View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Dagstuhl Research Online Publication Server

Fast Spaced Seed Hashing*

Samuele Girotto[1], Matteo Comin[2], and Cinzia Pizzi[3]

1 Department of Information Engineering, University of Padova, Padova, Italy
2 Department of Information Engineering, University of Padova, Padova, Italy
comin@dei.unipd.it
3 Department of Information Engineering, University of Padova, Padova, Italy
cinzia.pizzi@dei.unipd.it

Abstract

Hashing k-mers is a common function across many bioinformatics applications and it is widely used for indexing, querying and rapid similarity search. Recently, spaced seeds, a special type of pattern that accounts for errors or mutations, are routinely used instead of k-mers. Spaced seeds allow to improve the sensitivity, with respect to k-mers, in many applications, however the hashing of spaced seeds increases substantially the computational time. Hence, the ability to speed up hashing operations of spaced seeds would have a major impact in the field, making spaced seed applications not only accurate, but also faster and more efficient.

In this paper we address the problem of efficient spaced seed hashing. The proposed algorithm exploits the similarity of adjacent spaced seed hash values in an input sequence in order to efficiently compute the next hash. We report a series of experiments on NGS reads hashing using several spaced seeds. In the experiments, our algorithm can compute the hashing values of spaced seeds with a speedup, with respect to the traditional approach, between 1.6x to 5.3x, depending on the structure of the spaced seed.

1998 ACM Subject Classification I.1.2 Algorithms

Keywords and phrases k-mers, spaced seeds, efficient hashing

Digital Object Identifier 10.4230/LIPIcs.WABI.2017.7

1 Introduction

The most frequently used tools in bioinformatics are those searching for similarities, or local alignments, between biological sequences. k-mers, i.e. words of length k, are at the basis of many sequence comparison methods, among which the most widely used and notable example is BLAST [1].

BLAST uses the so-called "hit and extend" method, where a hit consists of a match of a 11-mers between two sequences. Then these matches are potential candidates to be extended and to form a local alignment. It can be easily noticed that not all local alignments include an identical stretch of length 11. As observed in [3] allowing for not consecutive matches increases the chances of finding alignments. The idea of optimizing the choice of the positions for the required matches, in order to design the so called *spaced seeds*, has been investigated in many studies, and it was used in PatternHunter [16], another popular similarity search software.

In general contiguous k-mers counts are a fundamental step in many bioinformatic applications [5, 6, 9, 20, 22, 21, 24]. However, spaced seeds are now routinely used, instead of

* This work was supported by the MIUR PRIN project n. 20122F87B2 "Compositional approaches for the characterization and mining of omics data".

© Samuele Girotto, Matteo Comin, and Cinzia Pizzi;
licensed under Creative Commons License CC-BY
17th International Workshop on Algorithms in Bioinformatics (WABI 2017).
Editors: Russell Schwartz and Knut Reinert; Article No. 7; pp. 7:1–7:14

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

contiguous k-mers, in many problems involving sequence comparison like: multiple sequence alignment [7], protein classification [18], read mapping [23] and for alignment-free phylogeny reconstruction [13]. More recently, it was shown that also metagenome reads clustering and classification can benefit from the use of spaced seeds [4, 8, 19].

A spaced seed of length k and weight $w < k$ is a string over the alphabet $\{1, 0\}$ that contains w '1' and $(k - w)$ '0' symbols. A spaced seed is a mask where the symbols '1' and '0' denote respectively match and don't care positions. The design of spaced seeds is a challenging problem itself, tackled by several studies in the literature [10, 11, 16]. Ideally, one would like to maximize the sensitivity of the spaced seeds, which is however an NP-hard problem [15].

The advantage of using spaced seeds, rather than contiguous k-mers, in biological sequence analysis, comes from the ability of such pattern model to account for mutations, allowing for some mismatches in predefined positions. Moreover, from the statistical point of view, the occurrences of spaced seeds at neighboring sequence positions are statistically less dependent than occurrences of contiguous k-mers [15]. Much work has been dedicated to spaced seeds over the years, we refer the reader to [2] for a survey on the earlier work.

Large-scale sequence analysis often relies on cataloging or counting consecutive k-mers in DNA sequences for indexing, querying and similarity searching. An efficient way of implementing such operations is through the use of hash based data structures, e.g. hash tables. In the case of contiguous k-mers this operation is fairly simple because the hashing value can be computed by extending the hash computed at the previous position, since they share $k - 1$ symbols [17]. For this reason, indexing all contiguous k-mers in a string can be a very efficient process.

However, when using spaced seeds these observations do not longer hold. As a consequence, the use of spaced seeds within a string comparison method generally produces a slow down with respect to the analogous computation performed using contiguous k-mers. Therefore, improving the performance of spaced seed hashing algorithms would have a great impact on a wide range of bioinformatics tools.

For example, from a recent experimental comparison among several metagenomic read classifiers [14], Clark [20] emerged as one of the best performing tools for such a task. Clark is based on discriminative contiguous k-mers, and it is capable of classifying about 3.5M reads per minute. When contiguous k-mers are replaced by spaced seeds, as in Clark-S [19], while the quality of the classification improves, the classification rate is reduced to just 200K reads per minute.

