronald. Simplified amino acid alphabets for protein fold recognition and implications for folding. *Protein Eng*, 13(3):149–152, 2000.
43. V.H. Nguyen and D. Lavenier. PLAST: Parallel local alignment search tool. *BMC Bioinformatics*, To appear.
44. L. Noé and G. Kucherov. Improved hit criteria for DNA local alignement. *Bioinformatics*, 5(149), 2004.
45. L. Noé and G. Kucherov. YASS: enhancing the sensitivity of DNA similarity search. *Nucleic Acids Res*, 33:W540–W543, 2005.
46. W.R. Pearson and D.J. Lipman. Improved tools for biological sequence comparison. *Proc. Natl. Acad. Sci.*, 85:3244–3248, 1988.
47. P. Peterlongo, L. Noé, D. Lavenier, G. Georges, J. Jacques, G. Kucherov, and M. Giraud. Protein similarity search with subset seeds on a dedicated reconfigurable hardware. *Parallel Biocomputing Conference (PBC 07)*, volume 4967 of *Lecture Notes in Computer Science (LNCS)*, 2007.
48. P. Peterlongo, L. Noé, D. Lavenier, V.H. Nguyen, G. Kucherov, and M. Giraud. Optimal neighborhood indexing for protein similarity search. *BMC Bioinformatics*, 9(534), 2008.
49. P. Peterlongo, N. Pisanti, F. Boyer, A. Pereira do Lago, and M.-F. Sagot. Lossless filter for multiple repetitions with hamming distance. *J Discrete Algorithm*, 6(3):497–509, 2008.
50. P. Peterlongo, N. Pisanti, F. Boyer, and M.-F. Sagot. Lossless filter for finding long multiple approximate repetitions using a new data structure, the bi-factor array. *International Symposium on String Processing Information Retrieval (SPIRE 05)*, Buenos Aries, Argentina, 2005, pp. 179–190.
51. P. Peterlongo, G.A.T. Sacomoto, A. Pereira do Lago, N. Pisanti, and M.-F. Sagot. Lossless filter for multiple repeats with bounded edit distance. *BMC Algorithm Mol Biol*, 4(3), 2009. To appear.
52. K.R. Rasmussen, J. Stoye, and E.W. Myers. Efficient q-gram filters for finding all ϵ-matches over a given length. *J Comput Biol*, 13(2):296–308, 2006.
53. M. Roytberg, A. Gambin, L. Noé, S. Lasota, E. Furletova, E. Szczurek, and G. Kucherov. Efficient seeding techniques for protein similarity search. *Proceedings of Bioinformatics Research and Development (BIRD 08)*, volume 13 of *Communications in Computer and Information Science*, 2008, pp. 466–478.
54. F. Sanger, S. Nicklen, and A.R. Coulson. DNA sequencing with chain-terminating inhibitors. *Proc Ntl Acad Sci U S A*, 74(12):5463–5467, 1977.
55. M.C Schatz, C. Trapnell, A.L. Delcher, and A. Varshney. High-throughput sequence alignment using graphics processing units. *BMC Bioinformatics*, 8:474, 2007.
56. J. Shendure and H. Ji. Next-generation DNA sequencing. *Nat Biotechnol*, 26(10):1135–1145, 2008.
57. R.K. Singh, S.G. Tell, C.T. White, D. Hoffman, V.L. Chi, and B.W. Erickson. A scalable systolic multiprocessor system for analysis of biological sequences. *Symposium on Research on Integrated Systems*, Seattle, WA, 1993, pp. 168–182.

58. T.F. Smith and M.S. Waterman. Identification of common molecular subsequences. *J Mol Biol*, 147:195–197, 1981.
59. O. Thorsen, B. Smith, C.P. Sosa, K. Jiang, H. Lin, A. Peters, and W. Fen. Parallel genomic sequence-search on a massively parallel system. *International Conference on Computing Frontiers (CF 07)*, Las Vegas, NV, 2007.
60. E. Ukkonen. Approximate string matching with q-grams and maximal matches. *Theor Comput Sci*, 92(1):191–211, 1992.