Iterative Spaced Seed Hashing: Closing the gap between spaced seed hashing and k-mer hashing

E.Petrucci[1], L.Noé[2], C.Pizzi[1], and M.Comin[1]

[1] Department of Information Engineering, University of Padova, Padova, Italy
{cinzia.pizzi, comin}@dei.unipd.it
[2] CRIStAL UMR9189, Université de Lille, France

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 paper 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 in order 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 of 6.2x, outperforming previous methods. Software and Datasets are available at ISSH

Keywords: k-mers · spaced seeds · gapped q-gram · efficient hashing

1 Introduction

In computational biology, sequence classification is a common task with many applications such as phylogeny reconstruction [16], protein classification [20], metagenomic [11,18,21]. Even if sequence classification is addressable via alignment, the scale of modern datasets has stimulated the development of faster alignment-free similarity methods [1,3,4,16,23].

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, e.g. hash tables. Then, to classify sequences are also broken into k-mers and queried against the hash table to check for shared k-mers.

In [17] it has been shown that requiring the matches to be non-consecutive 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 “don’t care” 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 [15,19]. Spaced seeds are now routinely used, instead of k-mers, in many problems involving sequence comparison like: multiple sequence alignment [5], protein classification [20], read mapping [22], phylogeny reconstruction [16], metagenome reads clustering and classification [2,8,21].

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 [21] increases the quality of classification, but it also produces a slowdown of 17x with respect to the non-seed version. A similar reduction in time performance when using spaced seeds is reported also in other applications [2,20,22].

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 not 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 R. Harris [13], but hard coding was used to speed-up a non linear 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 [9]. In [6,10] we proposed a more promising direction, based on spaced seed self-correlation, in order 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 in order to complete the hash could still be high. In this paper we solved this problem through: 1) a better way to use previous hashes, maximizing re-use ; 2) an iterative algorithm that combines multiple previous hashes. In fact, our algorithm arranges multiple previous hashes in order 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 non-matching positions or wildcards, e.g. 1011001. A spaced seed Q can be represented as a set of non negative integers corresponding to the matching positions (1s) in the seed, e.g. $Q = \{0, 2, 3, 6\}$, a notation introduced in [14]. The *weight* of a seed, denoted as $|Q|$, corresponds to the number of 1s, while 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 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		

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 paper, for ease of discussion, we will consider as hashing function the simple encoding of a string, that is a special case of the Rabin-Karp rolling hash. Later, we will shown how more advanced hashing function can be implemented at no extra cost. Let's consider a coding function from the DNA alphabet $\mathcal{A} = \{A, C, G, T\}$ to a binary codeword, $encode : \mathcal{A} \rightarrow \{0, 1\}^{log_2|\mathcal{A}|}$, where $encode(A) = 00, encode(C) = 01, encode(G) = 10$, and $encode(T) = 11$. Following the above 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, that 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 k-th symbols in order to place it into the hashing. The vector m is important for the computation of the hashing value of a Q-gram.

Example 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 above 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, i.e. $h(x[2+Q]) = h(TACG) = 10010011$. 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$(\cdot)$ with a table, where DNA characters are mapped to random integers.

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 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 $n - s(Q)$.

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

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 above example, two consecutive Q-grams, like $x[0+Q] = ATCT$ 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 paper we present a solution for Problem 1 that maximizes the re-use 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 the $s(Q)-1$ previous hashes. A first attempt to solve this problem was recently proposed in [10], 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 in order to complete the hash could still be high. In this paper 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 re-use 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 $h(x[i-j+Q])$, that we can keep while computing $h(x[i+Q])$.

Example 3. Let's 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 show 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≪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. In order to complete $h(x[1+Q])$, the number of remaining symbols are $|Q| - |C_{0,1}| = 5$.

In the paper [10] 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 in order to compute $h(x[i+Q])$.

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 $C_{0,2}$.