The authors of Clark-S attributed such a difference to the use of spaced seeds. In particular, the possible sources of slowdown are two: the hashing of spaced seeds, and the use of multiple spaced seeds. In fact, Clark-S uses three different spaced seeds simultaneously in its processing. However, while the number of spaced seeds used could explain a 3x slowdown, running Clark-S is 17x slower than the original k-mer based Clark. Thus, the main cause of loss of speed performances can be ascribe to the use of spaced seed instead of contiguous k-mers. A similar reduction in time performance when using spaced seeds is reported also in other studies [4, 18, 23]. We believe that the main cause is the fact that spaced seeds can not be efficiently hashed, as opposed to contiguous k-mers.

In this paper we address the problem of the computation of spaced seed hashing for all the positions in an given input sequence, and present an algorithm that is faster than the standard approach to solve this problem. Moreover, since using multiple spaced seeds simultaneously on the same input string can increase the sensitivity [13], we also developed a variant of our algorithm for simultaneous hashing of multiple spaced seeds.

In general, when computing a hash function there are also other properties of the resulting hash that might be of interest like: bit dependencies, hash distributions, collisions etc. However, the main focus of this paper is the fast computation of spaced seed hashing, using the most simple hash function. Note that our method can be extended to implement, for example, the cyclic polynomial hash used in [17] with no extra costs.

In the next section we briefly summarize the properties of spaced seeds and describe our algorithm, together with a variant for handling multiple seed hashing. Experimental results on NGS reads hashing for various spaced seeds are reported in Section 3. Conclusions are driven in Section 4.

2 Methods

A *spaced-seed* S (or just a seed) is a string over the alphabet $\{1, 0\}$ where the 1s correspond to matching positions. The *weight* of a seed corresponds to the number of 1s, while the overall *length*, or span, is the sum of the number of 0s and 1s.

Another way to denote a spaced seed is through the notation introduced in [12]. A spaced seed can be represented by its *shape* Q that is the set of non negative integers corresponding to the positions of the 1s in the seed. A seed can be described by its shape Q where its weight W is denoted as $|Q|$, and its span $s(Q)$ is equal to $\max Q + 1$. For any integer i and shape Q, the positioned shape $i + Q$ is defined as the set $\{i + k, k \in Q\}$. Let us consider the positioned shape $i + Q = \{i_0, i_1, \ldots, i_{W-1}\}$, where $i = i_0 < i_1 < \ldots < i_{W-1}$, and let $x = x_0 x_1 \ldots x_{n-1}$ be a string over the alphabet $\mathcal{A}$. For any position i in the string x, with $0 \leq i \leq n - s(Q)$, the positioned spaced seed $i + Q$ identifies a string of length $|Q|$ that we call Q-gram. A Q-gram at position i in x is the string $x_{i_0} x_{i_1} \ldots x_{i_{W-1}}$ and it is denoted by $x[i + Q]$.

▶ **Example 1.** Let $Q = \{0, 2, 3, 4, 6, 7\}$, then Q is the seed 10111011, its weight is $|Q| = 6$ and its span is $s(Q) = 8$. Let us consider the string $x = ACTGACTGGA$, then the Q-gram $x[0 + Q] = ATGATG$ can be defined as:

x	A	C	T	G	A	C	T	G	G	A
Q	1	0	1	1	1	0	1	1		
$x[0+Q]$	A		T	G	A		T	G		

Similarly all other Q-grams are $x[1 + Q] = CGACGG$, and $x[2 + Q] = TACTGA$.

2.1 Spaced Seed Hashing

In order to hash any string, first we need to have a coding function from the alphabet $\mathcal{A}$ to a binary codeword. For example let us consider the function $encode : \mathcal{A} \rightarrow \{0, 1\}^{log_2 |\mathcal{A}|}$, with the following values $encode(A) = 00, encode(C) = 01, encode(G) = 10, encode(T) = 11$. Based on this function we can compute the encodings of all symbols of the Q-gram $x[0 + Q]$ as follows:

$x[0+Q]$	A	T	G	A	T	G
encodings	00	11	10	00	11	10

There exist several hashing functions, in this paper we consider the Rabin-Karp rolling hash, defined as $h(x[0 + Q]) = encode(A) * |\mathcal{A}|^0 + encode(T) * |\mathcal{A}|^1 + encode(G) * |\mathcal{A}|^2 + encode(A) * |\mathcal{A}|^3 + encode(T) * |\mathcal{A}|^4 + encode(G) * |\mathcal{A}|^5$. In the original Rabin-Karp rolling hash all math is done in modulo n, here for simplicity we avoid that. In the case of DNA

sequences $|\mathcal{A}| = 4$, that is a power of 2 and thus the multiplications can be implemented with a shift. In the above example, the hashing value associated to the Q-gram $ATGATG$ simply corresponds to the list of encoding in Little-endian: 101100101100.

To compute the hashing value of a Q-gram from its encodings one can define the function $h(x[i+Q])$, for any given position i of the string x, as:

$$h(x[i+Q]) = \bigvee_{k \in Q} \left(encode(x_{i+k}) \ll m(k) * log_2|\mathcal{A}|\right), \tag{1}$$

where $m(k)$ is the number of shifts to be applied to the encoding of the k-th symbols. For a spaced seed Q the function m is defined as $m(k) = |\{i \in Q, \text{ such that } i < k\}|$. In other words, given a position k in the seed, m stores the number of matching positions that appear to the left of k. The vector m is important for the computation of the hashing value of a Q-gram.

