On Weighted K-Mer Dictionaries

Giulio Ermanno Pibiri
ISTI-CNR, Pisa, Italy

Abstract

We consider the problem of representing a set of k-mers and their abundance counts, or weights, in compressed space so that assessing membership and retrieving the weight of a k-mer is efficient. The representation is called a *weighted dictionary* of k-mers and finds application in numerous tasks in Bioinformatics that usually count k-mers as a pre-processing step. In fact, k-mer counting tools produce very large outputs that may result in a severe bottleneck for subsequent processing.

In this work we extend the recently introduced SSHash dictionary (Pibiri, *Bioinformatics* 2022) to also store compactly the weights of the k-mers. From a technical perspective, we exploit the order of the k-mers represented in SSHash to encode *runs* of weights, hence allowing (several times) better compression than the empirical entropy of the weights. We also study the problem of reducing the number of runs in the weights to improve compression even further and illustrate a lower bound for this problem. We propose an efficient, greedy, algorithm to reduce the number of runs and show empirically that it performs well, i.e., very similarly to the lower bound. Lastly, we corroborate our findings with experiments on real-world datasets and comparison with competitive alternatives. Up to date, SSHash is the only k-mer dictionary that is exact, weighted, associative, fast, and small.

2012 ACM Subject Classification Applied computing → Bioinformatics

Keywords and phrases K-Mers, Weights, Compression, Hashing

Supplementary Material Source code: `https://github.com/jermp/sshash`.

Funding This work was partially supported by the projects: MobiDataLab (EU H2020 RIA, grant agreement Nº101006879) and OK-INSAID (MIUR-PON 2018, grant agreement NºARS01_00917).

1 Introduction

Recent advancements in the so-called Next Generation Sequencing (NGS) technology made possible the availability of very large collections of DNA. However, before being able to actually analyze the data at this scale, efficient methods are required to index and search such collections. One popular strategy to address this challenge is to consider short substrings of fixed length k, known as k-mers. Software tools based on k-mers are predominant in Bioinformatics and they have found applications in genome assembly [3, 13], variant calling [15, 37], pan-genome analysis [2, 18], meta-genomics [38], sequence comparison [31, 33, 34], just to name a few but noticeable ones.

For several such applications it is important to quantify how many times a given k-mer is present in a DNA database. In fact, many efficient k-mer counting tools have been developed for this task [8, 21, 16, 30]. The output of these tools is a table where each distinct k-mer in the database is associated to its abundance count, or *weight*. The weights are either exact or approximate. (In this work, we focus on exact weights.) These genomic tables are usually very large and take several GBs – in the range of 40-80 bits/k-mer or more according to recent experiments [12, 19, 21]. Therefore, the tables should be compressed effectively while permitting efficient random access queries in order to be useful for on-line processing tasks. This is precisely the goal of this work. We better formalize the problem as follows.

Let $\mathcal{K}$ be the set of the n distinct k-mers extracted from a given DNA string. In particular, $\mathcal{K}$ can be regarded as a set of n pairs $\langle g, w(g)\rangle$, where g is a k-mer and $w(g)$ is the *weight* of g. Our objective is to build a compressed, weighted, dictionary for $\mathcal{K}$, i.e., a data structure representing the k-mers *and* weights of $\mathcal{K}$ in compressed space such that it is efficient to check the *exact* membership of g to $\mathcal{K}$ and, if g actually belongs to $\mathcal{K}$, retrieve $w(g)$.

In our previous investigation on the problem, we proposed a *sparse and skew hashing* scheme for k-mers (SSHash, henceforth) [22] – a compressed dictionary that relies on k-mer *minimizers* [31] and *minimal perfect hashing* [23, 24] to support fast membership (in both random and streaming query modality) in succinct space. However, we did not consider the weights of the k-mers. In this work, therefore, we enrich the SSHash data structure with the weight information. The main practical result is that, by exploiting the *order* of the k-mers represented in SSHash, the compressed exact weights take only a small extra space on top of the space of SSHash. This extra space is proportional to the number of *runs* (maximal sub-sequences formed by all equal symbols) in the weights and not proportional to the number of distinct k-mers. As a consequence, the weights are represented in a much smaller space than the empirical entropy lower bound.

We also study the problem of reducing the number of runs in the weights and model it as a *graph covering* problem, for which we give an expected linear-time algorithm. The optimization algorithm effectively reduces the number of runs, hence improving space even further, and almost matches the best possible reduction according to a lower bound.

When empirically compared to other weighted dictionaries that can be either somewhat smaller but much slower or much larger, SSHash embodies a robust trade-off between index space and query efficiency.

2 Related Work

A solution to the weighted k-mer dictionary problem can be obtained using the popular FM-index [11]. The FM-index represents the original DNA string taking the Burrows-Wheeler transform (BWT) [5] of the string. Reporting the weight of a k-mer is solved using the *count* operation of the FM-index which involves $O(k)$ rank queries over the BWT.

Another solution using the BWT is the so-called BOSS data structure [4] that is a succinct representation of the *de Bruijn* graph of the input – a graph where the nodes are the k-mers and the edges model the overlaps between the k-mers. The BOSS data structure has been recently enriched with the weights of the k-mers [12], by delta-encoding the weights on a spanning branching of the graph. Since consecutive k-mers often have equal (or very similar) weights, good space effectiveness is achieved by this technique.

Other solutions, instead, rely on hashing for faster query evaluation compared to BWT-based indexes. For example, both deBGR [19] and Squeakr [21] uses a *counting quotient filter* [20] to store the k-mers and the weights. They can either return approximate weights, i.e., wrong answers with a prescribed (low) probability, for better space usage of exact weights at the price of more index space. In any case, the memory consumption of these solutions is not competitive with that of BWT-based ones as they do not employ sophisticated compression techniques and were designed for other purposes, e.g., dynamic updates.

A closely related problem is that of realizing *maps* from k-mers to weights, i.e., data structures that do *not* explicitly represent the k-mers and so return arbitrary answers for out-of-set keys. In the context of this work, we distinguish between such approaches, *maps*, and dictionaries that instead represent *both* the k-mers and the weights. Besides minimal perfect hashing [23, 24], some efficient maps have been proposed and tailored specifically for genomic counts, such as based on *set-min sketches* [36] and *compressed static functions* (CSFs) [35]. These proposals leverage on the repetitiveness of the weights (low-entropy distributions) to obtain very compact space.

Lastly in this section, we report that other works [17, 14] considered the multi-document version of the problem studied here, that is, how to retrieve a vector of weights for a query k-mer, where each component of the vector represents the weight of the k-mer in a distinct document. Also such count vectors are usually very "regular" (or can be made so by introducing some approximation) [17] and present runs of equal symbols that can be compressed effectively with *run-length encoding* (RLE).

3 Representing Runs of Weights

In this section we describe the compression scheme for the weights that we use in SSHash. Recall that we indicate with $\mathcal{K}$ the set of n distinct $\langle k\text{-mer, weight}\rangle = \langle g, w(g)\rangle$ pairs, that we want to store in a dictionary. We first highlight the main properties of SSHash that we are going to exploit in the following to obtain good space effectiveness for the weights. (For all the other details concerning the SSHash index, we point the interested reader to our previous work [22].)

From a high-level perspective, SSHash implements the function $h : \Sigma^k \rightarrow \{0, 1, \ldots, n\}$, where $n = |\mathcal{K}|$ and Σ^k is the whole set of k-length strings over the DNA alphabet $\Sigma = \{\mathsf{A}, \mathsf{C}, \mathsf{G}, \mathsf{T}\}$. In particular, $h(g)$ is a unique value $1 \leq i \leq n$ if $g \in \mathcal{K}$; or $h(g) = 0$ otherwise, i.e., if $g \notin \mathcal{K}$. In other words, SSHash serves the same purpose of a minimal and perfect hash function (MPHF) [24] for $\mathcal{K}$ but, unlike a traditional MPHF, SSHash *rejects* alien k-mers. This is possible because the k-mers of $\mathcal{K}$ are actually represented in SSHah whereas the space of a traditional MPHF does not depend on the input keys.

