JOURNAL OF COMPUTATIONAL BIOLOGY
Volume 27, Number 2, 2020
© Mary Ann Liebert, Inc.
Pp. 223–233
DOI: 10.1089/cmb.2019.0298

Iterative Spaced Seed Hashing: Closing the Gap Between Spaced Seed Hashing and *k*-mer Hashing

ENRICO PETRUCCI,[1] LAURENT NOÉ,[2] CINZIA PIZZI,[1] and MATTEO COMIN[1]

ABSTRACT

Alignment-free classification of sequences has enabled high-throughput processing of sequencing data in many bioinformatics pipelines. Much work has been done to speed up the indexing of *k*-mers through hash-table and other data structures. These efforts have led to very fast indexes, but because they are *k*-mer based, they often lack sensitivity due to sequencing errors or polymorphisms. Spaced seeds are a special type of pattern that accounts for errors or mutations. They allow to improve the sensitivity and they are now routinely used instead of *k*-mers in many applications. The major drawback of spaced seeds is that they cannot be efficiently hashed and thus their usage increases substantially the computational time. In this article we address the problem of efficient spaced seed hashing. We propose an iterative algorithm that combines multiple spaced seed hashes by exploiting the similarity of adjacent hash values to efficiently compute the next hash. We report a series of experiments on HTS reads hashing, with several spaced seeds. Our algorithm can compute the hashing values of spaced seeds with a speedup in range of [3.5 × –7 ×], outperforming previous methods. Software and data sets are available at Iterative Spaced Seed Hashing.

Keywords: efficient hashing, *k*-mers, spaced seeds.

1. INTRODUCTION

In computational biology, sequence classification is a common task with many applications such as phylogeny reconstruction (Leimeister et al., 2014), protein classification (Onodera and Shibuya, 2013), and metagenomic (Girotto et al., 2016; Ounit and Lonardi, 2016; Marchiori and Comin, 2017). Even if sequence classification is addressable through alignment, the scale of modern data sets has stimulated the development of faster alignment-free similarity methods (Comin and Verzotto, 2014; Leimeister et al., 2014; Wood and Salzberg, 2014; Comin et al., 2015; Apostolico et al., 2016).

The most common alignment-free indexing methods are *k*-mer based. Large-scale sequence analysis often relies on cataloguing or counting consecutive *k*-mers (substring of length *k*) in DNA sequences for indexing, querying, and similarity searching. A common step is to break a reference sequence into *k*-mers and indexing them. An efficient way of implementing this operation is through the use of hash-based data structures, for example, hash tables. Then, to classify sequences are also broken into *k*-mers and queried against the hash table to check for shared *k*-mers.

[1]Department of Information Engineering, University of Padova, Padova, Italy.
[2]CRIStAL UMR9189, Universit de Lille, Lille, France.

In Ma et al. (2002) it has been shown that requiring the matches to be nonconsecutive increases the chance of finding similarities and they introduced spaced seeds. They are a modification to the standard k-mer where some positions on the k-mer are set to be or wildcard to catch the spaced matches between sequences. In spaced seeds, the matches are distributed so as to maximize the sensitivity, that is, the probability to find a local similarity.

Spaced seeds are widely used for approximate sequence matching in bioinformatics and they have been increasingly applied to improve the sensitivity and specificity of homology search algorithms (Kucherov et al., 2006; Noé and Martin, 2014). Spaced seeds are now routinely used, instead of k-mers, in many problems involving sequence comparison such as multiple sequence alignment (Darling et al., 2006), protein classification (Onodera and Shibuya, 2013), read mapping (Rumble et al., 2009), phylogeny reconstruction (Leimeister et al., 2014), and metagenome reads clustering and classification (Binda et al., 2015; Ounit and Lonardi, 2016; Girotto et al., 2017c).

In all these applications, the use of spaced seeds, as opposed to k-mers, has been reported to improve the performance in terms of sensitivity and specificity. However, the major drawback is that the computational cost increases. For example, when k-mers are replaced by spaced seeds, the metagenomic classification of reads of Clark-S (Ounit and Lonardi, 2016) not only increases the quality of classification, but it also produces a slowdown of $17\times$ with respect to the nonseed version. A similar reduction in time performance when using spaced seeds is reported also in other applications (Rumble et al., 2009; Onodera and Shibuya, 2013; Binda et al., 2015).

The main reason is that k-mers can be efficiently hashed. In fact, the hashing of a k-mer can be easily computed from the hashing of its predecessor, since they share $k-1$ symbols. For this reason, indexing all consecutive k-mers in a string can be a very efficient process. However, when using spaced seeds these observations do no longer hold. Therefore, improving the performance of spaced seed hashing algorithms would have a great impact on a wide range of bioinformatics applications. The first attempt to address this question was in the Thesis of Harris (2007), but hard coding was used to speed up a nonlinear packing. Recently, we develop an algorithm based on the indexing of small blocks of runs of matching positions that can be combined to obtain the hashing of spaced seeds (Girotto et al., 2018a). In Girotto et al. (2017a, 2018b) we proposed a more promising direction, based on spaced seed self-correlation, to reuse part of the hashes already computed. We showed how the hash at position i can be computed based on one best previous hash. Despite the improvement in terms of speedup, the number of symbols that need to be encoded to complete the hash could still be high. In this article* we solved this problem through: (1) a better way to use previous hashes, maximizing reuse; (2) an iterative algorithm that combines multiple previous hashes. In fact, our algorithm arranges multiple previous hashes to recover all $k-1$ symbols of a spaced seed, so that we only need to encode the new symbol, like with k-mer hashing.