► **Example 2.** In the following we report an example of hashing value computation for the Q-gram $x[0+Q]$.

x	A	C	T	G	A	C	T	G	G	A
Q	1	0	1	1	1	0	1	1		
m	0	1	1	2	3	4	4	5		
shifted encodings	00		11«2	10«4	00«6		11«8	10«10		
			1100							
				101100						
					00101100					
							1100101100			
hashing value								101100101100		

The hashing values for the others Q-grams can be determined through the function $h(x[i+Q])$ with a similar procedure. Following the above example the hashing values for the Q-grams $x[1+Q] = CGACGG$ and $x[2+Q] = TACTGA$ are respectively 101001001001 and 001011010011.

In this paper we decided to use the Rabin-Karp rolling hash, because it is very intuitive. There are other hashing functions, like the cyclic polynomial hash, that are usually more appropriate because of some desirable properties like uniform distribution in the output space, universality, higher-order independence [17]. In this paper we will focus on the efficient computation of the Rabin-Karp rolling hash. However, with the same paradigm proposed in the following sections, one can compute also the cyclic polynomial hash by replacing in Eq. (1): the function *encode*(A) with a seed table where the letters of the DNA alphabet are assigned different random 64-bit integers, shifts with rotations, OR with XOR.

2.2 Efficient Spaced Seed Hashing

In many applications [4, 7, 13, 18, 19, 23] it is important to scan a given string x and to compute the hashing values over all positions. In this paper we want to address the following problem.

► **Problem 1.** *Let us consider a string $x = x_0x_1 \dots x_i \dots x_{n-1}$, of length n, a spaced seed Q and an hash function h that maps strings into a binary codeword. We want to compute the hashing values $\mathcal{H}(x,Q)$ for all the Q-grams of x, in the natural order starting from the first position 0 of x to the last $n - s(Q)$:*

$$\mathcal{H}(x,Q) = \langle h(x[0+Q]), h(x[1+Q]), \dots h(x[n-s(Q)]) \rangle .$$

Clearly, in order to address Problem 1, it is possible to use Equation 1 for each position of x. Note that, in order to compute the hashing function $h(x[i+Q])$ for a given position, the number of symbols that have to be extracted from x and encoded into the hash is equal to the weight of the seed $|Q|$. Thus such an approach can be very time consuming, requiring the encoding of $|Q|(n - s(Q))$ symbols. In summary, loosely speaking, in the above process each symbol of x is read and encoded into the hash $|Q|$ times.

In this paper we present a solution for Problem 1 that is optimal in the number of encoded symbols. The scope of this study is to minimize the number of times that a symbol needs to be read and encoded for the computation of $\mathcal{H}(x, Q)$. Since the hashing values are computed in order, starting from the first position, the idea is to speed up the computation of the hash at a position i by reusing part of the hashes already computed at previous positions.

As mentioned above, using Equation (1) in each position of an input string x is a simple possible way to compute the hashing values $\mathcal{H}(x, Q)$. However, we can study how the hashing values are built in order to develop a better method. For example, let us consider the simple case of a contiguous k-mers. Given the hashing value at position i it is possible to compute the hashing for position $i+1$, with three operations: a rotation, the deletion of the encoding of the symbol at position i, and the insertion of the encoding of the symbol at position $i+k$, since the two hashes share $k-1$ symbols. In fact in [17] the authors showed that this simple observation can speed up the hashing of a string by recursively applying these operations. However, if we consider the case of a spaced seed Q, we can clearly see that this observation does not hold. In fact, in the above example, two consecutive Q-grams, like $x[0+Q] = ATGATG$ and $x[1+Q] = CGACGG$, do not necessarily have much in common.

In the case of spaced seeds the idea of reusing part of the previous hash to compute the next one needs to be further developed. More precisely, because of the shape of a spaced seed, we need to explore not only the hash at the previous position, but all the $s(Q)-1$ previous hashes.

Let us assume that we want to compute the hashing value at position i and that we already know the hashing value at position $i-j$, with $j < s(Q)$. We can introduce the following definition of $\mathcal{C}_j = \{k - j \in Q : k \in Q \wedge m(k-j) = m(k) - m(j)\}$ as the positions in Q that after j shifts are still in Q with the propriety of $m(k-j) = m(k) - m(j)$. In other words, if we are processing the position i of x and we want to reuse the hashing value already computed at position $i-j$, $\mathcal{C}_j$ represents the symbols of $h(x[i-j+Q])$ that we can keep while computing $h(x[i+Q])$. More precisely, we can keep the encoding of $|\mathcal{C}_j|$ symbols from that hash and insert the remaining $|Q| - |\mathcal{C}_j|$ symbols at positions $Q \setminus C_j$.

▶ **Example 3.** If we know the first hashing value $h(x[0+Q])$ and we want to compute the second hash $h(x[1+Q])$, the following example show how to construct C_1.

k		0	1	2	3	4	5	6	7
Q		1	0	1	1	1	0	1	1
Q«1	1	0	1	1	1	0	1	1	
m(k)		0	1	1	2	3	4	4	5
m(k)−m(1)	−1	0	0	1	2	3	3	4	
C_1				2	3			6	

The symbols at positions $C_1 = \{2, 3, 6\}$ of the hash $h(x[1+Q])$ have already been encoded in the hash $h(x[0+Q])$ and we can keep them. In order to complete $h(x[1+Q])$, the remaining $|Q| - |\mathcal{C}_1| = 3$ symbols need to be read from x at positions $i+k$, where $i = 1$ and $k \in Q \backslash C_1 = \{0, 4, 7\}$.

x	A	C	T	G	A	C	T	G	G	A
$x[0+Q]$	A		T	G	A		T	G		
C_1				2	3			6		
$Q\backslash C_1$		0				4			7	
$x[1+Q]$		C		G	A	C		G	G	

Note that the definition of $|\mathcal{C}_j|$ is not equivalent to the overlap complexity of two spaced seeds, as defined in [11]. In some cases, like the one presented above, the overlap complexity coincides with $|\mathcal{C}_1| = 3$. However, there are other cases where $|\mathcal{C}_j|$ is smaller then the overlap complexity.