The value $i = h(g)$ for the k-mer $g \in \mathcal{K}$ is the handle of g, or its "hash" code. The hash codes can be used to associate some satellite information to the k-mers such as, for example, the weights themselves using an array $W[1..n]$ where $W[h(g)] = w(g)$. The crucial property of SSHash in which we are interested is that the function h *preserves the relative order* of the k-mers, that is: if $g_1[1..k]$ and $g_2[1..k]$ are two k-mers with $g_1[2..k] = g_2[1..k-1]$ (i.e., g_2 comes

```
>5 5 5 5 5 5 5 5 5 5 5 5 5 5
GGTAATGCAGCCAGGGATGCAACGACCGCAACAGAAAAAGCCCG

>4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 4 4 4 4
CAGCTCATTACAGAAAAAATACCGCTCACCGCCCTGCACCGTCAGGTCAATTTCCCTGAGCACCACCCGCGGTGACTGCTCTGATTTAACC

>4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
CAGCTATGCAGGAGACAAGAATCGCCAGCTTACCCGTTACAGCGATACCCGCTGGCATG

>13 13 13 13 13 13 13
TCAGGTGTACGGTGTGCGTAAAGTCTGGCGTCAGTTG
```

Figure 1 An example collection $\mathcal{S}$ of 4 weighted sequences (for $k = 31$) drawn from the genome of *E. Coli* (Sakai strain). With alternating colors we render the change of weight in the runs. There are 111 k-mers in the example but just 6 runs in the weights: $\mathit{RLW} = \langle 5, 14\rangle\langle 4, 18\rangle\langle 2, 8\rangle\langle 1, 31\rangle\langle 4, 33\rangle\langle 13, 7\rangle$. Note that a run can cross the boundary between two (or more) sequences, as it happens for the run $\langle 4, 18\rangle$ which covers completely the third but also the part of the second sequence.

immediately after g_1 in a string), then $h(g_2) = h(g_1) + 1$. Therefore, consecutive k-mers, i.e., those sharing an overlap of $k - 1$ symbols, are also given consecutive hash codes. This is achieved in SSHash by pre-processing the input set $\mathcal{K}$ into a so-called *spectrum-preserving string set* (or SPSS) $\mathcal{S}$ – a collection of strings $\mathcal{S} = \{S_1, \ldots, S_m\}$ where each k-mer of $\mathcal{K}$ appears exactly once. We omit the details here on how the collection $\mathcal{S}$ can be built; we only report that there are efficient algorithms for this purpose that also try to minimize the total number of symbols in $\mathcal{S}$, i.e., the quantity $\sum_{i=1}^{m} |S_i|$. One such algorithm is the UST algorithm [29] that we also use to prepare the input for SSHash.

Therefore, once an order $S_1, \ldots, S_m$ for the strings of $\mathcal{S}$ is fixed, then also an order $i = 1, \ldots, n$ for the k-mers g_i is uniquely determined. Let $W[1..n]$ be the sequence of weights in this order. Then, we have: $h(g_i) = i$ and $W[i] = w(g_i)$, for $i = 1, \ldots, n$.

This order-preserving behavior of h induces a property on the sequence of weights $W[1..n]$ that significantly aids compression: W contains *runs*, i.e., maximal sub-sequences of *equal weights*. This is so because consecutive k-mers are very likely to have the same weight due to the high specificity of the strings. This a known fact, also observed in prior work [12, 17, 35]. Here, we are exploiting the order of the k-mers given by SSHash to preserve the natural order of the weights in W. Note that this cannot be achieved by approximate schemes that do not represent the k-mers themselves, like a generic MPHF or a CSF. Even if the k-mers were available, those schemes are unable to assign consecutive hashes to consecutive k-mers, actually shuffling the weights at random and, thus, making W very difficult to compress.

It is standard to represent a sequence W featuring r runs of equal symbols using *run-length encoding* (RLE), i.e., W is modeled as a sequence of run-length pairs $\mathit{RLW} = \langle w_1, \ell_1\rangle\langle w_2, \ell_2\rangle \cdots \langle w_r, \ell_r\rangle$ where w_i and ℓ_i are, respectively, the value of the run and the length of the i-th run in W. Figure 1 shows an example of RLW for a collection $\mathcal{S}$ with 4 weighted strings.

Encoding RLW

Let $\mathcal{D}$ be the set of all distinct w_i in RLW. Clearly, $r \geq |\mathcal{D}|$ as we must have at least one run per distinct weight. We store $\mathcal{D}$ using $|\mathcal{D}|\lceil\log_2 \mathit{max}\rceil$ bits where $\mathit{max} \geq 1$ is the largest w_i. We use $\mathcal{D}$ to uniquely represent each w_i in RLW with $\lceil\log_2 |\mathcal{D}|\rceil$ bits. Since runs are maximal sub-sequences in W by definition, then $w_i \neq w_{i+1}$ for every $i = 1, \ldots, r - 1$ (adjacent weights must be different). Then we take the prefix-sums of the sequence $0, \ell_1, \ldots, \ell_{r-1}$ into an array

$L[1..r]$ and encode it with Elias-Fano [9, 10][1]. By construction we have that $\sum_{i=1}^{r} \ell_i = n$ since the runs must cover the whole set of k-mers. So the largest element in L is actually $n - \ell_r$ and we spend at most $r\lceil \log_2(n/r) \rceil + 2r + o(r)$ bits for L. Summing up, we spend at most

$$r \cdot \left(\lceil \log_2 |\mathcal{D}| \rceil + \left\lceil \log_2 \left(\frac{n}{r} \right) \right\rceil + 2 + o(1) \right) + |\mathcal{D}| \lceil \log_2 max \rceil \text{ bits}$$

for representing RLW on top of the space of SSHash. In conclusion, the weights are represented in space proportional to the number of runs in W (i.e., $r = |RLW|$) and *not* proportional to the number of k-mers, which is n. As a consequence, this space is likely to be considerably less than the empirical entropy $H_0(W)$ as we are going to see with the experiments in Section 5.

To retrieve the weight $w(g)$ from $i = h(g)$, all that is required is to identify the run containing i. This operation is done in $O(\log(n/r))$ time with a predecessor query over L given that we represent L with Elias-Fano. If the identified run is the j-th run in W, then w_j is retrieved in $O(1)$ from $\mathcal{D}$.

4 Reducing the Number of Runs

In Section 3 we presented an encoding scheme for the k-mer weights whose space is proportional to the *number of runs* in the sequence of weights W. Therefore, in this section we consider the problem of reducing the number of runs in the weights to optimize the space of the encoding.

Rules of the Game

We assume that the strings in $\mathcal{S}$ are *atomic* entities: it is not allowed to partition them into sub-strings (e.g., in correspondance of the runs of weights in the strings). In fact, since the strings are obtained by the UST algorithm [29] with the purpose of *minimizing* the number of nucleotides as we explained in Section 3, breaking them will lead to an increased space usage for the k-mers, actually dwarfing any space-saving effort spent for the weights. With this constraint specified, there are only *two* degrees of freedom that can be exploited to obtain better compression for W: (1) the *order* of the strings, and (2) the *orientation* of the strings. Altering $\mathcal{S}$ using these two degrees of freedom does not affect the correctness nor the (relative) order-preserving property of the function $h : \Sigma^k \to \{0, 1, \ldots, n\}$ implemented by SSHash. In fact, as evident from our description in Section 3, the output of h will still be $\{1, \ldots, n\}$ as the k-mers themselves do *not* change (even when taking reverse-complements into account as they are considered to be identical). What changes is just the absolute order of the k-mers as a consequence of permuting the order of the strings $\{S_1, \ldots, S_m\}$ in $\mathcal{S}$.

Therefore, our goal is to permute the order of the strings in $\mathcal{S}$ and possibly change their orientations to reduce the number of runs in W. We now consider an illustrative example to motivate why both these two operations – those of changing the order and orientation of a string – are important to reduce the number of runs. Refer to Figure 2a which shows an example collection of $m = |\mathcal{S}| = 4$ weighted strings (for $k = 3$). Applying the permutation

[1] Elias-Fano represents a monotone integer sequence $S[1..n]$ with $S[n] \le U$ in at most $n\lceil \log_2(U/n) \rceil + 2n$ bits. With $o(n)$ extra bits it is possible to decode any $S[i]$ in constant time and support predecessor queries in $O(\log(U/n))$ time. For a complete description of the method, we point the reader to the survey by Pibiri and Venturini [28, Section 3.4]. We also remark that Elias-Fano has been recently used in many compressed, practical, data structures (see, e.g., [25, 26, 27]).

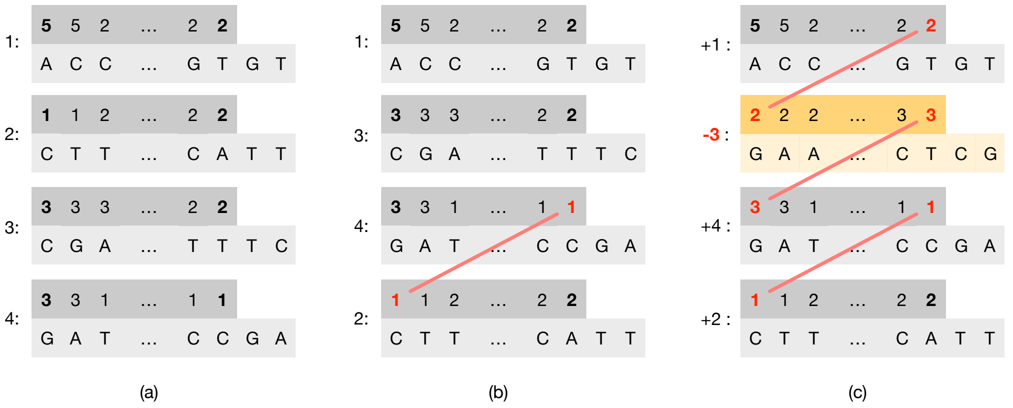

Figure 2 In (a), an example input collection $\mathcal{S}$ of $m = |\mathcal{S}| = 4$ weighted strings (for $k = 3$), where the end-point weights are highlighted in bold font. In (b), the order of the strings is changed according to the permutation $\pi = [1, 4, 2, 3]$ and, as a result, the number of runs is reduced by 1 (the last run in string 4 is glued with the first run of string 2). Lastly, in (c), it is shown that changing the orientation of string 3 (taking the reverse complement of the string and reversing the order of the k-mer weights) makes it possible to glue other two runs. Given that reducing the number of runs by $m - 1$ is the best achievable reduction, the number of runs in (c) is therefore the minimum for the original collection in (a).

```
permute($\mathcal{S}$, $\pi$):
    let $\tilde{\mathcal{S}} = \{\tilde{S}_1, \ldots, \tilde{S}_m\}$ be a new collection of empty strings
    for $i = 1$; $i \leq m$; $i = i + 1$ :
        $j = \pi[i]$
        if $j < 0$ : $\tilde{S}_{-j}$ = reverse($S_i$)
        else : $\tilde{S}_j = S_i$
    return $\tilde{\mathcal{S}}$
```