2. METHODS: ITERATIVE SPACED SEED HASHING

2.1. Spaced seed hashing: background

A *spaced-seed* Q (or just a seed) is a string over the alphabet $\{1, 0\}$ where the 1s correspond to matching positions and 0 to nonmatching positions or wildcards, for example, 1011001. A spaced seed Q can be represented as a set of non-negative integers corresponding to the matching positions (1s) in the seed, for example, $Q=\{0, 2, 3, 6\}$, a notation introduced in Keich et al. (2004). The *weight* of a seed, denoted as $|Q|$, corresponds to the number of 1s, whereas the *length*, or span $s(Q)$, is equal to $\max(Q)+1$.

Given a string x, the positioned spaced seed $x[i+Q]$ identifies a string of length $|Q|$, where $0 \leq i \leq n-s(Q)$. The positioned spaced seed $x[i+Q]$, also called Q-gram, is defined as the string $x[i+Q]=\{x_{i+k}, k \in Q\}$.

Example 2.1. Given the seed 1011001, defined as $Q=\{0, 2, 3, 6\}$, with weight $|Q|=4$ and span $s(Q)=7$. Let us consider the string $x=AATCACTTG$.

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

*A preliminary version of this work was presented at ISBRA 2019 (Petrucci et al., 2019).

The Q-gram at position 0 of x is defined as $x[0+Q]=ATCT$. Similarly the other Q-grams are $x[1+Q]=ACAT$ and $x[2+Q]=TACG$.

In this article, for ease of discussion, we will consider as hashing function the simple encoding of a string, which is a special case of the Rabin–Karp rolling hash. Later, we will show how more advanced hashing function can be implemented at no extra cost. Let us consider a coding function from the DNA alphabet $\mathcal{A}=\{A, C, G, T\}$ to a binary codeword, $encode : \mathcal{A} \to \{0, 1\}^{\lceil log_2|\mathcal{A}|\rceil}$, where $encode(A)=00$, $encode(C)=01$, $encode(G)=10$, and $encode(T)=11$. Following the previous example, we can compute the encodings of all symbols of the Q-gram $x[0+Q]$ as follows:

$x[0+Q]$	A	T	C	T
encodings	00	11	01	11

Finally, the hashing value of the Q-gram $ATCT$ is 11011100, which is the merge of the encodings of all symbols using little-endian notation. More formally, a standard approach to compute the hashing value of a Q-gram at position i of the string x is the following function $h(x[i+Q])$:

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

where $m(k)$ is the number of matching positions that appears to the left of k. 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 shifts that we need to apply to the encoding of the kth symbols to place it into the hashing. The vector m is important for the computation of the hashing value of a Q-gram.

Example 2.2. In this example, we report an example of hashing value computation for the Q-gram $x[1+Q]$.

x	A	A	T	C	A	C	T	T	G
Q		1	0	1	1	0	0	1	
m		0	1	1	2	3	3	3	
Shifted encodings		00≪0		01≪2	00≪4			11≪6	
		$\underline{00}$							
				$\underline{01}00$					
					$\underline{00}0100$				
Hashing value								$\underline{11}000100$	

The previous example shows how the hashing value of $x(1+Q)$ can be computed through the function $h(x[1+Q])=h(ACAT)=11000100$. The hashing value of the other Q-gram can be determined with a similar procedure, that is, $h(x[2+Q])=h(TACG)=10010011$. The hashing function $h(\cdot)$ is a special case of the Rabin–Karp rolling hash. However, more advanced hashing functions can be defined in a similar way. For example, the cyclic polynomial rolling hash can be computed by replacing shifts with rotations, OR with XOR, and the function encode() with a table, where DNA characters are mapped to random integers.

In this article we want to address the following problem.

Problem 2.1. Let us consider a string $x=x_0x_1\ldots x_i\ldots x_{n-1}$, of length n, a spaced seed Q, and a hash function h that maps strings into a binary codeword. We want to compute all hashing values $\mathcal{H}(x, Q)$ for all the Q-grams of x, starting from the first position *0* of x to the last $n-s(Q)$.

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

To compute the hash of a contiguous k-mer it is possible to use the hash of its predecessor. In fact, given the hashing value at position i, the hashing for position $i+1$ can be obtained with two operations, a shift and the insertion of the encoding of the new symbol, since the two hashes share $k-1$ symbols. 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 previous example, two consecutive Q-grams, such as $x[0+Q]=ATCT$ and $x[1+Q]=ACAT$, do not necessarily have much in common. Since the hashing values are computed in order, 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. In this article we present a solution for Problem 2.1 that maximizes the reuse of previous hashes so that only one symbol needs to be encoded in the new hash, as with k-mers hashing.