▶ **Example 4.** Let us consider the hash at position 2 $h(x[2+Q])$, and the hash at position 0 $h(x[0+Q])$. In this case we are interested in $\mathcal{C}_2$.

k			0	1	2	3	4	5	6	7
Q			1	0	1	1	1	0	1	1
Q«2	1	0	1	1	1	0	1	1		
m(k)			0	1	1	2	3	4	4	5
m(k)−m(2)	−1	0	0	1	2	3	3	4		
C_2			0				4			

The only symbols that can be preserved from $h(x[0+Q])$ in order to compute $h(x[2+Q])$ are those at positions 0 and 4, whereas the overlap complexity is 3.

For completeness we report all values of $\mathcal{C}_j$:

$$\mathcal{C} = <\mathcal{C}_1, \ldots, \mathcal{C}_7> \\ = <\{2,3,6\}, \{0,4\}, \{0,3,4\}, \{0,2,3\}, \{2\}, \{0\}, \{0\}> \ .$$

In order to address Problem 1, we need to find, for a given position i, the best previous hash that ensures to minimize the number of times that a symbol needs to be read and encoded, in order to compute $h(x[i+Q])$. We recall that $|\mathcal{C}_j|$ represents the number of symbols that we can keep from the previous hash at position $i-j$, and thus the number of symbols that needs to be read and encoded are $|Q \setminus C_j|$. To solve Problem 1 and to minimize the number of symbols that needs to be read, $|Q \setminus C_j|$, it is enough to search for the j that maximizes $|\mathcal{C}_j|$. The best previous hash can be detected with the following function:

$$ArgBH(s) = \arg\max_{j \in [1,s]} |\mathcal{C}_j| \, .$$

If we have already computed the previous j hashings, the best hashing value can be found at position $i - ArgBH(j)$, and will produce the maximum saving $|\mathcal{C}_{ArgBH(j)}|$ in terms of symbols that can be kept. Following the above observation we can compute all hashing values $\mathcal{H}(x,Q)$ incrementally, by using dynamic programming as described by the pseudocode of Algorithm 1.

The above dynamic programming algorithm scans the input string x and computes all hashing value according to the spaced seed Q. In order to better understand the amount of savings we evaluate the above algorithm by counting the number of symbols that are read and encoded. First, we can consider the input string to be long enough so that we can discard the transient of the first $s(Q)-1$ hashes. Let us continue to analyze the spaced seed 10111011. If we use the standard function $h(x[i+Q])$ to compute all hashes, each symbol of x is read $|Q| = 6$ times. With our algorithm, we have that $|\mathcal{C}_{ArgBH(7)}| = 3$ and thus

Algorithm 1 Fast Spaced Seed Hashing

```
1:  for i := 0 to |x| − s(Q) do
2:    if (i == 0) then
3:      h_0 := compute h(x[0 + Q]);
4:    else if (i < s(Q) − 1) then
5:      h_i := h_{i−ArgBH(i)} ≫ m(ArgBH(i)) ∗ log_2|A|;
6:      for all k ∈ Q\C_{ArgBH(i)} do
7:        insert encode(x_{i+k}) at position m(k) ∗ log_2|A| of h_i;
8:      end for
9:    else
10:     h_i := h_{i−ArgBH(s(Q)−1)} ≫ m(ArgBH(s(Q) − 1)) ∗ log_2|A|;
11:     for all k ∈ Q\C_{ArgBH(s(Q)−1)} do
12:       insert encode(x_{i+k}) at position m(k) ∗ log_2|A| of h_i;
13:     end for
14:   end if
15: end for
```

half of the symbols do need to be encoded again, overall each symbol is read 3 times. The amount of saving depends on the structure of the spaced seed. For example, the spaced seed 10101010101, with the same weight $|Q| = 6$, is the one that ensures the best savings ($|\mathcal{C}_{ArgBH(10)}| = 5$). In fact, with our algorithm, we can compute all hashing values while reading each symbol of the input string only once, as with contiguous k-mers. To summarize, if one needs to scan a string with a spaced seed and to compute all hashing values, the above algorithm guarantees to minimize the number of symbols to read.

2.3 Efficient Multiple Spaced Seed Hashing

Using multiple spaced seeds, instead of just one spaced seed, is reported to increase the sensitivity [13]. Therefore, applications that exploit such an observation (for example [4, 8, 19]) will benefit from further speedup that can be obtained from the information already computed from multiple spaced seeds.