Algorithm 1 The algorithm permute takes as input a collection $\mathcal{S} = \{S_1, \ldots, S_m\}$ of weighted strings and a signed permutation π and returns the permuted collection $\tilde{\mathcal{S}} = \pi(\mathcal{S})$. The function reverse takes the reverse-complement of a string and reverse its weights.

$\pi = [1, 4, 2, 3]$ as shown in Figure 2b reduces the number of runs by 1 because the run at the junction of string 4 and 2 can be glued. Lastly, applying the *signed* permutation $\pi = [+1, +4, -2, +3]$ as in Figure 2c reduces the number of runs by 3, which is the best possible. Our objective is to compute such a signed permutation π for an input collection of strings, in order to permute $\mathcal{S}$ as shown in Algorithm 1.

Figure 2 also suggests that the final result π solely depends on the weight of the first and last k-mer of each sequence – which we call the *end-point* weights (or just end-points) of a sequence – and not on the other weights nor the nucleotide sequences. Therefore, it is useful to model an input collection $\mathcal{S}$ using a graph, defined as follows.

▶ **Definition 1** (End-point Weight Graph). Given a collection of weighted sequences $\mathcal{S}$, let G be a graph where:

- There is a node u for each sequence of $\mathcal{S}$ and u has two sides – a *left* and a *right* side – respectively labelled with the end-point weights of the sequence.
- There is an edge between any two distinct nodes u and v that have a side with the same weight.

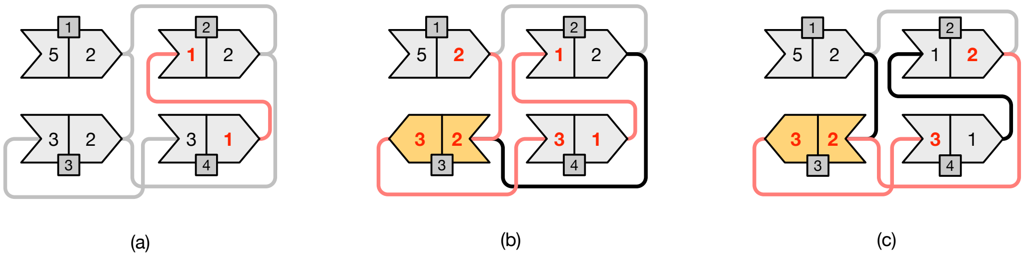

Figure 3 The same example of Figure 2 but modeled using end-point weight graphs. Each node is represented using an arrow-like shape with two-matching sides. Only opposite sides having the same weight can be matched. The numbers inside the shapes represent the end-point weights; the extra darker square contains the node identifier. An arrow oriented from *left-to-right* models a node with *positive* sign; vice versa, an arrow oriented from *right-to-left* models a node with *negative* sign. Gray edges represent edges that *cannot* be traversed without changing the orientation of one of the two connected nodes. Black edges represent edges that can be traversed. Lastly, we highlight in red the edges that belong to paths in a graph cover. The example in (a) corresponds to that of Figure 2b where no node has changed orientation and, therefore, we have three paths in the cover: $(+4 \rightarrow +2)$, $(+3)$, and $(+1)$. Other two different covers are shown in (b) and (c). In (b) the cover contains the single path $(+1 \rightarrow -3 \rightarrow +4 \rightarrow +2)$ and corresponds to the example of Figure 2c where the node 3 was changed orientation from + to − (shown in yellow color). In (c) the cover contains the two paths $(+2 \rightarrow -3 \rightarrow +4)$ and the singleton path $(+1)$.

The graph G is called the *end-point weight graph* for $\mathcal{S}$ and indicated with $ewG(\mathcal{S})$.

In the following, we indicate a node u in $ewG(\mathcal{S})$ using the identifier (id) of the corresponding sequence of $\mathcal{S}$. Also, we associate to u a *sign* $\in \{-1, +1\}$ (or orientation), indicating whether the sequence should be reverse-complemented. In summary, a node u in $ewG(\mathcal{S})$ is the 4-tuple $(id, left, right, sign)$.

▶ **Definition 2** (Oriented Path). An *oriented* path in $ewG(\mathcal{S})$ is either a single node (singleton path) or a sequence of nodes $(u_1 \rightarrow \cdots \rightarrow u_\ell)$ where each consecutive pair of nodes $u_i \rightarrow u_{i+1}$ is oriented in such a way that $u_i.right = u_{i+1}.left$, for any $1 \leq i < \ell$.

Since we will be interested only in oriented paths, we just refer to them as "paths". For ease of notation, we will indicate a path in our examples as a sequence of signed numbers $(i_1 \rightarrow \cdots \rightarrow i_\ell)$ where each number represents a node's *id* and its sign represents the node's *sign*. The first and the last node in the path are called, respectively, the *front* and the *back* of the path. The weights *front.left* and *back.right* are the two end-points of the path.

Given this graph model, it follows that the problem of finding a signed permutation π for $\mathcal{S}$ is equivalent to that of computing a (disjoint-node) *path cover* C for $ewG(\mathcal{S})$, i.e., a set of paths in $ewG(\mathcal{S})$ that visit *all* the nodes and where each node belongs to *exactly one* path. In fact note that, given a cover C for $ewG(\mathcal{S})$, there is a linear-time reduction from C to π as illustrated in Algorithm 2. Since the cover C is a disjoint-node path cover, the correctness of the algorithm is immediate as well as its complexity of $\Theta(m)$.

Figure 3 illustrates the same example of Figure 2 but with end-point weight graphs. In Figure 3b we would obtain a cover $C = \{(+1 \rightarrow -3 \rightarrow +4 \rightarrow +2)\}$ formed by a single path. In this case the permutation π, following Algorithm 2, would be $\pi[1] = +1$, $\pi[3] = -2$, $\pi[4] = +3$, and $\pi[2] = +4$. This is indeed the same permutation discussed in Figure 2c. Another example: for the graph in Figure 3c, the cover would be $C = \{(+2 \rightarrow -3 \rightarrow +4), (+1)\}$ and the permutation π would be $\pi[2] = +1$, $\pi[3] = -2$, $\pi[4] = +3$, and $\pi[1] = +4$.

```
reduce($C$):
    $j = 1$
    let $\pi[1..m]$ be a new array
    for each path $p \in C$ :
        for each node $u \in p$ :
            $\pi[u.id] = u.sign \cdot j$
            $j = j + 1$
    return $\pi$
```

Algorithm 2 The algorithm reduce takes as input a path cover C computed for $ewG(\mathcal{S})$ and returns the corresponding signed permutation π. The complexity of the algorithm is $\Theta(m)$ since the number of nodes in $ewG(\mathcal{S})$ is m and each node appears exactly once in C.

4.1 A Lower Bound to the Number of Runs

We showed that changing the order and orientation of the strings in $\mathcal{S}$ can reduce the number of runs in the weights. The crucial question is: by how much? We are interested in deriving a lower bound to the number of runs achievable after applying the signed permutation π to $\mathcal{S}$. Since we modeled the problem of computing π as the problem of finding a path cover C for $ewG(\mathcal{S})$, we reason in terms of $ewG(\mathcal{S})$ and C.

Let $|C|$ be the number of paths in the cover C. Let r_i be the number of runs in S_i and let R be the total number of runs, i.e., $R = \sum_{i=1}^{m} r_i$. Then there are at least $R - m$ runs in $\mathcal{S}$ *regardless* the order of the sequences. Therefore, a straightforward lower bound to the number of runs would be $\max\{|\mathcal{D}|, R - m + 1\}$. This lower bound assumes (very optimistically) that we are able to obtain a cover with a single path, i.e., $|C| = 1$, hence reducing the total number of runs R by $m - 1$ which is the best reduction achievable with m sequences. Note, however, that the bound cannot be lower than $|\mathcal{D}|$ – the number of distinct weights in the input – because, clearly, there must be at least one run per distinct weight value. We would like to improve the bound $\max\{|\mathcal{D}|, R - m + 1\}$ knowing that, in general, we could not be able to form one single path.