2.2. *Iterative spaced seed hashing*

In the case of spaced seeds, one can reuse part of previous hashes to compute the next one; however, we need to explore not only the hash at the previous position, as with k-mers, but also the $s(Q)-1$ previous hashes. A first attempt to solve this problem was recently proposed in Girotto et al. (2018b), where the hash at position i is computed based on one best previous hash. Despite the improvement in terms of speedup with respect to the standard hashing method, the number of symbols that need to be read to complete the hash could still be high. In this article we reduced this value to just one symbol by working in two directions: (1) we devise a better way to use a previous hash, maximizing reuse (2) we propose an iterative algorithm that combines multiple 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 $C_{g,j} = \{k \in Q : k+j \in Q \wedge m(k) = m(k+j) - m(j) + m(g)\}$ as the positions in Q that after j shifts are still in Q with the propriety that k and $k+j$ positions are both in Q and they are separated by $j-g-1$ (not necessarily consecutive) ones. In other words, if we are processing the position i of x and we want to reuse the hashing value already computed at position $i-j$, $C_{g,j}$ represents the symbols, starting at position g of $h(x[i-j+Q])$, which we can keep while computing $h(x[i+Q])$.

Example 2.3. Let us consider $Q = \{0, 1, 2, 4, 6, 8, 10\}$. 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 shows how to construct $C_{0,1}$.

K		0	1	2	3	4	5	6	7	8	9	10
Q		1	1	1	0	1	0	1	0	1	0	1
$Q \ll 1$	1	1	1	0	1	0	1	0	1	0	1	
$m(k)$		0	1	2	3	3	4	4	5	5	6	6
$m(k+1) - m(1) + m(0)$	−1	0	1	2	2	3	3	4	4	5	5	
$C_{0,1}$		0	1									

The symbols at positions $C_{0,1} = \{0, 1\}$ of the hash $h(x[1+Q])$ have already been encoded in the hash $h(x[0+Q])$ and we can keep them. To complete $h(x[1+Q])$, the number of remaining symbols are $|Q| - |C_{0,1}| = 5$.

In the article (Girotto et al., 2018b) we use only the symbols in $C_{0,j}$, that is, g was always 0. As we will see in the next examples, if we are allowed to remove the first g symbols from the hash of $h(x[i-j+Q])$, we can recover more symbols to compute $h(x[i+Q])$.

Example 2.4. 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 $C_{0,2}$.

k			0	1	2	3	4	5	6	7	8	9	10
Q			1	1	1	0	1	0	1	0	1	0	1
$Q \ll 2$	1	1	1	0	1	0	1	0	1	0	1		
$m(k)$			0	1	2	3	3	4	4	5	5	6	6
$m(k+2) - m(2) + m(0)$	−2	−1	0	1	1	2	2	3	3	4	4		
$C_{0,2}$			0										

Thus, the only position that we can recover is $C_{0,2} = \{0\}$. In contrast, if we are allowed to skip the first position of the hash $h(x[0+Q])$ and consider $C_{1,2}$, instead of $C_{0,2}$, we have

K			0	1	2	3	4	5	6	7	8	9	10
Q			1	1	1	0	1	0	1	0	1	0	1
$Q \ll 2$	1	1	1	0	1	0	1	0	1				
$m(k)$			0	1	2	3	3	4	4	5	5	6	6
$m(k+2) - m(2) + m(1)$	−1	0	1	2	2	3	3	4	4	5	5		
$C_{1,2}$					2		4		6		8		

where we can reuse the symbols $C_{1,2} = \{2, 4, 6, 8\}$ of $h(x[0+Q])$ to compute $h(x[2+Q])$. This example shows how the original definition of C_j in Girotto et al. (2018b), which in this work corresponds to $C_{0,2} = \{0\}$, was not optimal and more symbols could be recovered from the same hash with $C_{1,2} = \{2, 4, 6, 8\}$.

In Girotto et al. (2018b), the hash value at a given position was reconstructed starting from the best previous hash. However, the number of symbols to be inserted to complete the hash could still be high. In this article we propose a new method that not only considers the best previous hash, but also all previous hashes at once. For a given hash to be computed h_i, we devised an iterative algorithm that is able to find a combination of the previous hashes that covers all symbols of h_i, apart from the last one. That is, we can combine multiple hashes to recover $|Q|-1$ symbols of h_i, so that we only need to read the new symbol, like with k-mer hashing.

Let us assume that we have already computed a portion of the hash h_i, and that the remaining symbols are $Q' \subset Q$. We can search the best previous hash that covers the largest number of positions of Q'. To this end, we define the function $Best\,Prev(s, Q')$ that searches for this best previous hash:

$$Best\,Prev\,(s, Q') = argmax_{z\in[0,\,s-1],\,k\in[1,\,s]}|\mathcal{C}_{z,\,k} \cap Q'|.$$

This function will return a pair (g, j) that identifies the best previous hash at position h_{i-j} from which, after removing the first g symbols, we can recover $|C_{g,j} \cap Q'|$ symbols. To extract these symbols from h_{i-j} we define a mask, $Mask_{g,j}$, that filters these positions. The algorithm iteratively searches the best previous hashes, until all $|Q|-1$ symbols have been recovered. An overview of the method is shown as follows.