Our algorithm can be extended to accommodate the need of hashing multiple spaced seeds simultaneously, without backtracking. Let us assume that we have a set $S = s_1, s_2, \ldots, s_{|S|}$ of spaced seeds, from which we can compute the corresponding vectors m_{s_i}. To this purpose, Algorithm 1 needs to be modified as follows. First of all, a new cycle (between steps 2 and 14) is needed to iterate the processing among the set of all spaced seeds. Next, $\mathcal{C}_j$ needs to be redefined so that it compares not only a given spaced seed with itself, but all spaced seeds vs all. In the new definition, $\mathcal{C}_j^{yz} = \{k - j \in s_y : k \in s_z \wedge m_{s_y}(k - j) = m_{s_z}(k) - m_{s_z}(j)\}$, evaluates the number of symbols in common between the seed s_y and the j-th shift of the seed s_z. The function $\mathcal{C}_j^{yz}$ allows to identify, while computing the hash of s_y, the number of symbols in common with the j-th shift of seed s_z. Similarly, we need to redefine $ArgBH(i)$ so that it detects not only the best previous hash, but also the best seed. We define $ArgBSH(y, s) = \arg\max_{z \in [1,|S|], j \in [1,s]} |\mathcal{C}_j^{yz}|$ that returns, for the seed s_y, the pair (s_z, j) representing the best seed s_z and best hash j. With these new definitions we can adjust our algorithm so that, while computing the hash of s_y for a given position i, it can start from the best previous hash identified by the pair $ArgBSH(y, s) = (s_z, j)$. The other steps for the insertion of the remaining symbols (steps 6–7 and 11-12) do not need to be modified.

Table 1 The nine spaced seeds used in the experiments grouped according to their type.

Spaced seeds maximizing the hit probability[19]	
Q1	11110111011100101110010110111111
Q2	1111101011100101101110011011111
Q3	1111101001110101101100111011111
Spaced seeds minimizing the overlap complexity[10]	
Q4	1111010111010011001110111110111
Q5	1110111011101111010010110011111
Q6	1111101001011100111110101101111
Spaced seeds maximizing the sensitivity[10]	
Q7	1111011110011010111110101011011
Q8	1110101011101100110100111111111
Q9	1111110101101011100111011001111

3 Results and discussion

In this section we will discuss the improvement in terms of time speedup of our approach ($T_{FastHash}$) with respect to the time T_{Eq1} needed for computing spaced seeds hashing repeatedly using Eq. (1): speedup $= \frac{T_{Eq1}}{T_{FastHash}}$.

3.1 Spaced seeds and datasets description

The spaced seeds we used have been proposed in literature as maximizing the hit probability [19], minimizing the overlap complexity [10] and maximizing the sensitivity [10]. We tested nine of such spaced seeds, three for each category. The spaced seeds are reported in Table 1 and labeled Q1, Q2, . . . , Q9. Besides these spaced seeds, we also tested Q0, which corresponds to an exact match with a 22mer (all 22 positions are set to 1), and Q10, a spaced seed with repeated '10' and a total of 22 symbols equal to '1'. All spaced seeds Q0–Q10 have the same weight $|Qi| = 22$. Furthermore, in order to compare seeds with different weights but similar density, we computed with rasbhari two sets of seeds with weights 11 and 32 and lengths respectively 16 and 45 (see Tables 3 and 4 in the Appendix).

The datasets we used were taken from previous scientific papers on metagenomic read binning and classification [9, 25]. We considered both simulated datasets (S,L,R), and synthetic datasets (MiSeq, HiSeq, MK_a1, MK_a2, and simBA5). The datasets S_x and L_x contain sets of paired-end reads of length approximately 80bp generated according to the Illumina error profile with an error rate of 1%, while the datasets R_x contain Roche 454 single-end long reads of length approximately 700bp, and a sequencing error of 1%. The synthetic datasets represent mock communities built from real shotgun reads of various species. Table 2 shows, for each dataset, the number of reads and their average length.

All the experiments where run on a laptop equipped with an Intel i74510U cpu at 2GHz, and 16 GB RAM.

3.2 Analysis of the time performances

Figure 1 plots, for each spaced seed, the speedup that is obtainable with our approach with respect to the standard hashing computation. As a reference, the baseline given by the standard approach is about 17 minutes to compute the hash for a given seed on all datasets.

Table 2 Number of reads and average lengths for each of the dataset used in our experiments.

Datasets	number of reads	avg. read length
S6	1426457	80
S7	3307100	80
S9	4468336	80
S10	9981172	80
L5	1016418	80
L6	1182178	80
HiSeq	9989713	91
simBA5	5439738	100
MixK1	9629886	101
MixK2	7149900	101
MiSeq	9933556	131
R7	290473	702
R8	374576	715
R9	588256	715

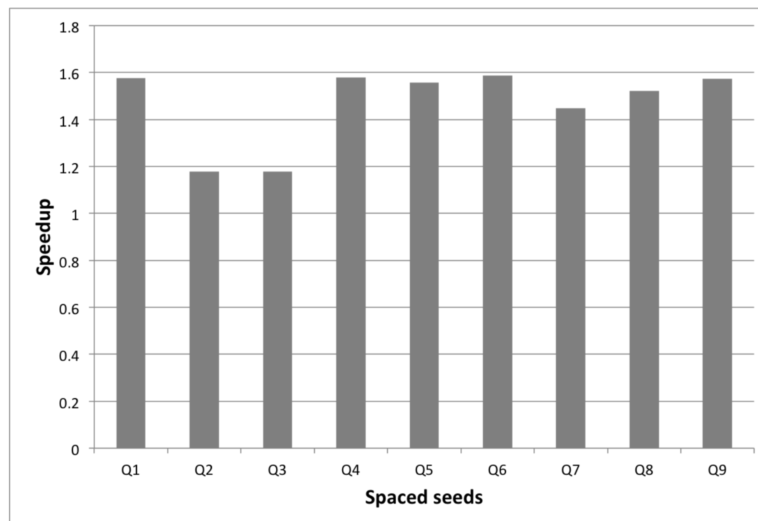

Figure 1 The speedup of our approach with respect to the standard hashing computation, as a function of the spaced seeds used in our experiments.