We observe that the final number of runs r in the permuted $\mathcal{S}$ will be equal to $R - m + |C|$. In fact, every path in C must begin (resp. end) with a node whose left side (resp. right side) cannot be glued with any other path's side. Therefore, a new run begins with the first node of every path. Since we wish to minimize the quantity $R - m + |C|$, and considering that $R - m$ is constant for a given $\mathcal{S}$, it follows that the problem reduces to that of minimizing $|C|$, the number of paths in the cover. In other words, the problem of minimizing the number of runs r is equivalent to that of finding a minimum-cardinality path cover C for $ewG(\mathcal{S})$.

Therefore our strategy is to give a lower bound to the number of paths $|C|$. To do so, we compute the number of end-point weights, say n_e, that must appear as end-points of the paths (left side of the front node of a path, or right side of the back node of a path). Since a path has exactly two end-points, then it follows that $|C| \geq \lceil n_e/2 \rceil$ and, in turn, that the final number of runs r in $\pi(\mathcal{S})$ is *at least* $R - m + \lceil n_e/2 \rceil \geq \max\{|\mathcal{D}|, R - m + 1\}$.

Since we now focus on the end-point weights of the nodes, in this section we will denote a node $(id, left, right, sign)$ just by its weights $(left, right)$. We start with a preliminary Lemma and a Definition. (See Appendix A for all the proofs omitted from the section.)

▶ **Lemma 3.** Let us consider $d > 0$ equal nodes (w, x). If d is even, then the d nodes originate a path of either end-points (w, w) or (x, x). If d is odd, then the path has end-points (w, x).

▶ **Definition 4** (Incidence Set). Given the weight w, a set I_w of nodes where w appears as end-point is called an *incidence set* for w. Let $n(I_w)$ be the number of times w appears in the nodes of I_w. Note that $n(I_w) \geq |I_w|$ because there could be nodes (w, w) in I_w.

▶ **Example 5.** The sets

$$\begin{aligned}
I_w^1 &= \{(w,x),(w,y),(w,z),(w,t),(w,l)\}\\
I_w^2 &= \{(w,w),(w,x),(w,y),(w,z)\}\\
I_w^3 &= \{(w,w),(w,w),(w,w),(x,w),(w,x)\}\\
I_w^4 &= \{(w,x),(w,x),(y,w),(w,y),(w,y),(w,z),(w,z),(w,z),(w,z)\}
\end{aligned}$$

are four different incidence sets for w with $n(I_w^1) = n(I_w^2) = 5$, $n(I_w^3) = 8$, and $n(I_w^4) = 9$. The set $\{(w,x),(x,y)\}$, instead, is not an incidence set for w because the node (x,y) does not have w as an end-point.

Next, we give the following central Lemma that will help in counting the number of weights that must appear as end-points of the paths in C.

▶ **Lemma 6.** Given an incidence set I_w, if $n(I_w)$ is *odd* then only one path will contain w as end-point among all the paths that can be created from the nodes in I_w.

▶ **Example 7.** Let us consider an example for Lemma 6. The sets I_w^1 and I_w^2 in Example 5 are both canonical since all other weights different from w are distinct, except for one single node (w,w) in I_w^2. It is then easy to see that no matter what paths are created, there will always be one extra node that will remain alone. The set I_w^4 in Example 5, with $n(I_w^4) = 9$, is not canonical instead and we have: $D_x = \{(w,x),(w,x)\}$, $D_y = \{(y,w),(w,y),(w,y)\}$, and $D_z = \{(w,z),(w,z),(w,z),(w,z)\}$, with $d_x = |D_x| = 2$, $d_y = |D_y| = 3$, and $d_z = |D_z| = 4$. Since $d_x = 2$, then the nodes in D_x can be collapsed into either (w,w) or (x,x). Since $d_y = 3$, then the nodes in D_y can be collapsed to (w,y). Lastly, since $d_z = 4$, the nodes in D_z can be collapsed to either (w,w) or (z,z). Again, regardless the choices done, I_w^4 can always be reduced to a canonical set.

Let now eW be the set of the distinct end-point weights of $\mathcal{S}$. For every weight $w \in eW$, let I_w^{max} be the incidence set of *maximum-cardinality*, i.e., the one obtained by considering *all* the nodes in $ewG(\mathcal{S})$. We partition eW into three disjoint sets, eW_{odd}, eW_{even}, and eW_{equal}. Based on the properties of I_w^{max}, w belongs to one of the three sets as follows.

- If $n(I_w^{max})$ is odd, then $w \in eW_{odd}$. In this case, we are sure by Lemma 6 that w will appear as end-point of some path in the cover.
- If all the nodes in I_w^{max} are equal to (w,w), then $w \in eW_{equal}$. In this case $n(I_w^{max})$ will always be even and, for Lemma 6, w will appear *twice* as end-points of the same path in the cover.
- If $n(I_w^{max})$ is even but the nodes in I_w^{max} are *not* all equal to (w,w), then $w \in eW_{even}$. In this case, we cannot be sure that w will *not* appear as end-point. If $n(I_w^{max})$ is even *and* I_w^{max} contains *all distinct* nodes, then w will not appear. But if I_w^{max} contains two identical nodes, say of the form (w,x), then w may or may not appear. In fact, as already noted, depending on whether a path is created as $(w,x) \to (x,w)$ or $(x,w) \to (w,x)$, w will appear (in the first case) or not (in the second case). The set I_w^3 from Example 5 illustrates this case.

▶ **Lemma 8.** $|W_{odd}|$ is even.

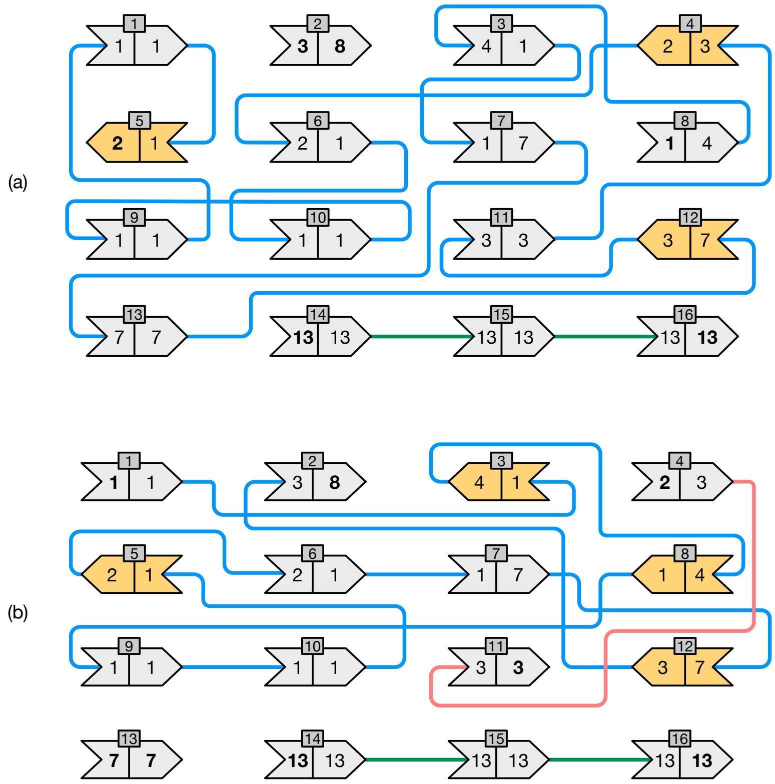

Figure 4 Two different path covers for an example graph with 16 nodes. Nodes linked by edges with the same color belong to the same cover; yellow nodes are those whose orientation was changed. For the graph in the picture, the lower bound in Corollary 9 yields $|C| \geq 3$. The covers in (a) contains 3 paths and is, therefore, optimal. The cover in (b), instead, contains 4 paths.

Therefore, we obtain the following Corollary.

▶ **Corollary 9.** The number of paths in C is at least $(2|eW_{equal}| + |eW_{odd}|)/2$.

Proof. Immediate by first applying Lemma 6 to the maximal incidence sets, then noting that each each path has exactly two end-points, and lastly taking into account the cardinality of eW_{odd} for Lemma 8. ◀

Let us now compute the lower bound of Corollary 9 on some example graphs.

▶ **Example 10.** First, we re-consider the example graph from Figure 2. In that example we have a set of end-point weights $eW = \{1, 2, 3, 5\}$, where $I_1^{max} = \{(1,2),(3,1)\}$, $I_2^{max} = \{(5,2),(1,2),(3,2)\}$, $I_3^{max} = \{(3,2),(3,1)\}$, and $I_5^{max} = \{(5,2)\}$. Since $n(I_2^{max}) = 3$ and $n(I_5^{max}) = 1$, then $eW_{odd} = \{2,5\}$ and $eW_{equal} = \varnothing$. Therefore we sure that both weights 2 and 5 will appear as end-point weights of some paths in the cover. For Corollary 9, any path cover will contain at least $(2|eW_{equal}| + |eW_{odd}|)/2 = (2 \cdot 0 + 2)/2 = 1$ path. Indeed the cover shown in Figure 2b contains one single path and so, it is optimal, whereas the cover in Figure 2c contains 2 paths and is not optimal.