Our iterative algorithm scans the input string x and computes all hashing values according to the spaced seed Q. To better understand the amount of savings we evaluate the 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 11101010101 that corresponds to $Q = \{0, 1, 2, 4, 6, 8, 10\}$. If we use the standard function $h(x[i+Q])$ to compute all hashes, each symbol of x is read $|Q| = 7$ times.

Iterative Spaced Seed Hashing

```
1: Compute $C_{g,k}$ and $Mask(g,k)$ $\forall g, k$;
2: $h_0 := $ compute $h(x[0+Q])$;
3: **for** $i := 1$ to $s(Q)-1$ **do**
4:   $Q' = Q$;
5:   **while** $|Q'| \neq 1$ **do**
6:     $(g,k) = Best\,Prev\,(i, Q')$;
7:     **if** $(Q' \cap C_{g,k}) == \emptyset$ **then**
8:       Exit while;
9:     **else**
10:      $h_i := h_i$ OR $((h_{i-k}$ AND $Mask(g,k)) >> k * log_2|\mathcal{A}|)$;
11:      $Q' = Q' - C_{g,k}$;
12:    **end if**
13:  **end while**
14:  **for all** $k \in Q'$ **do**
15:    insert $encode(x_{i+k})$ at position $m(k) * log_2|\mathcal{A}|$ of $h_i$;
16:  **end for**
17: **end for**
18: **for** $i := s(Q)$ to $|x| - s(Q)$ **do**
19:  $Q' = Q$;
20:  **while** $|Q'| \neq 1$ **do**
21:    $(g,k) = Best\,Prev(s(Q)-1, Q')$;
22:    $h_i := h_i$ OR $((h_{i-k}$ AND $Mask(g,k)) >> k * log_2|\mathcal{A}|)$;
23:    $Q' = Q' - C_{g,k}$;
24:  **end while**
25:  insert $encode(x_{i+s(Q)-1})$ at last position of $h_i$;
26: **end for**
```

In the first iteration of our algorithm (lines = 19 – 25) $Q' = Q$ and the best previous hash $Best\,Prev$ $(s(Q)-1, Q') = (1, 2)$ is $C_{1,2} = \{2, 4, 6, 8\}$. Thus, while computing h_i we can recover these four symbols from h_{i-2}. At the end of the first iteration Q' is updated to $\{0, 1, 10\}$. During the second iteration the best previous hash $Best\,Prev\,(s(Q)-1, Q') = (0, 1)$ is $C_{0,1} = \{0, 1\}$. As earlier, we can append these two symbols from h_{i-1} to the hash h_i. Now, we have that $Q' = \{10\}$, that is, only one symbol is left. The last symbol is read and encoded into h_i, and the hash is complete. In summary, after two iterations all $|Q|-1$ symbols of h_i have been encoded into the hash, and we only need to read one new symbol from the sequence. Moreover, if one needs to scan a string with a spaced seed and to compute all hashing values, the aforementioned algorithm guarantees to minimize the number of symbols to read. In fact, with our algorithm, we can compute all hashing values while reading each symbol of the input string only once, as with *k*-mers.

3. RESULTS AND DISCUSSION

In this section we will present the results of some experiments in which Iterative Spaced Seed Hashing (ISSH) is compared against two other approaches available in literature: Fast Indexing for Spaced seed Hashing (FISH) (Girotto et al., 2018a) (block based) and Fast Spaced seed Hashing (FSH) (Girotto et al., 2018b) (overlap based).

3.1. Experimental settings

We use the same settings as in previous studies (Girotto et al., 2018a,b). The spaced seeds belong to three different types of spaced seeds, according to the objective function used to generate them: maximizing the hit probability (Ounit and Lonardi, 2016), minimizing the overlap complexity (Hahn et al., 2016), and maximizing the sensitivity (Hahn et al., 2016). We tested three spaced seeds for each type, all with weight $W=22$ and length $L=31$. This list of spaced seeds is presented in Table 1 with labels from $Q1$ to $Q9$. Furthermore, we used other sets of spaced seeds, built with *rashbari* (Hahn et al., 2016), which have weights from 11 to 32 and the same length. The complete list of the spaced seeds used is reported in Appendix Tables A2–A4. The data sets of metagenomic reads to be hashed were taken from previous articles on binning and classification (Wood and Salzberg, 2014; Girotto et al., 2016, 2017b). Details about the reads data sets are shown in Appendix Table A1. All the experiments have been performed on a laptop equipped with an Intel i7-3537U CPU at 2 GHz and 8 GB of RAM.

3.2. Analysis of the time performances

The first comparison we present is between the performances of ISSH, FISH, and FSH in terms of speedup with respect to the standard hash computation [i.e., applying Eq. (1) to each position]. Figure 1 shows the average speedup among all data sets, for each of the spaced seeds $Q1-Q9$, obtained by the three different methods.