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≪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 $C_{0,2} = \{0\}$. On the other hand, 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≪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(1)	-1	0	1	2	2	3	3	4	4	5	5		
$C_{1,2}$					2		4		6		8		

Where, we can re-use the symbols $C_{1,2} = \{2, 4, 6, 8\}$ of $h(x[0 + Q])$ in order to compute $h(x[2 + Q])$. This example shows how the original definition of C_j in [10], that 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 [10], 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 paper we propose a new method that not only consider the best previous hash, but 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 in order to recover $|Q| - 1$ symbols of h_i, so that we only need to read the new symbol, like with k-mer hashing.

Let's 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 $BestPrev(s, Q')$ that searches for this best previous hash:

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

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. In order 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 below:

Our iterative algorithm scans the input string x and computes all hashing values according to the spaced seed Q. In order 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.

Algorithm 1 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:         $(g,k) = BestPrev(i, Q')$;
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:        $(g,k) = BestPrev(s(Q)-1, Q')$;
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 $BestPrev(s(Q)-1, Q') = (1,2)$ is $C_{1,2} = \{2,4,6,8\}$. Thus, while computing h_i we can recover these 4 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 $BestPrev(s(Q)-1, Q') = (0,1)$ is $C_{0,1} = \{0,1\}$. As above, 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 above 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 ISSH is compared against two other approaches available in literature: FISH [9] (block-based) and FSH [10] (overlap-based).

3.1 Experimental settings

We use the same settings as in previous studies [9,10]. The spaced seeds belong to three different types of spaced seeds, according to the objective function used to generate them: maximizing the hit probability [21]; minimizing the overlap complexity [12]; and maximizing the sensitivity [12]. We tested three spaced seeds for each type, all with weight $W = 22$ and length $L = 31$ (see Appendix of [9]). Furthermore, we used other sets of spaced seeds, built with *rashbari* [12], which have weights from 11 to 32 and the same length. The complete list of the spaced seeds used is reported in the Appendix of [9].The datasets of metagenomic reads to be hashed were taken from previous papers on binning and classification [11,7,23]. 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 datasets, for each of the spaced seeds $Q1$-$Q9$, obtained by the three different methods.

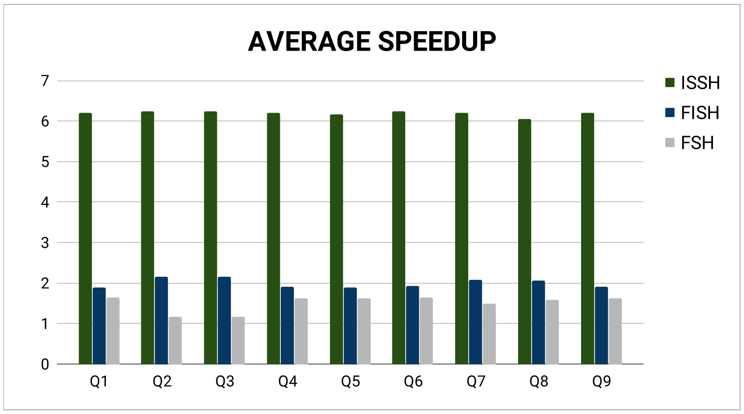

Fig. 1. The average speedup obtaind by ISSH, FISH and FSH with respect to the standard computation.

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 datasets. ISSH takes just over 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, are lower for ISSH, for which the speedups are in the range [6.05-6.25] while 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 between the spaced seed structure and the time needed for the computation. FISH depends on the number of blocks of 1s, while 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 non-overlapping 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 datasets considered.

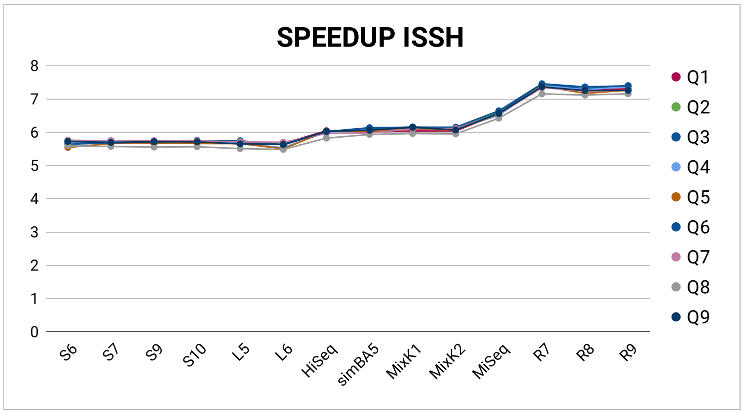

Fig. 2. Speedup of ISSH of all the single spaced seeds for each of the considered datasets, ordered by reads length.