▶ **Example 11.** We now consider the larger example in Figure 4 for a graph with 16 nodes. In this example we have a set of end-point weights $eW = \{1, 2, 3, 4, 7, 8, 13\}$ and the incidence sets are as follows: $I_1^{max} = \{(1,1),(4,1),(2,1),(2,1),(1,7),(1,4),(1,1),(1,1)\}$;

$I_2^{max} = \{(2,3),(2,1),(2,1)\}$; $I_3^{max} = \{(3,8),(2,3),(3,3),(3,7)\}$; $I_4^{max} = \{(4,1),(1,4)\}$; $I_7^{max} = \{(1,7),(3,7),(7,7)\}$; $I_8^{max} = \{(3,8)\}$; $I_{13}^{max} = \{(13,13),(13,13),(13,13)\}$. Since we have $n(I_1^{max}) = 11$, $n(I_2^{max}) = 3$, $n(I_3^{max}) = 5$, and $n(I_8^{max}) = 1$, then $eW_{odd} = \{1,2,3,8\}$ and $|eW_{odd}| = 4$ which is even (Lemma 8). Therefore we are sure that the weights 1, 2, 3, and 8 will appear as end-points of some paths in the cover. The incidence set I_{13}^{max} is made of nodes $(13,13)$, so $eW_{equal} = \{13\}$ and $|eW_{equal}| = 1$. Also in this case we are sure the weight 13 will appear (twice) as end-point of some path. ($eW_{even} = \{4,7\}$.) For Corollary 9 we derive that a path cover for the example graph will contain at least $(2 \cdot 1 + 4)/2 = 3$ paths.

Figure 4 shows two different path covers for the same graph, that are $C_a = \{(+8 \to +3 \to +7 \to +13 \to -12 \to +11 \to -4 \to +6 \to +10 \to +9 \to +1 \to -5), (+2), (+14 \to +15 \to +16)\}$ and $C_b = \{(+1 \to -3 \to -8 \to +9 \to +10 \to -5 \to +6 \to +7 \to +12 \to +2), (+4 \to +11), (+13), (+14 \to +15 \to +16)\}$. The cover C_a is optimal for the lower bound as is contains 3 paths; the cover C_b contains 4 paths (1 more than necessary). It is not difficult to see that we cannot find a path cover with less than 3 paths for the graph in Figure 4.

Lastly, we summarize the main result of this section with the following Theorem.

▶ **Theorem 12.** Let $\mathcal{S}$ be a collection of m weighted strings. Let eW be the set of the distinct end-point weights of the strings in $\mathcal{S}$ and $R = \sum_{i=1}^{m} r_i$, where r_i is the number of runs in the weights of the i-th string of $\mathcal{S}$. Then $\mathcal{S}$ can be permuted to form at least

$$r_{lo} = R - m + (2|eW_{equal}| + |eW_{odd}|)/2 \text{ runs in the weights,} \qquad (1)$$

where $eW_{equal} = \{w \in eW \mid I_w^{max} = \{(w,w)\}^d$, for some $d > 0\}$ and $eW_{odd} = \{w \in eW \mid n(I_w^{max})$ is odd$\}$ are defined using the incidence sets for the nodes of $ewG(\mathcal{S})$.

Proof. There are exactly $R - m$ runs in $\mathcal{S}$ that do not depend on the order of the strings in $\mathcal{S}$. Then $\mathcal{S}$ can be permuted as to have $R - m + |C|$ runs where C is a disjoint-node path cover for $ewG(\mathcal{S})$. The theorem follows for Corollary 9 on C. ◀

4.2 Computing a Cover

We showed, via a linear-time reduction (Algorithm 2), that the problem of finding a permutation for $\mathcal{S}$ with the goal of reducing the number of runs in the weights is equivalent to that of computing a path-cover C for the graph $ewG(\mathcal{S})$. In Section 4.1 we also gave a lower bound to the number of runs that our strategy can achieve. The lower bound depends on the number of paths in C (Corollary 9). In this section we therefore present an algorithm to actually compute C in *expected linear-time* in the number of nodes of $ewG(\mathcal{S})$. Recall that $ewG(\mathcal{S})$ has $m = |\mathcal{S}|$ nodes, so the complexity is $\Theta(m)$.

The algorithm is given in Algorithm 3. It manipulates the incidence sets for the end-point weights and a set of unvisited nodes, respectively indicated with *incidence* and *unvisited* in the pseudo-code, and initialized in the lines 4-7. The main loop in the lines 8-22 takes an arbitrary unvisited node and starts a new path from that node. The inner loop in the lines 11-20 greedily tries to extend the current path as much as possible: at every step of the loop, (1) a node is appended to the path, (2) it is erased from the set of unvisited nodes and from the incidence sets where it belongs to, then (3) the next node to append is selected from one of the two incidence sets for the path's end-point weights. When no extension is possible for both ends, the current path is printed in the lines 21-22. In practice, the output can be written to a file, from which the signed permutation π can be derived with Algorithm 2.

```
1  cover($ewG(\mathcal{S})$):
2      $incidence = \varnothing$
3      $unvisited = \varnothing$
4      for each node $u \in ewG(\mathcal{S})$:
5          $unvisited$.insert($u$)
6          $incidence[u.left]$.insert($u$)
7          $incidence[u.right]$.insert($u$)
8      while $unvisited \neq \varnothing$ :
9          $u = unvisited$.take()                    ▷ take an unvisited node $u$
10         $p = \varnothing$                         ▷ a new path
11         while true :
12             extend $p$ with $u$                   ▷ append $u$ to the front or to the back of $p$
13             $unvisited$.erase($u$)
14             $incidence[u.left]$.erase($u$)
15             $incidence[u.right]$.erase($u$)
16             if $incidence[p.back.right] \neq \varnothing$ :     ▷ first, try to extend to the right
17                 $u = incidence[p.back.right]$.take()
18             else if $incidence[p.front.left] \neq \varnothing$ :  ▷ then, try to extend to the left
19                 $u = incidence[p.front.left]$.take()
20             else : break                          ▷ $p$ cannot be extended anymore
21         for each $u \in p$ :                      ▷ print $p$
22             print $(u.sign, u.id)$
```

Algorithm 3 The cover algorithm takes an input end-point weight graph $ewG(\mathcal{S})$ and prints a set of paths covering the nodes of $ewG(\mathcal{S})$.

If we use hashing to implement the sets *incidence* and *unvisited*, then the operations insert/erase/take are all supported in $O(1)$ expected time. Also appending a node to one of the two path's end-points can be done in constant (amortized) time using a double-ended queue to represent a path. Figure 5 illustrates how the path is extended with a node (line 12). Therefore, the algorithm runs in expected $\Theta(m)$ time and consumes $\Theta(m)$ space because: (1) at most $2m$ nodes (and at least m) are inserted in *incidence* and exactly m in *unvisited* during the initialization lines 4-7; (2) during the main loop in the lines 8-22, each node is visited, appended to a path, and printed exactly once[2].

5 Experiments

In this section we evaluate the proposed weight compression scheme for SSHash and compare it to several competitive baselines. We first describe our experimental setup.

Experiments were run using a server machine equipped with an Intel i9-9940X processor (clocked at 3.30 GHz) and 128 GB of RAM. All the tested software was compiled

[2] An important implementation remark: In practice, we implemented the *incidence* sets using a single open-addressing hash-table with a sorted array of $2m$ nodes and an extra array of offsets into the vector to distinguish between the different incidence sets. This makes the algorithm much faster and lighter than a straightforward implementation using a linear-chaining hash-table (e.g., the C++'s `std::unordered_set`), with chains usually implemented as linked lists.

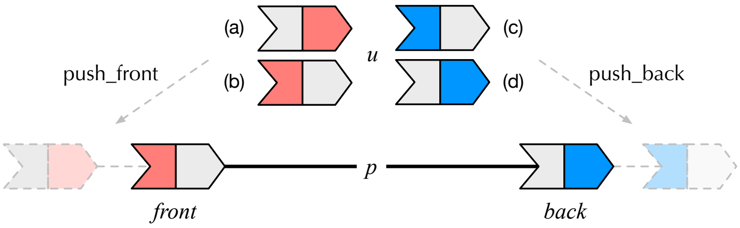

Figure 5 A graphical visualization of line 12 in Algorithm 3 which extends the current path p with a node u. When p is not empty, four different cases can arise, as illustrated in (a), (b), (c), and (d). In cases (b) and (d), the sign of u is changed to match one of the two path's end-points.

Table 1 Some basic statistics for the datasets used in the experiments, for $k = 31$, such as: number of distinct k-mers (n), number of distinct weights ($|\mathcal{D}|$), largest weight (max), expected weight value (E), and empirical entropy of the weights ($H_0(W)$).