First of all it can be noticed that our approach improves over the standard algorithm for all of the considered spaced seeds. The smallest improvements are for the spaced seeds Q2 and Q3, both belonging to the class of spaced seeds maximizing the hit probability, for which the speedup is almost 1.2x, and the running time is about 15 minutes. For all the other spaced seeds the speedup is close to 1.6x, thus saving about 40% of the time required by the standard computation, and ending the computation in less than 11 minutes on average.

Figure 2 shows the performances of our approach with respect to the single datasets. In this experiment we considered the best performing spaced seed in each of the classes that we considered, namely Q1, Q6, and Q9, and the two additional special cases Q0 and Q10.

We notice that for the spaced seeds Q0 and Q10 the standard approach requires respectively, 12 and 10 minutes, to process all datasets. This is already an improvement of the standard method with respect to the 17 minutes required with the other seeds Q1–Q9.

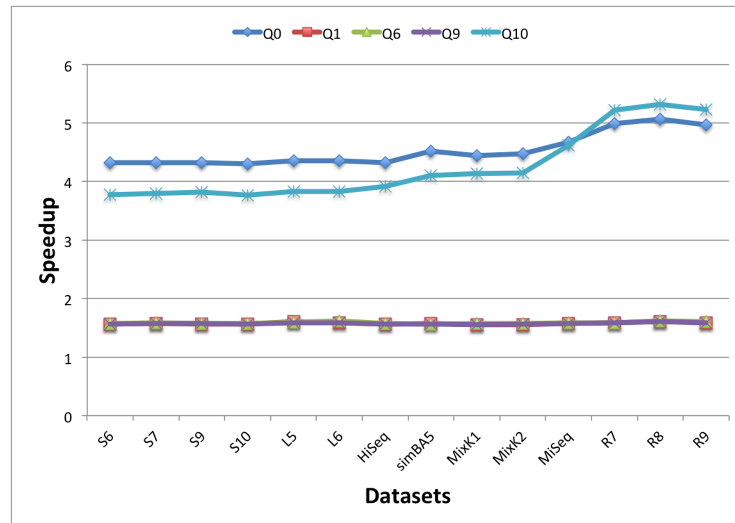

Figure 2 Details of the speedup on each of the considered datasets. Q0 is the solid 22mer, Q10 is the spaced seed with repeated 10. The other reported spaced seeds are the ones with the best performances for each class: Q1 (maximizing the hit probability), Q6 (minimizing the overlap complexity) and Q9 (maximizing the sensitivity).

Nevertheless, with our algorithm the hashing of all dataset can be completed in just 2.7 minutes for Q0 e 2.5 minutes for Q10, with a speedup of 4.5x and 4.2x.

We observe that while the speedup for the spaced seeds Q1, Q6, and Q9 is basically independent on the dataset and about 1.6x, the speedup for both the 22-mer Q0 and the 'alternate' spaced seed Q10 is higher, spanning from 4.3x to 5.3x, depending on the seed and on the dataset. In particular, the speedup increases with the length of the reads and it achieves the highest values for the long read datasets R_7, R_8 and R_9. This behavior is expected, as these datasets have longer reads with respect to the others, thus the effect of the initial transient is mitigated.

3.3 Multiple spaced seed hashing

When the analysis of biological data to perform requires the use of multiple spaced seeds, it is possible to compute the hash of all seeds simultaneously while reading the input string with the method described in Section 2.3.

In Figure 3 we report the comparison between the speedup we obtained when computing the hash for each spaced seed Q1,...,Q9 independently (light grey), and the speedup we obtained when using the multiple spaced seeds approach (dark grey).

In most cases, multiple spaced seed hashing allows for a further improvement of about 2–5%, depending on the dataset. In terms of absolute values, the standard computation to hash all datasets requires 159 minutes, the computation of all seeds independently with the approach described in Section 2.2 takes 109 minutes, while the simultaneous computation of multiple spaced seeds with our method (see Section 2.3) takes 107 minutes. When considering all datasets the average speedup increases from 1.45x (indipendent computation) to 1.49x (simultaneous computation). The small improvement can be justified by the fact that the spaced seeds considered are by construction with minimal overlap.

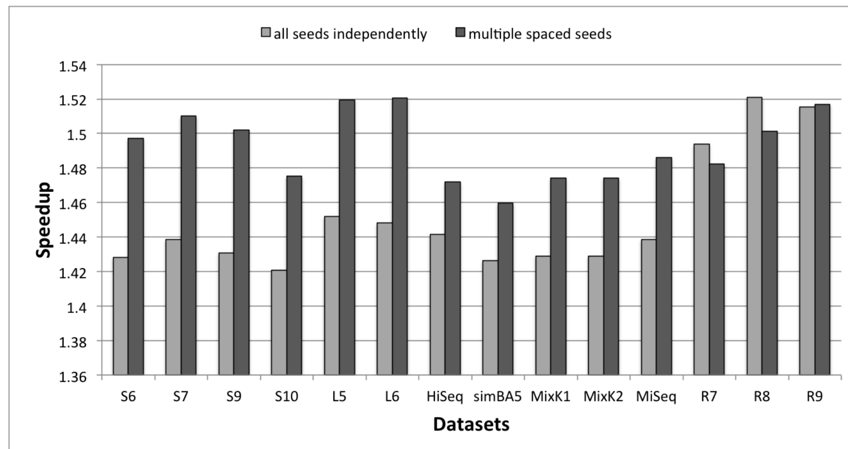

Figure 3 Details of the time speedup of our approach with the multiple spaced seeds hashing (dark grey) and of our approach with each spaced seed hashed independently (light grey).