It can be seen that ISSH is much faster than both FISH and FSH for all the spaced seeds. In terms of actual running time, the standard approach [Eq. (1)] requires about 14 minutes to compute the hashes for a single spaced seed on all data sets. ISSH takes just >2 minutes with an average speedup of 6.2. As for the other two approaches, FISH and FSH, they compute the hashes in 6 and 9 minutes, respectively, with an average speedup of 2 (FISH) and 1.5 (FSH).

We also notice that the variation among the speedups, relative to different spaced seeds using the same method, is lower for ISSH, for which the speedups are in the range [6.05–6.25], whereas for FISH and FSH the range is [1.89–2.16] and [1.18–1.58], respectively. For all the tested methods there is a correlation

TABLE 1. THE SPACED SEEDS $Q1-Q9$ DEPLOYED IN THE EXPERIMENTS GROUPED BY THEIR TYPE

Spaced seeds maximizing the hit probability (Ounit and Lonardi, 2016)	
$Q1$	1111011101110010111001011011111
$Q2$	1111101011100101101110011011111
$Q3$	1111101001110101101100111011111
Spaced seeds minimizing the overlap complexity (Hahn et al., 2016)	
$Q4$	1111010111010011001110111110111
$Q5$	1110111011101111010010110011111
$Q6$	1111101001011100111110101101111
Spaced seeds maximizing the sensitivity (Hahn et al., 2016)	
$Q7$	1111011110011010111110101011011
$Q8$	1110101011101100110100111111111
$Q9$	1111110101101011100111011001111

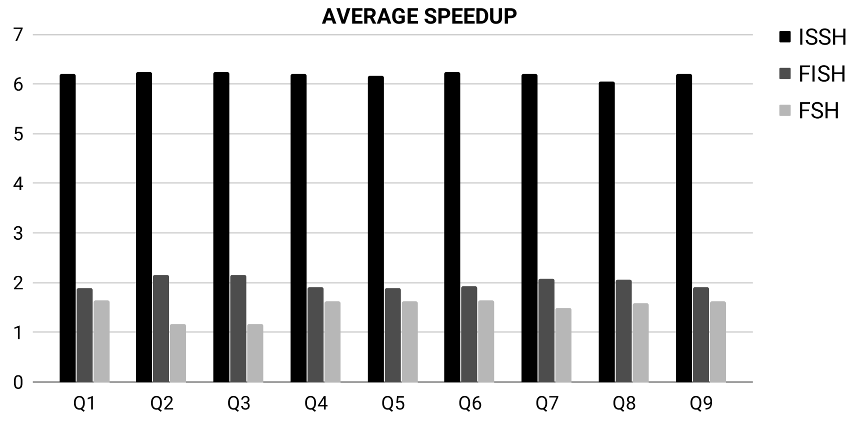

FIG. 1. The average speedup obtained by ISSH, FISH, and FSH with respect to the standard computation. FISH, Fast Indexing for Spaced seed Hashing; FSH, Fast Spaced seed Hashing; ISSH, Iterative Spaced seed Hashing.

between the spaced seed structure and the time needed for the computation. FISH depends on the number of blocks of 1s, whereas both ISSH and FSH depend on the spaced seed self-correlation. ISSH performances are also sensitive to the number of iterations. However, the experiments show that, even if FSH performs a single iteration, the time required to naively compute the hash for all the nonoverlapping positions is more than the time required by ISSH to perform more iterations. Moreover, for all the tested spaced seeds the number of iterations needed by ISSH was on average 4.

Figure 2 gives an insight on the performance of ISSH with respect to each spaced seed and each data sets considered.

First of all, we notice that the performances are basically independent on the spaced seed used. Next, for what concerns the data sets characteristics, it can be observed that the speedup increases with the reads length, reaching the highest values for the data sets R7, R8, and R9, which have the longest reads. This behavior is expected: when considering longer reads the slowdown caused by the initial transient—in which more than one symbol has to be encoded—is less relevant with respect to the total running time.

In Figure 3 we report the speedups on each data sets obtained by $Q7$, a typical spaced seed (the other spaced seeds performances are similar) using ISSH, FISH, and FSH.

All the results are compatible with the previous observations: ISSH, if compared with FISH and FSH, allows to compute the hashing values faster for all the data sets. Furthermore, by using ISSH, the improvement on long reads data sets is larger than the improvement obtained with FISH or FSH.

3.3. Effect of spaced seeds weight on time performances

The experiments presented in this article point out the connection between the density of a spaced seed and the speedup. We considered four sets of nine spaced seeds, generated with *rasbhari* (Hahn et al., 2016), with weights 14, 18, 22, and 26 and a fixed length of 31 (see in the Appendix Tables A2–A4).