Dataset	n	$\lvert\mathcal{D}\rvert$	$\lceil\log_2 \lvert\mathcal{D}\rvert\rceil$	max	$\lceil\log_2 max\rceil$	E	$H_0(W)$
E-Coli	5,235,781	22	5	27	5	1.05	0.206
S-Enterica-100	13,074,614	587	10	3,483	12	37.47	4.420
Human-Chr-13	90,911,778	806	10	6,354	13	1.08	0.160
C-Elegans	94,006,897	398	9	3,478	12	1.07	0.223

with gcc 11.2.0 under Ubuntu 19.10 (Linux kernel 5.3.0, 64 bits), using the flags `-O3` and `-march=native`. Our implementation of SSHash is written in C++17 and available at `https://github.com/jermp/sshash`.

All timings were collected using a single core of the processor. The dictionaries are loaded in internal memory before executing queries. For all the experiments, we fix k to 31.

Datasets

We use the following genomic collections: E-Coli and C-Elegans are, respectively, the full genomes of *E. Coli* (Sakai strain) and *C. Elegans* that were also used in the experimentation by Shibuya et al. [35]; S-Enterica-100 is a pan-genome of 100 genomes of *S. Enterica*, collected by Rossi et al. [32]; Human-Chr-13 is the 13-th human chromosome from the genome assembly GRCh38. Table 1 reports some basic statistics for the collections. The weights were collected using the tool BCALM (v2) [7]. In general, note the very low empirical entropy of the weights, $H_0(W)$. This is expected since most k-mers actually appear once for large-enough values of k. Instead, the weights on the pan-genome S-Enterica-100 have much higher entropy due to the fact that many k-mers have weight equal to the number of genomes in the collection (in this specific case, equal to 100). This is useful to test the effectiveness of our encoding on both low- and high- entropy inputs.

Weight Compression in SSHash

We now consider the space effectiveness of the encoding scheme described in Section 3. Table 2 reports the space as average bits per k-mer: we see that, in all cases, the space is well below the empirical entropy lower bound $H_0(W)$ – usually below by several times. The optimization strategy described in Section 4 brings further advantage. (The space shown is comprehensive of the $|\mathcal{D}|\lceil\log_2 max\rceil$ bits used to represent the distinct weights in the

Table 2 Space for the weights in SSHash reported in bits/k-mer, *before* and *after* the run-reduction optimization from Section 4. For reference, we also report how many times the achieved space is better than the empirical entropy of the weights $H_0(W)$.

Dataset	$H_0(W)$	*before*		*after*	
E-Coli	0.206	0.017	(12.11×)	0.014	(15.10×)
S-Enterica-100	4.420	0.592	(7.47×)	0.401	(11.02×)
Human-Chr-13	0.160	0.136	(1.18×)	0.107	(1.50×)
C-Elegans	0.223	0.069	(3.23×)	0.055	(4.05×)

Table 3 The number of input strings (m) for SSHash as computed by UST [29], the lower bound on the number of runs (r_{lo}) computed using Equation (1) in Theorem 12, and number of actual runs (r) after the optimization. We report the increase, in percentage, of r compared to r_{lo}. The last two columns show the run-time of the path cover Algorithm 3, in total milliseconds (ms) and average nanoseconds per node (ns/node).

Dataset	m	r_{lo}	r		Alg. 3 (ms)	Alg. 3 (ns/node)
E-Coli	2,102	3,723	3,723	(+0.0000%)	0.6	285
S-Enterica-100	150,604	277,649	277,658	(+0.0032%)	53.0	352
Human-Chr-13	266,113	462,175	462,197	(+0.0048%)	94.6	355
C-Elegans	140,452	247,661	247,669	(+0.0032%)	47.1	335

collection. Note that this space takes a negligible fraction of the total space since $|\mathcal{D}|$ is very small as reported in Table 1.)

Table 3, instead, shows the performance of the path cover Algorithm 3. As already mentioned in Section 3, the set of strings indexed by SSHash is obtained by building a spectrum-preserving string set (SPSS) from the raw genome, using the algorithm UST [29] over the output of BCALM [7]. (At our code repository `https://github.com/jermp/sshash` we provide further details on how to take these preliminary steps before indexing with SSHash.) The number of strings in each collection, m, determines the run-time of Algorithm 3 whose complexity is $\Theta(m)$. The linear-time complexity is evident from the reported timings and makes the algorithm very fast, taking on average a fraction of a microsecond per node.

The other important point to note is that Algorithm 3 is empirically *optimal* regarding the number of runs given that the final number of runs, r, is only slightly higher than the lower bound r_{lo} computed using Equation (1).

(In Appendix B we report additional experimental results.)

Overall Comparison

In Table 4 we show a comparison between the following weighted dictionaries (see also Section 2; links to the code repositories are included in the References):

- The dBG-FM index [6] based on the popular FM-index [11]. In particular, this representation implements a weighted k-mer dictionary via the *count* query which returns the number of occurrences of a given k-mer in the input. The *count* query, in turn, is implemented using rank queries over the BWT. The dBG-FM implementation has a main trade-off parameter, s, to control the practical performance of rank queries. We test the values $s = 32, 64, 128$.

Table 4 Dictionary space in average bits/k-mer and count time in average μsec/k-mer. For reference, we report in gray color the space and time of SSHash *without* the weight information.

Dictionary	E-Coli		S-Enterica-100		Human-Chr-13		C-Elegans	
	space	query-time	space	query-time	space	query-time	space	query-time
dBG-FM, $s = 128$	3.20	14.73	113.78	16.47	3.23	17.40	3.18	18.05
dBG-FM, $s = 64$	4.02	7.91	142.25	11.13	4.07	11.33	4.01	10.89
dBG-FM, $s = 32$	5.65	4.62	198.71	8.57	5.73	8.20	5.67	7.90
cw-dBG, $s = 128$	2.79	109.13	5.59	120.72	2.80	100.88	2.77	127.86
cw-dBG, $s = 64$	2.86	70.93	5.74	85.73	2.86	73.91	2.84	84.19
cw-dBG, $s = 32$	2.99	52.29	6.03	66.25	2.99	59.85	2.97	62.54
SSHash+BCSF	5.07	0.82	11.12	0.89	6.15	1.25	6.00	1.28
SSHash+AMB	4.90	1.34	9.27	1.65	6.08	1.95	5.88	1.97
w-SSHash	4.80	0.37	6.57	0.48	6.04	0.84	5.75	0.85
SSHash	4.79	0.34	6.15	0.41	5.93	0.76	5.69	0.77

- The recent cw-dBG [12] dictionary based on the data structure called BOSS [4]. Similarly to an FM-index, also cw-dBG has a trade-off parameter that we vary as $s = 32, 64, 128$. (The authors used $s = 64$ in their own experiments.)
- The *non*-weighted SSHash itself coupled with the fast compressed static function (CSF) tailored for low-entropy distributions, proposed by Shibuya et al. [35]. As reviewed in Section 2, a CSF does not represent the k-mers but just realizes a map from k-mers to their weights. Such map is collision-free only over the set of k-mers that was used to actually build the function. Therefore, we use SSHash as an efficient dictionary for the k-mers and the CSF to represent the weights. The authors proposed two different versions of their approach, BCSF and AMB, with different space/time trade-offs.
- The weighted SSHash dictionary proposed in this work, which we refer to as w-SSHash in the following, *after* the run-reduction optimization (Table 2 and 3). We use the *regular* index variant of SSHash. The main parameter of the index – the *minimizer* length – is always set to $\lceil \log_4 N \rceil + 1$ where N is the number of nucleotides in the SPSSs of the datasets, following the recommendation given in the previous paper [22]. Therefore, we use the following minimizer lengths: 13, 14, 15, and 15, for respectively, E-Coli, S-Enterica-100, Human-Chr-13, and C-Elegans. Also the AMB algorithm by Shibuya et al. [35] is based on minimizers and we use the same lengths.

We did not compare against deBGR [19] and Squeakr [21] as the authors of cw-dBG showed in their experimentation [12] that both tools take considerably more space than cw-dBG, e.g., one order of magnitude more space. Here, we are interested in a good balance between space effectiveness and query efficiency.

To measure query-time – the time it takes to retrieve the weight $w(g)$ given the k-mer g – we sampled 10^6 k-mers uniformly at random from the collections and use them as queries. We report the mean between 5 measurements. Half of the queries were transformed into their reverse complements to make sure we benchmark the dictionaries in the most general case.

The space of w-SSHash is generally competitive with that of the fastest variant of dBG-FM ($s = 32$), but w-SSHash has (more than) one order of magnitude better query time. Note that on S-Enterica-100 the dBG-FM index is space-inefficient since it redundantly represents many repeated k-mers. Using a higher sampling rate reduces the space of dBG-FM at the price of slowing down query-time; however, the most space-efficient variant tested ($s = 128$) is not even 2× smaller than w-SSHash.