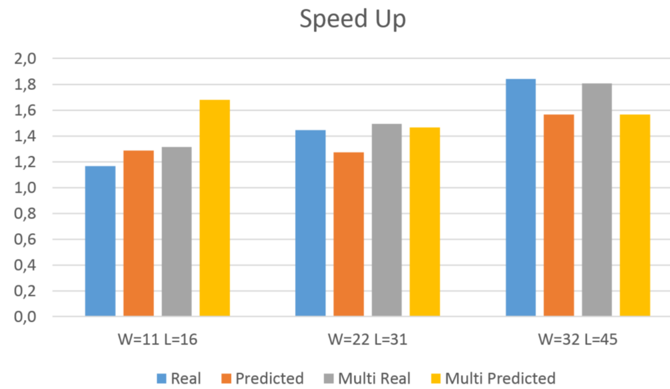

Figure 4 The theoretical and real speedup of our approach with respect to the standard hashing computation, as a function of the spaced seeds weight.

3.4 Spaced Seeds with Different Weights

In order to compare the performance of our method on spaced seeds with different weights we generated other two sets of nine spaced seeds with rasbhari, all with similar density (see Tables 3 and 4 in the Appendix). In Figure 4 are reported the average speedup (Real), over all datasets, for the three different groups of nine seeds. In the same Figure we include also the speedup when all nine seeds are used simultaneously (Multi) and the theoretical speedup predicted by our method (Predicted).

It can be observed that if the weight of the seeds grows then also the real speedup grows. This is expected, because if a seed has more 1s, then the chances to reuse part of the seed increase. As, for the theoretical predicted speedups, these are usually in line with the real speedups even if the absolute values are not necessarily close. We suspect that the model we use, where shifts and insertions have the same cost, is too simplistic. Probably, the real computational cost for the insertion of a symbol is greater than the cost for shifting, and also cache misses might play a role.

If the theoretical speedup for multiple seeds is greater than the theoretical speedup for independent seeds, this indicates that in principle, with multiple seeds, it is possible to

improve with respect to the computation of seeds independently. It is interesting to note that the real results confirm these predictions. For example, in the multiple seeds with weights 32, it is impossible to improve both theoretically and in practice. In the other two cases, the computation of multiple seeds is faster in practice as correctly predicted by the theoretical speedup.

4 Conclusions

We presented a new approach for spaced seeds hashing that exploits the information available from previous matches in order to minimize the number of positions that need to be recomputed. The experiments we performed on several datasets showed that our method has a speedup of 1.6x with respect to the standard approach used to compute spaced seeds hashing, for several kind of spaced seeds defined in the literature. Furthermore, the gain greatly improved in special cases, where seeds show a high autocorrelation, and for which a speed up of about 4x to 5x can be achieved.

References

1. Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, 1990. `doi:10.1016/S0022-2836(05)80360-2`.
2. Daniel G. Brown, Ming Li, and Bin Ma. A tutorial of recent developments in the seeding of local alignment. *Journal of Bioinformatics and Computational Biology*, 02(04):819–842, 2004. `doi:10.1142/S0219720004000983`.
3. Jeremy Buhler. Efficient large-scale sequence comparison by locality-sensitive hashing. *Bioinformatics*, 17(5):419, 2001. `doi:10.1093/bioinformatics/17.5.419`.
4. Karel Břinda, Maciej Sykulski, and Gregory Kucherov. Spaced seeds improve k-mer-based metagenomic classification. *Bioinformatics*, 31(22):3584, 2015. `doi:10.1093/bioinformatics/btv419`.
5. Matteo Comin and Morris Antonello. Fast entropic profiler: An information theoretic approach for the discovery of patterns in genomes. *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, 11(3):500–509, May 2014. `doi:10.1109/TCBB.2013.2297924`.
6. Matteo Comin, Andrea Leoni, and Michele Schimd. Clustering of reads with alignment-free measures and quality values. *Algorithms for Molecular Biology*, 10(1):4, 2015. `doi:10.1186/s13015-014-0029-x`.
7. Aaron E. Darling, Todd J. Treangen, Louxin Zhang, Carla Kuiken, Xavier Messeguer, and Nicole T. Perna. Procrastination leads to efficient filtration for local multiple alignment. In Philipp Bücher and Bernard M. E. Moret, editors, *Algorithms in Bioinformatics: 6th International Workshop, WABI 2006, Zurich, Switzerland, September 11-13, 2006. Proceedings*, pages 126–137. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. `doi:10.1007/11851561_12`.
8. Samuele Girotto, Matteo Comin, and Cinzia Pizzi. Binning metagenomic reads with probabilistic sequence signatures based on spaced seeds. *To Appear*, 2017.
9. Samuele Girotto, Cinzia Pizzi, and Matteo Comin. MetaProb: accurate metagenomic reads binning based on probabilistic sequence signatures. *Bioinformatics*, 32(17):i567–i575, September 2016. `doi:10.1093/bioinformatics/btw466`.
10. Lars Hahn, Chris-André Leimeister, Rachid Ounit, Stefano Lonardi, and Burkhard Morgenstern. Rasbhari: Optimizing spaced seeds for database searching, read mapping and alignment-free sequence comparison. *PLOS Computational Biology*, 12(10):1–18, 10 2016. `doi:10.1371/journal.pcbi.1005107`.