First of all, we notice that the performances are basically independent on the spaced seed used. Next, for what concerns the datasets characteristics, it can be observed that the speedup increases with the reads length, reaching the highest values for the datasets 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 datasets obtained by $Q7$, a typical spaced seed (the other spaced seeds performances are similar) using ISSH, FISH and FSH.

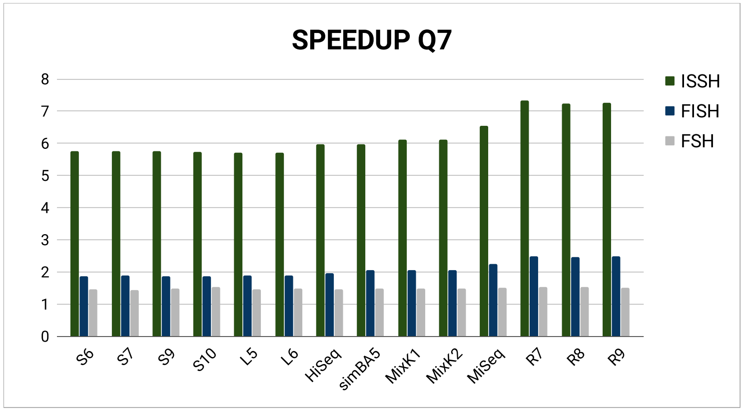

Fig. 3. Details of the speedup on the spaced seed $Q7$ on each datasets, ordered by reads length, using ISSH, FISH and FSH.

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

3.3 Effect of Spaced Seeds Weight on Time Performances

The experiments presented here 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* [12], with weights 14, 18, 22 and 26 and a fixed length of 31.

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.

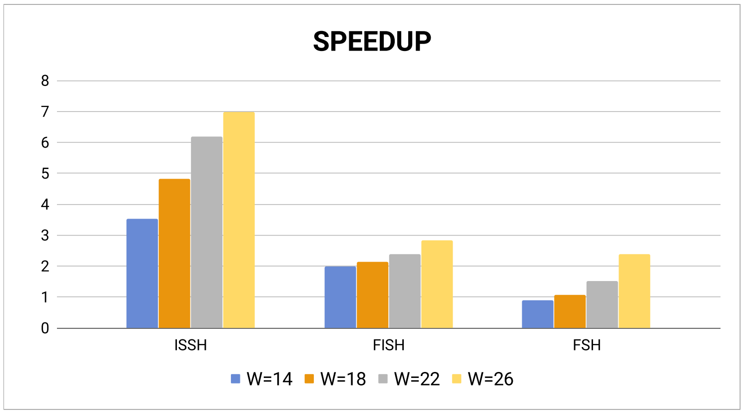

Fig. 4. The speedup of ISSH, FISH and FSH as a function of the spaced seeds density (L=31 and W=14, 18, 22, and 26).

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, while ISSH spends only 2.75 and 2.16 minutes to perform the same task.

4 Conclusions

In this paper we present ISSH (Iterative Spaced Seed Hashing), an iterative algorithm that combines multiple previous hashes in order to maximize the re-use of already computed hash values. The average speedup of ISSH with respect to the standard computation of hash values is in range of [3.5x- 7x], 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.

References

1. Apostolico, A., Guerra, C., Landau, G.M., Pizzi, C.: Sequence similarity measures based on bounded hamming distance. Theoretical Computer Science **638**, 76 – 90 (2016)
2. Břinda, K., Sykulski, M., Kucherov, G.: Spaced seeds improve k-mer-based metagenomic classification. Bioinformatics **31**(22), 3584 (2015)
3. Comin, M., Verzotto, D.: Beyond fixed-resolution alignment-free measures for mammalian enhancers sequence comparison. IEEE/ACM Transactions on Computational Biology and Bioinformatics **11**(4), 628–637 (July 2014)
4. Comin, M., Leoni, A., Schimd, M.: Clustering of reads with alignment-free measures and quality values. Algorithms for Molecular Biology **10**(1), 4 (2015)