The cw-dBG index is the smallest tested dictionary. Its space effectiveness is comparable to that of dBG-FM $s = 128$, and indeed generally twice as better as that of w-SSHash. The price to pay for this enhanced compression ratio is a significant penalty at query-time. Indeed, w-SSHash can be two order of magnitude faster than cw-dBG. Consider, for example, the two dictionaries built for S-Enterica-100: we have 0.5 vs. 66-120 μs per query.

The two CSFs, BCSF and AMB, make SSHash 2-3× slower than w-SSHash and even consistently larger. This comparison motivates the need for a unified data structure to handle efficiently both the k-mers *and* the weights, like w-SSHash. While the increase in space due to the CSF is not much for the low-entropy datasets because both BCSF and AMB are very space-efficient in those cases, the gap is more evident on S-Enterica-100.

As a last note, observe that there is no significant slowdown in accessing the weights in w-SSHash compared to a simpler membership query (the time reported in shaded color in Table 4), hence proving the RLE-based scheme to be efficient too and not only very effective.

6 Conclusions

In this work we extended the recent SSHash [22] dictionary to also store the weights of the k-mers in compressed format. In particular, we represented the weights using compressed *runs* of equal symbols. While using run-length encoding to compress highly repetitive sequences is not novel per se and indeed a folklore strategy at the basis of many other data structures, this allows to use a very small extra space (e.g., much less than the empirical entropy of the weights) on top of SSHash with only a slight penalty at retrieval time. The crucial point is that it is possible to use run-length encoding because SSHash *preserves the (relative) order* of the k-mers in the indexed sequences. The main practical take-away is, therefore, that SSHash handles weighted k-mer sets in an *exact* manner without noticeable extra costs. Our software is publicly available to encourage its use and reproducibility of results.

We also introduced the concept of *end-point weight graph* (*ewG*) and showed its usefulness in reducing the number of runs in the weights. Precisely, we showed that minimizing the number of runs in a collection of sequences corresponds to the problem of computing a minimum-cardinality path cover for the *ewG* of the sequences. We presented a greedy algorithm that computes a cover in expected linear-time (in the number of nodes of the graph) and showed that it is empirically almost optimal according to a lower bound on the number of runs. As a result of this optimization, the space spent to represent the weights is unlikely to be improved using run-length encoding.

Although several approaches in the literature [17, 21, 35, 36] also consider *approximate* weights, we did not pursue this direction here as the weights are already encoded space-efficiently in SSHash and in an *exact* way, so there may be no need for approximation.

The distribution of weights in large collections is usually expected to be very skew, i.e., most k-mers actually appear once and few of them repeat many times [35, 36]. A common strategy to save space is then to avoid the representation of the most frequent weight(s). Note that, since we represent runs of weights and not the individual weights, we are already optimizing (potentially very large) sub-sets of weights equal to the most frequent one. That is, run-length encoding is also a good match for such skew distributions.

References

1 Fatemeh Almodaresi, Hirak Sarkar, Avi Srivastava, and Rob Patro. A space and time-efficient index for the compacted colored de Bruijn graph. *Bioinformatics*, 34(13):i169–i177, 2018.

2 Uwe Baier, Timo Beller, and Enno Ohlebusch. Graphical pan-genome analysis with compressed suffix trees and the burrows–wheeler transform. *Bioinformatics*, 32(4):497–504, 2016.

3 Anton Bankevich, Sergey Nurk, Dmitry Antipov, Alexey A Gurevich, Mikhail Dvorkin, Alexander S Kulikov, Valery M Lesin, Sergey I Nikolenko, Son Pham, Andrey D Prjibelski, et al. Spades: a new genome assembly algorithm and its applications to single-cell sequencing. *Journal of computational biology*, 19(5):455–477, 2012.

4 Alexander Bowe, Taku Onodera, Kunihiko Sadakane, and Tetsuo Shibuya. Succinct de Bruijn graphs. In *International Workshop on Algorithms in Bioinformatics (WABI)*, pages 225–235. Springer, 2012.

5 Michael Burrows and David Wheeler. A block-sorting lossless data compression algorithm. In *Digital SRC Research Report*. Citeseer, 1994.

6 Rayan Chikhi, Antoine Limasset, Shaun Jackman, Jared T Simpson, and Paul Medvedev. On the representation of de Bruijn graphs. In *International conference on Research in computational molecular biology*, pages 35–55. Springer, 2014. URL: `https://github.com/jts/dbgfm`.

7 Rayan Chikhi, Antoine Limasset, and Paul Medvedev. Compacting de Bruijn graphs from sequencing data quickly and in low memory. *Bioinformatics*, 32(12):i201–i208, 2016. URL: `https://github.com/GATB/bcalm`.

8 Sebastian Deorowicz, Marek Kokot, Szymon Grabowski, and Agnieszka Debudaj-Grabysz. Kmc 2: fast and resource-frugal k-mer counting. *Bioinformatics*, 31(10):1569–1576, 2015.

9 Peter Elias. Efficient storage and retrieval by content and address of static files. *Journal of the ACM*, 21(2):246–260, 1974.

10 Robert Mario Fano. On the number of bits required to implement an associative memory. *Memorandum 61, Computer Structures Group, MIT*, 1971.

11 Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Proceedings 41st Annual Symposium on Foundations of Computer Science*, pages 390–398. IEEE, 2000.

12 Giuseppe Italiano, Nicola Prezza, Blerina Sinaimeri, and Rossano Venturini. Compressed weighted de Bruijn graphs. In *32nd Annual Symposium on Combinatorial Pattern Matching (CPM 2021)*, volume 191, pages 1–16, 2021. URL: `https://github.com/nicolaprezza/cw-dBg`.

13 Shaun D Jackman, Benjamin P Vandervalk, Hamid Mohamadi, Justin Chu, Sarah Yeo, S Austin Hammond, Golnaz Jahesh, Hamza Khan, Lauren Coombe, Rene L Warren, et al. Abyss 2.0: resource-efficient assembly of large genomes using a bloom filter. *Genome research*, 27(5):768–777, 2017.

14 Mikhail Karasikov, Harun Mustafa, Gunnar Rätsch, and André Kahles. Lossless indexing with counting de bruijn graphs. *bioRxiv*, 2021.

15 Parsoa Khorsand and Fereydoun Hormozdiari. Nebula: ultra-efficient mapping-free structural variant genotyper. *Nucleic acids research*, 49(8):e47–e47, 2021.

16 Guillaume Marçais and Carl Kingsford. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, 27(6):764–770, 2011.

17 Camille Marchet, Zamin Iqbal, Daniel Gautheret, Mikaël Salson, and Rayan Chikhi. Reindeer: efficient indexing of k-mer presence and abundance in sequencing datasets. *Bioinformatics*, 36(Supplement_1):i177–i185, 2020.

18 Shoshana Marcus, Hayan Lee, and Michael C Schatz. Splitmem: a graphical algorithm for pan-genome analysis with suffix skips. *Bioinformatics*, 30(24):3476–3483, 2014.

19 Prashant Pandey, Michael A Bender, Rob Johnson, and Rob Patro. deBGR: an efficient and near-exact representation of the weighted de Bruijn graph. *Bioinformatics*, 33(14):i133–i141, 2017.

20 Prashant Pandey, Michael A Bender, Rob Johnson, and Rob Patro. A general-purpose counting filter: Making every bit count. In *Proceedings of the 2017 ACM international conference on Management of Data*, pages 775–787, 2017.

21 Prashant Pandey, Michael A Bender, Rob Johnson, and Rob Patro. Squeakr: an exact and approximate k-mer counting system. *Bioinformatics*, 34(4):568–575, 2018.

22 Giulio Ermanno Pibiri. Sparse and Skew Hashing of K-Mers. *Bioinformatics. To Appear.*, xx(Supplement_xxx):xxx–yyy, 2022. URL: `https://doi.org/10.1101/2022.01.15.476199`.

23 Giulio Ermanno Pibiri and Roberto Trani. Parallel and external-memory construction of minimal perfect hash functions with PTHash. *CoRR*, abs/2106.02350, 2021. `arXiv:2106.02350`.

24 Giulio Ermanno Pibiri and Roberto Trani. PTHash: Revisiting FCH minimal perfect hashing. In *SIGIR '21: The 44th International ACM SIGIR Conference on Research and Development in Information Retrieval, Virtual Event, Canada, July 11-15, 2021*, pages 1339–1348, 2021.

25 Giulio Ermanno Pibiri and Rossano Venturini. Clustered Elias-Fano indexes. *ACM Trans. Inf. Syst.*, 36(1):2:1–2:33, 2017.

26 Giulio Ermanno Pibiri and Rossano Venturini. Efficient data structures for massive n-gram datasets. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 615–624, 2017.

27 Giulio Ermanno Pibiri and Rossano Venturini. Handling massive N-gram datasets efficiently. *ACM Trans. Inf. Syst.*, 37(2):25:1–25:41, 2019.