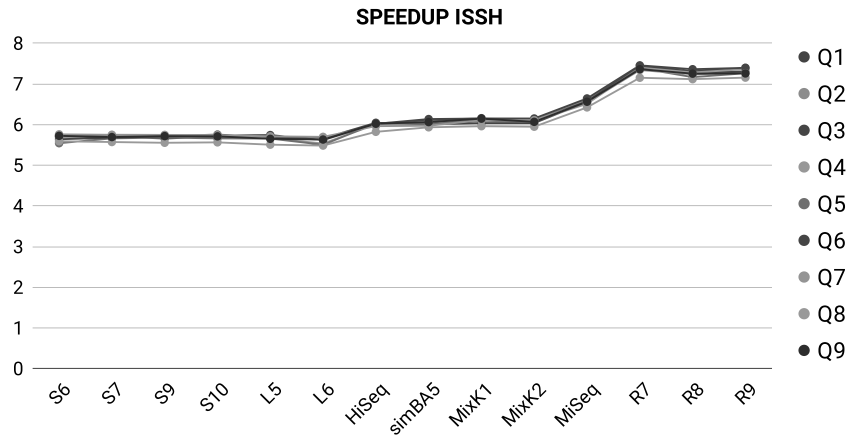

FIG. 2. Speedup of ISSH of all the single spaced seeds for each of the considered data sets, ordered by reads length.

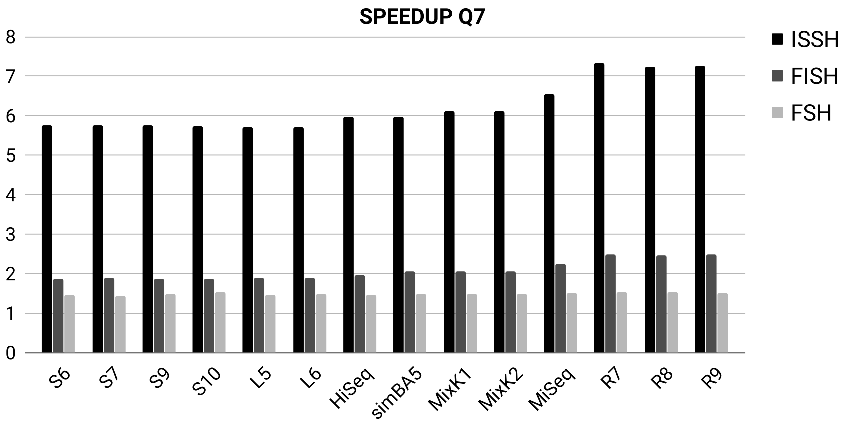

FIG. 3. Details of the speedup on the spaced seed $Q7$ on each data sets, ordered by reads length, using ISSH, FISH, and FSH.

In Figure 4 we compare the average speedup of ISSH, FISH, and FSH for these sets of spaced seeds as a function of the weight W. We notice that the speedup grows as the weight increases. This phenomenon is consistent among all the methods we analyzed. It is reasonable to think that such difference is due to how the hashes are computed with the standard method using Eq. (1) (against which all methods are compared), because denser spaced seeds imply hashes with a larger number of symbols that need to be encoded and joined together. Moreover, for ISSH we have that denser spaced seeds have more chances of needing fewer previously calculated hashes to compute each of the $|Q|-1$ symbols, thus saving further iterations.

Both these effects are emphasized when looking at the actual running times needed by the least dense group ($W=14$) and by the most dense group ($W=26$) of spaced seeds. The standard method requires 9.73 and 15.11 minutes, respectively, whereas ISSH spends only 2.75 and 2.16 minutes to perform the same task.

3.4. *Effect of number of iterations on time performances*

The experiments described in this section have been essential to understand how many of the previously calculated hashes we should use to further speed up the computations. At the beginning we were not sure whether it was better or not to increase the number of iterations, for each of the $|x|-s(Q)$ hashes that need to be computed, to recover only few symbols. To address this problem we considered the speedup obtained with ISSH by progressively limiting the number of iterations, which is the number of previous hashes used to compute a new one, from one to five (number needed to cover all the $|Q|-1$ symbols for all the spaced seeds considered). The results are shown in Figure 5 where the spaced seeds used have length $L=31$ and weight $W=14$.

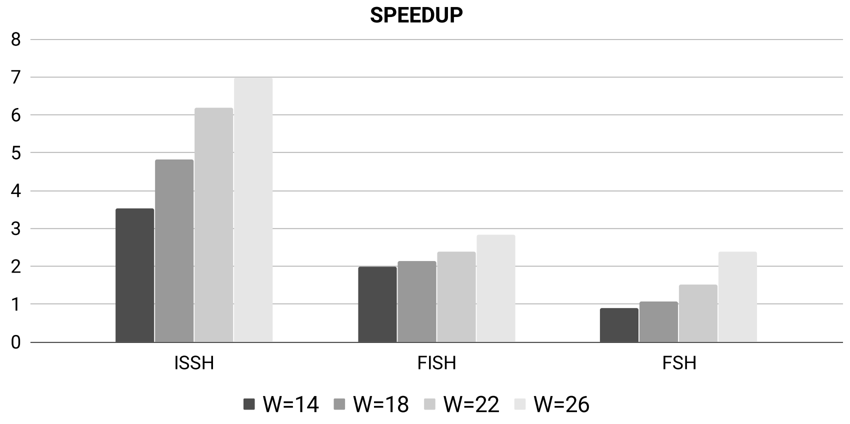

FIG. 4. The speedup of ISSH, FISH, and FSH as a function of the spaced seeds density ($L=31$ and $W=14$, 18, 22, and 26).