5. Darling, A.E., Treangen, T.J., Zhang, L., Kuiken, C., Messeguer, X., Perna, N.T.: Procrastination Leads to Efficient Filtration for Local Multiple Alignment, pp. 126–137. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)
6. Girotto, S., Comin, M., Pizzi, C.: Fast spaced seed hashing. In: Proceedings of the 17th Workshop on Algorithms in Bioinformatics (WABI). Leibniz International Proceedings in Informatics, vol. 88, pp. 7:1–7:14 (2017)
7. Girotto, S., Comin, M., Pizzi, C.: Higher recall in metagenomic sequence classification exploiting overlapping reads. BMC Genomics **18**(10), 917 (Dec 2017)
8. Girotto, S., Comin, M., Pizzi, C.: Metagenomic reads binning with spaced seeds. Theoretical Computer Science **698**, 88–99 (October 2017)
9. Girotto, S., Comin, M., Pizzi, C.: Efficient computation of spaced seed hashing with block indexing. BMC Bioinformatics **19**(15), 441 (Nov 2018)
10. Girotto, S., Comin, M., Pizzi, C.: Fsh: fast spaced seed hashing exploiting adjacent hashes. Algorithms for Molecular Biology **13**(1), 8 (Mar 2018)
11. Girotto, S., Pizzi, C., Comin, M.: MetaProb: accurate metagenomic reads binning based on probabilistic sequence signatures. Bioinformatics **32**(17), i567–i575 (Sep 2016)
12. Hahn, L., Leimeister, C.A., Ounit, R., Lonardi, S., Morgenstern, B.: Rasbhari: Optimizing spaced seeds for database searching, read mapping and alignment-free sequence comparison. PLOS Computational Biology **12**(10), 1–18 (10 2016)
13. Harris, R.S.: Improved Pairwise Alignment of Genomic Dna. Ph.D. thesis, University Park, PA, USA (2007)
14. Keich, U., Li, M., Ma, B., Tromp, J.: On spaced seeds for similarity search. Discrete Applied Mathematics **138**(3), 253 – 263 (2004)
15. Kucherov, G., Noé, L., Roytberg, M.A.: A unifying framework for seed sensitivity and its application to subset seeds. Journal of Bioinformatics and Computational Biology **4**(2), 553–569 (November 2006)
16. Leimeister, C.A., Boden, M., Horwege, S., Lindner, S., Morgenstern, B.: Fast alignment-free sequence comparison using spaced-word frequencies. Bioinformatics **30**(14), 1991 (2014)
17. Ma, B., Tromp, J., Li, M.: Patternhunter: faster and more sensitive homology search. Bioinformatics **18**(3), 440 (2002)
18. Marchiori, D., Comin, M.: Skraken: Fast and sensitive classification of short metagenomic reads based on filtering uninformative k-mers. In: Proceedings of the 10th International Joint Conference on Biomedical Engineering Systems and Technologies - Volume 3: BIOINFORMATICS, (BIOSTEC 2017). pp. 59–67. INSTICC, SciTePress (2017)
19. Noé, L., Martin, D.E.K.: A coverage criterion for spaced seeds and its applications to support vector machine string kernels and k-mer distances. Journal of Computational Biology **21**(12), 947–963 (December 2014)
20. Onodera, T., Shibuya, T.: The gapped spectrum kernel for support vector machines. In: Proceedings of the 9th Conference on Machine Learning and Data Mining in Pattern Recognition. pp. 1–15. MLDM'13, Springer-Verlag (2013)
21. Ounit, R., Lonardi, S.: Higher classification sensitivity of short metagenomic reads with clark-s. Bioinformatics **32**(24), 3823 (2016)
22. Rumble, S.M., Lacroute, P., Dalca, A.V., Fiume, M., Sidow, A., Brudno, M.: Shrimp: Accurate mapping of short color-space reads. PLOS Computational Biology **5**(5), 1–11 (05 2009)
23. Wood, D.E., Salzberg, S.L.: Kraken: ultrafast metagenomic sequence classification using exact alignments. Genome Biology **15**, R46 (2014)