28 Giulio Ermanno Pibiri and Rossano Venturini. Techniques for inverted index compression. *ACM Comput. Surv.*, 53(6):125:1–125:36, 2021.

29 Amatur Rahman and Paul Medvedev. Representation of k-mer sets using spectrum-preserving string sets. In *International Conference on Research in Computational Molecular Biology*, pages 152–168. Springer, 2020. URL: `https://github.com/medvedevgroup/UST`.

30 Guillaume Rizk, Dominique Lavenier, and Rayan Chikhi. Dsk: k-mer counting with very low memory usage. *Bioinformatics*, 29(5):652–653, 2013.

31 Michael Roberts, Wayne Hayes, Brian R Hunt, Stephen M Mount, and James A Yorke. Reducing storage requirements for biological sequence comparison. *Bioinformatics*, 20(18):3363–3369, 2004.

32 Mirko Rossi, Mickael Santos Da Silva, Bruno Filipe Ribeiro-Gonçalves, Diogo Nuno Silva, Miguel Paulo Machado, Mónica Oleastro, Vítor Borges, Joana Isidro, Luis Viera, Jani Halkilahti, Anniina Jaakkonen, Federica Palma, Saara Salmenlinna, Marjaana Hakkinen, Javier Garaizar, Joseba Bikandi, Friederike Hilbert, and João André Carriço. INNUENDO whole genome and core genome MLST schemas and datasets for Salmonella enterica. July 2018. URL: `https://doi.org/10.5281/zenodo.1323684`.

33 Kristoffer Sahlin. Effective sequence similarity detection with strobemers. *Genome research*, 31(11):2080–2094, 2021.

34 Kristoffer Sahlin. Strobemers: an alternative to k-mers for sequence comparison. *bioRxiv*, 2021.

35 Yoshihiro Shibuya, Djamal Belazzougui, and Gregory Kucherov. Space-efficient representation of genomic k-mer count tables. In *International Workshop on Algorithms in Bioinformatics (WABI)*, volume 201, pages 8–1, 2021. URL: `https://github.com/yhhshb/locom`.

36 Yoshihiro Shibuya, Djamal Belazzougui, and Gregory Kucherov. Set-min sketch: a probabilistic map for power-law distributions with application to k-mer annotation. *Journal of Computational Biology*, 29(2):140–154, 2022.

37 Daniel S Standage, C Titus Brown, and Fereydoun Hormozdiari. Kevlar: a mapping-free framework for accurate discovery of de novo variants. *Iscience*, 18:28–36, 2019.

38 Derrick E Wood and Steven L Salzberg. Kraken: ultrafast metagenomic sequence classification using exact alignments. *Genome biology*, 15(3):1–12, 2014.

A Omitted Proofs from Section 4.1

▶ **Lemma 3.** Let us consider $d > 0$ equal nodes (w, x). If d is even, then the d nodes originate a path of either end-points (w, w) or (x, x). If d is odd, then the path has end-points (w, x).

Proof. We proceed by induction on d. Base case: if $d = 1$ (odd case), then there is only the singleton path (w, x); if $d = 2$ (even case), then we can either form the path $(w, x) \to (x, w)$ of end-points (w, w) or the path $(x, w) \to (w, x)$ of end-points (x, x). So the base case is verified. Now we assume the Lemma holds true for a generic $d > 2$ and we want to prove it for $d + 1$. If d is even, then $d + 1$ is odd and we can either have a path $(w, w) \to (w, x)$ or a path $(w, x) \to (x, x)$. In both cases the end-points are (w, x). Symmetrically: if d is odd, then $d + 1$ is even and we can either have a path $(w, x) \to (x, w)$ with end-points (w, w) or a path $(x, w) \to (w, x)$ with end-points (x, x). ◀

▶ **Lemma 6.** Given an incidence set I_w, if $n(I_w)$ is *odd* then only one path will contain w as end-point among all the paths that can be created from the nodes in I_w.

Proof. Let us first consider the special case where all the other weights in I_w are distinct, so there are no equal nodes in I_w except for, possibly, nodes of the form (w, w). In this case, we say that I_w is *canonical*. If there are some nodes (w, w), they can be trivially collapsed to a single path of end-points (w, w) by Lemma 3. So, without loss of generality, either we have *one* node (w, w) in I_w and $n(I_w) = |I_w| + 1$, or we do not and $n(I_w) = |I_w|$. In this special case, since $n(I_w)$ is odd, it is always possible to create $\frac{n(I_w)-1}{2}$ paths, each having 2 nodes. These paths will *not* contain w as end-point because all the end-points where w appears are used to link the nodes. Therefore, there will be exactly one unpaired node where w appears.

Now, observe that we can relax the restriction on the other weights to be all distinct. For every weight $x \neq w$ that appears for $d_x > 1$ times in I_w, there are d_x equal nodes (w, x). Let $D_x \subseteq I_w$ be the set of such nodes (hence, $d_x = |D_x|$). By applying Lemma 3 to the nodes of each set D_x:

- If d_x is even, then the nodes in D_x can be collapsed into a path of end-points (w, w) or (x, x). If the node (w, w) is created, we obtain a new incidence set $I'_w = I_w \setminus D_x \cup \{(w, w)\}$. Instead, if the node (x, x) is created, then $I'_w = I_w \setminus D_x$ since (x, x) cannot be in an incidence set for w. In both cases $n(I'_w)$ will still be odd since we subtract an even number from $n(I_w)$.
- If d_x is odd, the nodes are collapsed into the path of end-points (w, x) and the new incidence set is $I'_w = I_w \setminus D_x \cup \{(w, x)\}$. Again, $n(I'_w)$ will still be odd since we subtract an odd number from $n(I_w)$ but sum one.

After each set D_x is processed in this way, we are left with an incidence set for w that is canonical.

◀

▶ **Lemma 8.** $|W_{odd}|$ is even.

Proof. Observe that $\sum_{w \in eW} n(I_w^{max})$ is even and equal to $2m$ because we count the occurrences of the weights appearing as end-points of the sequences and each sequence has two end-points. Since $eW = eW_{odd} \cup eW_{even} \cup eW_{equal}$, the above sum can be re-written as

$$\sum_{w \in eW} n(I_w^{max}) = \sum_{w \in eW_{odd}} n(I_w^{max}) + \sum_{w \in eW_{even}} n(I_w^{max}) + \sum_{w \in eW_{equal}} n(I_w^{max}) = 2m.$$

It follows that also

$$\sum_{w \in eW_{odd}} n(I_w^{max}) = 2m - \sum_{w \in eW_{even}} n(I_w^{max}) - \sum_{w \in eW_{equal}} n(I_w^{max})$$

must be even since it is obtained by difference of even quantities. Since each term in the sum $\sum_{w \in eW_{odd}} n(I_w^{max})$ is odd by definition, the whole sum is even if and only if $|eW_{odd}|$ is even, as the sum of an odd number of odd numbers is odd. ◀

Table 5 The performance of Alg. 3 on the datasets Cod, Kestrel, Human, and Bacterial, for which we report the number of distinct k-mers (n) and the number of strings (m) after running UST [29] on the collections. The performance of the algorithm is expressed as: the number of actual runs (r) after the run-reduction optimization in comparison with the lower bound on the number of runs (r_{lo}) computed using Equation (1), and running time (in total sec and average ns/node).

Dataset	n	m	r_{lo}	r		Alg. 3 (sec)	Alg. 3 (ns/node)
Cod	502,465,200	2,406,681	4,183,202	4,183,230	(+0.00067%)	1.2	500
Kestrel	1,150,399,205	682,344	1,140,743	1,140,747	(+0.00035%)	0.3	440
Human	2,505,445,761	13,014,641	22,680,047	22,680,099	(+0.00023%)	7.5	580
Bacterial	5,350,807,438	26,448,286	56,662,230	56,662,304	(+0.00013%)	17.2	650

Table 6 The performance of w-SSHash on the permuted string collections Cod, Kestrel, Human, and Bacterial. We report the empirical entropy of the weights ($H_0(W)$), the dictionary space in average bits/k-mer, and query-time in average μsec/k-mer. The space is indicated as $x + y$, where x is the space of SSHash (without the weights) and y is the space for the encoding of the weights.

Dataset	$H_0(W)$	space		query-time
Cod	0.441	6.98+0.19	(2.35×)	1.3
Kestrel	0.089	6.49+0.02	(3.80×)	1.1
Human	0.453	8.28+0.22	(2.06×)	1.6
Bacterial	1.890	8.22+0.24	(7.81×)	1.9

B Additional Experimental Results

In Table 5 and Table 6 we report the performance of Alg. 3 and of w-SSHash on four additional, larger, collections that we also used in our previous work [22], namely the full genomes of *G. Morhua* (Cod), *F. Tinnunculus* (Kestrel), and *H. Sapiens* (Human), and a collection of more than 8000 bacterial genomes (Bacterial) [1]. Precisely, the results in Table 6 are for regular w-SSHash dictionaries with minimizer lengths equal to 17, 17, 20, and 20, for respectively, Cod, Kestrel, Human, and Bacterial.