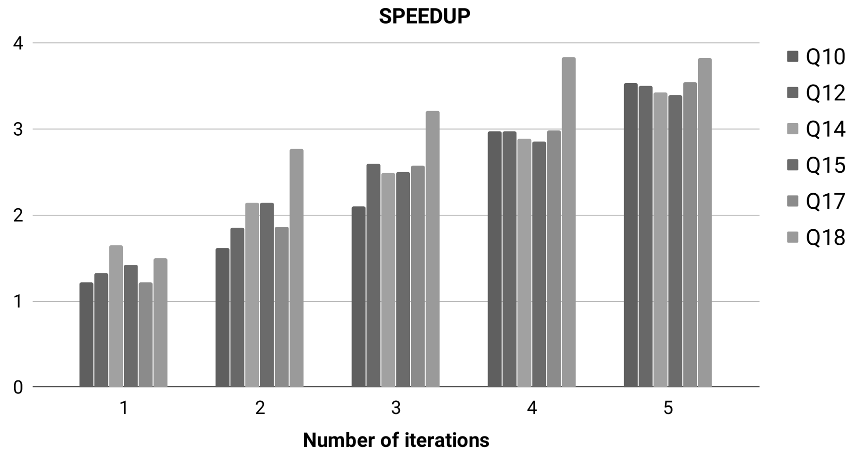

FIG. 5. Speedup of ISSH, on the seeds of Appendix Table A2, with an increasing limit to the number of iterations used to compute each hash.

To compute the remaining symbols it has been used a similar approach to the one described in Girotto et al. (2018b) for FSH, which is also used by ISSH to complete the hashes computed in the transient part. We can see that gradually increasing the number of iterations the speedup becomes greater: from speedups similar to the ones obtained with FSH, which only considers one single previous hash (that correspond to a single ISSH iteration), to the higher speedups of this new approach.

In this article we can also observe again that the number of iterations has an impact on the speedup. For the spaced seed $Q18$ only four iterations are required to recover all the $|Q|-1$ positions, one less than the other spaced seeds, and this leads to a higher speedup for $Q18$ if compared with the other spaced seeds with the same density.

In Figure 6 the same speedups shown in Figure 5 are plotted, but this time against the number of symbols recovered.

Even in this case, it is clear that as ISSH recovers more symbols—which means having to apply the encode function for fewer positions—the speedup increases. Another interesting observation is that a small variation is present between the speedup of computations in which the same number of symbols is recovered, but the number of iterations used is different. For example, if we consider the cases in which 10 symbols are recovered, the speedups are almost the same, but for $Q14$ and $Q15$ two iterations are needed, whereas for $Q10$ an additional one is required. In summary, the speedup depends more on the number of symbols recovered rather than the number of iterations.

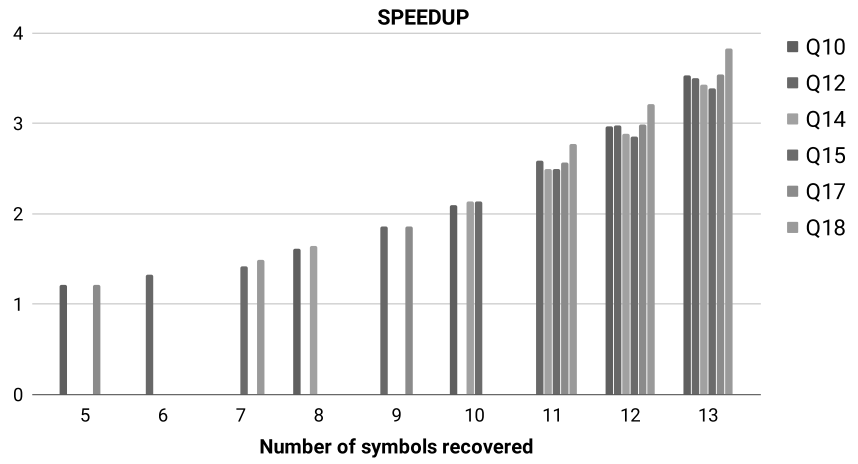

FIG. 6. Speedup of ISSH obtained recovering a certain number of symbols for the seeds of Appendix Table A2.

4. CONCLUSIONS

In this article we present ISSH, an iterative algorithm that combines multiple previous hashes to maximize the reuse of already computed hash values. The average speedup of ISSH with respect to the standard computation of hash values is in range of [3.5×–7×], depending on spaced seed density and reads length. In all experiments ISSH outperforms previously proposed algorithms. Possible directions of research are the combination of multiple spaced seeds and the investigation of global optimization schemes.

AUTHOR DISCLOSURE STATEMENT

The authors declare they have no competing financial interests.

FUNDING INFORMATION

There was no funding received for this article.