11 Lucian Ilie, Silvana Ilie, and Anahita Mansouri Bigvand. SpEED: fast computation of sensitive spaced seeds. *Bioinformatics*, 27(17):2433, 2011. doi:10.1093/bioinformatics/btr368.

12 Uri Keich, Ming Li, Bin Ma, and John Tromp. On spaced seeds for similarity search. *Discrete Applied Mathematics*, 138(3):253–263, 2004. doi:10.1016/S0166-218X(03)00382-2.

13 Chris-Andre Leimeister, Marcus Boden, Sebastian Horwege, Sebastian Lindner, and Burkhard Morgenstern. Fast alignment-free sequence comparison using spaced-word frequencies. *Bioinformatics*, 30(14):1991, 2014. doi:10.1093/bioinformatics/btu177.

14 Stinus Lindgreen, Karen L. Adair, and Paul Gardner. An evaluation of the accuracy and speed of metagenome analysis tools. *Scientific Reports*, 6, 2016. Article No. 19233. doi:10.1038/srep19233.

15 Bin Ma and Ming Li. On the complexity of the spaced seeds. *Journal of Computer and System Sciences*, 73(7):1024–1034, 2007. Bioinformatics {III}. doi:10.1016/j.jcss.2007.03.008.

16 Bin Ma, John Tromp, and Ming Li. Patternhunter: faster and more sensitive homology search. *Bioinformatics*, 18(3):440, 2002. doi:10.1093/bioinformatics/18.3.440.

17 Hamid Mohamadi, Justin Chu, Benjamin P. Vandervalk, and Inanc Birol. ntHash: recursive nucleotide hashing. *Bioinformatics*, page btw397, July 2016. doi:10.1093/bioinformatics/btw397.

18 Taku Onodera and Tetsuo Shibuya. The gapped spectrum kernel for support vector machines. In *Proceedings of the 9th International Conference on Machine Learning and Data Mining in Pattern Recognition*, MLDM'13, pages 1–15, Berlin, Heidelberg, 2013. Springer-Verlag. doi:10.1007/978-3-642-39712-7_1.

19 Rachid Ounit and Stefano Lonardi. Higher classification sensitivity of short metagenomic reads with CLARK-S. *Bioinformatics*, 32(24):3823, 2016. doi:10.1093/bioinformatics/btw542.

20 Rachid Ounit, Steve Wanamaker, Timothy J. Close, and Stefano Lonardi. CLARK: fast and accurate classification of metagenomic and genomic sequences using discriminative k-mers. *BMC Genomics*, 16(1):1–13, 2015. doi:10.1186/s12864-015-1419-2.

21 Laxmi Parida, Cinzia Pizzi, and Simona E. Rombo. Irredundant tandem motifs. *Theoretical Computer Science*, 525:89–102, 2014. Advances in Stringology. doi:10.1016/j.tcs.2013.08.012.

22 C. Pizzi, P. Rastas, and E. Ukkonen. Finding significant matches of position weight matrices in linear time. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 8(1):69–79, Jan 2011. doi:10.1109/TCBB.2009.35.

23 Stephen M. Rumble, Phil Lacroute, Adrian V. Dalca, Marc Fiume, Arend Sidow, and Michael Brudno. Shrimp: Accurate mapping of short color-space reads. *PLOS Computational Biology*, 5(5):1–11, 05 2009. doi:10.1371/journal.pcbi.1000386.

24 Ariya Shajii, Deniz Yorukoglu, Yun William Yu, and Bonnie Berger. Fast genotyping of known snps through approximate k-mer matching. *Bioinformatics*, 32(17):i538, 2016. doi:10.1093/bioinformatics/btw460.

25 Derrick E. Wood and Steven L. Salzberg. Kraken: ultrafast metagenomic sequence classification using exact alignments. *Genome Biology*, 15:R46, 2014. doi:10.1186/gb-2014-15-3-r46.

A Appendix

Table 3 Nine spaced seeds with $W = 11$ and length 16 computed with rasbhari minimizing overlap complexity.

Q10	1011101100101110
Q11	1100111100011110
Q12	1101011100011110
Q13	1101101110111000
Q14	1110110010110110
Q15	1111001100101110
Q16	1111001101110010
Q17	1111100011010110
Q18	1111110001011100

Table 4 Nine spaced seeds with $W = 32$ and length 45 computed with rasbhari minimizing overlap complexity.

Q19	100111111111111001001011110111100111011011 0111
Q20	101001111001011111011110111111111000101011 1111
Q21	110100110101111110001111101101111111111111 10001
Q22	110101011001100111110101110011001111111111 111
Q23	110111011111111110110111110101001000000111 11111
Q24	111011100111010001101111001111110011111111 0111
Q25	111100011011010010011111111011111111110001 1111
Q26	111101001101110011101110101011101110111011 111
Q27	111101101111110001111111000101110101111011 1011