REFERENCES

Apostolico, A., Guerra, C., Landau, G.M., et al. 2016. Sequence similarity measures based on bounded hamming distance. *Theor. Comput. Sci.* 638, 76–90.

Binda, K., Sykulski, M., and Kucherov, G. 2015. Spaced seeds improve k-mer-based metagenomic classification. *Bioinformatics* 31, 3584.

Comin, M., Leoni, A., and Schimd, M. 2015. Clustering of reads with alignment-free measures and quality values. *Algorithms Mol. Biol.* 10, 4.

Comin, M., and Verzotto, D. 2014. Beyond fixed-resolution alignment-free measures for mammalian enhancers sequence comparison. *IEEE/ACM Trans. Comput. Biol. Bioinf.* 11, 628–637.

Darling, A.E., Treangen, T.J., Zhang, L., et al. 2006. *Procrastination Leads to Efficient Filtration for Local Multiple Alignment*. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 126–137.

Girotto, S., Comin, M., and Pizzi, C. 2017a. Fast spaced seed hashing. Presented at the Proceedings of the 17th Workshop on Algorithms in Bioinformatics (WABI), volume 88, Leibniz International Proceedings in Informatics, pp. 7:1–7:14. Dagstuhl Publishing, Germany.

Girotto, S., Comin, M., and Pizzi, C. 2017b. Higher recall in metagenomic sequence classification exploiting overlapping reads. *BMC Genomics* 18, 917.

Girotto, S., Comin, M., and Pizzi, C. 2017c. Metagenomic reads binning with spaced seeds. *Theor. Comput. Sci.* 698, 88–99.

Girotto, S., Comin, M., and Pizzi, C. 2018a. Efficient computation of spaced seed hashing with block indexing. *BMC Bioinf.* 19, 441.

Girotto, S., Comin, M., and Pizzi, C. 2018b. FSH: Fast spaced seed hashing exploiting adjacent hashes. *Algorithms Mol Biol.* 13, 8.

Girotto, S., Pizzi, C., and Comin, M. 2016. MetaProb: Accurate metagenomic reads binning based on probabilistic sequence signatures. *Bioinformatics* 32, i567–i575.

Hahn, L., Leimeister, C.-A., Ounit, R., et al. 2016. rasbhari: Optimizing spaced seeds for database searching, read mapping and alignment-free sequence comparison. *PLOS Comput Biol.* 12, 1–18.

Harris, R.S. 2007. Improved pairwise alignment of genomic DNA [PhD thesis]. University Park, PA.

Keich, U., Li, M., Ma, B., et al. 2004. On spaced seeds for similarity search. *Discrete Appl. Math.* 138, 253–263.

Kucherov, G., Noé, L., and Roytberg, M.A. 2006. A unifying framework for seed sensitivity and its application to subset seeds. *J. Bioinf. Comput. Biol.* 4, 553–569.

Leimeister, C.-A., Boden, M., Horwege, S., et al. 2014. Fast alignment-free sequence comparison using spaced-word frequencies. *Bioinformatics* 30, 1991.

Ma, B., Tromp, J., and Li, M. 2002. Patternhunter: Faster and more sensitive homology search. *Bioinformatics* 18, 440.

Marchiori, D., and Comin, M. 2017. Skraken: Fast and sensitive classification of short metagenomic reads based on filtering uninformative k-mers. Presented at Proceedings of the 10th International Joint Conference on Biomedical Engineering Systems and Technologies—Volume 3: BIOINFORMATICS, (BIOSTEC 2017). INSTICC, SciTePress, Portugal, pp. 59–67.

Noé, L., and Martin, D.E.K. 2014. A coverage criterion for spaced seeds and its applications to support vector machine string kernels and k-mer distances. *J. Comput. Biol.* 21, 947–963.

Onodera, T., and Shibuya, T. 2013. The gapped spectrum kernel for support vector machines. Presented at Proceedings of the 9th Conference on Machine Learning and Data Mining in Pattern Recognition, MLDM'13. Springer-Verlag, Germany, pp. 1–15.

Ounit, R., and Lonardi, S. 2016. Higher classification sensitivity of short metagenomic reads with clark-s. *Bioinformatics* 32, 3823.

Petrucci, E., Noé, L., Pizzi, C., et al. 2019. Iterative spaced seed hashing: Closing the gap between spaced seed hashing and k-mer hashing, 208–219. *In* Cai, Z., Skums, P., and Li, M., eds. *Bioinformatics Research and Applications*. Springer International Publishing, Cham.

Rumble, S.M., Lacroute, P., Dalca, A.V., et al. 2009. Shrimp: Accurate mapping of short color-space reads. *PLoS Comput. Biol.* 5, 1–11.

Wood, D.E., and Salzberg, S.L. 2014. Kraken: Ultrafast metagenomic sequence classification using exact alignments. *Genome Biol.* 15, R46.

Address correspondence to:
Prof. Cinzia Pizzi and Prof. Matteo Comin
Department of Information Engineering
University of Padova
Via Gradenigo 6/A
Padova 35131
Italy

E-mail: pizzi@dei.unipd.it or comin@dei.unipd.it