



**HAL**  
open science

# Locality-Preserving Minimal Perfect Hashing of k-mers

Giulio Ermanno Pibiri, Yoshihiro Shibuya, Antoine Limasset

► **To cite this version:**

Giulio Ermanno Pibiri, Yoshihiro Shibuya, Antoine Limasset. Locality-Preserving Minimal Perfect Hashing of k-mers. 2022. hal-03832901

**HAL Id: hal-03832901**

**<https://hal.science/hal-03832901>**

Preprint submitted on 28 Oct 2022

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Locality-Preserving Minimal Perfect Hashing of $k$ -mers

Giulio Ermanno Pibiri<sup>1,2</sup>, Yoshihiro Shibuya<sup>3</sup>, and Antoine Limasset<sup>4</sup>

<sup>1</sup>Ca' Foscari University of Venice, Venice, Italy

<sup>2</sup>ISTI-CNR, Pisa, Italy

<sup>3</sup>University Gustave Eiffel, Marne-la-Vallée, France

<sup>4</sup>University of Lille and CNRS, Lille, France

## Abstract

Minimal perfect hashing is the problem of mapping a static set of  $n$  distinct keys into the address space  $\{1, \dots, n\}$  bijectively. It is well-known that  $n \log_2 e$  bits are necessary to specify a minimal perfect hash function  $f$ , when *no additional knowledge* of the input keys is to be used. However, it is often the case in practice that the input keys have intrinsic relationships that we can exploit to lower the bit complexity of  $f$ . For example, consider a string and the set of all its distinct sub-strings of length  $k$  – the so-called  $k$ -mers of the string. Two consecutive  $k$ -mers in the string have a strong intrinsic relationship in that they share an overlap of  $k - 1$  symbols. Hence, it seems intuitively possible to beat the classic  $\log_2 e$  bits/key barrier in this case. Moreover, we would like  $f$  to map consecutive  $k$ -mers to consecutive addresses, as to *preserve* as much as possible the relationships between the keys also in the co-domain  $\{1, \dots, n\}$ . This is a useful feature in practice as it guarantees a certain degree of *locality* of reference for  $f$ , resulting in a better evaluation time when querying consecutive  $k$ -mers from a string.

Motivated by these premises, we initiate the study of a new type of locality-preserving minimal perfect hash functions designed for  $k$ -mers extracted consecutively from a string (or collections of strings). We show a theoretic lower bound on the bit complexity of any  $(1 - \varepsilon)$ -locality-preserving MPHf, for a parameter  $0 < \varepsilon < 1$ . The complexity is lower than  $n \log_2 e$  bits for sufficiently small  $\varepsilon$ . We propose a construction that approaches the theoretic minimum space for growing  $k$  and present a practical implementation of the method. Lastly, we demonstrate the practical usefulness of our construction through experimentation: the functions can be several times smaller and even faster to query than the most efficient, albeit “general-purpose”, minimal perfect hash functions.

**Code Availability:** Our C++ implementation of the method is open-source and available on GitHub at <https://github.com/jermp/lphash>.

**Data Availability:** All the datasets used for the experiments of this work are available on Zenodo at <https://zenodo.org/record/7239205>.

**Keywords:** Locality-Preserving · Minimal Perfect Hashing · Spectrum-Preserving String Sets

# 1 Introduction

Given a universe set  $U$ , a function  $f : U \rightarrow [n] = \{1, \dots, n\}$  is a *minimal perfect hash function* (or, MPHf) for a set  $S \subseteq U$  with  $n = |S|$  if  $f(x) \neq f(y)$  for all  $x, y \in S$ ,  $x \neq y$ . In simpler words,  $f$  maps each key of  $S$  into a distinct integer in  $[n]$ . The function is allowed to return any value in  $[n]$  for a key  $x \in U \setminus S$ . Several authors proved that  $n \log_2 e = 1.442n$  bits are essentially necessary to represent such functions for  $|U| \gg n$  [32, 28]. Minimal perfect hashing is a central problem in data structure design and has received considerable attention, both in theory and practice. In fact, many practical constructions have been proposed [18, 6, 30, 20, 8, 4, 26, 15, 37]. These algorithms find MPHfs that take space close to the theoretic-minimum, e.g., 2 – 3 bits/key, retain very fast lookup time, and scale well to very large sets. Applications of minimal perfect hashing range from computer networks [27] to databases [10], as well as language models [38, 39, 45], compilers, and operating systems. MPHfs have been also used recently in Bioinformatics to implement fast and compact dictionaries for DNA strings [34, 33, 1, 31].

In its simplicity and versatility, the minimal perfect hashing problem does not take into account specific types of inputs, nor the intrinsic relationships between the input keys. Each key  $x \in S$  is considered independently from any other key in the set and, as such,  $\mathbb{P}[f(x) = i] \approx \frac{1}{n}$  for any fixed  $i \in [n]$ . In practice, however, the input keys often present some regularities that we could exploit to let  $f$  act “less randomly” on  $S$ . This, in turn, would permit to achieve a *lower* space complexity for  $f$ . We therefore consider in this paper the following special setting of the minimal perfect hashing problem: the elements of  $S$  are all the distinct sub-strings of length  $k$ , for some  $k > 0$ , from a given string  $X$  (or collection of strings). The elements of  $S$  are called  $k$ -mers. The crucial point is that any two consecutive  $k$ -mers in  $X$  have indeed a strong intrinsic relationship in that they share an overlap of  $k - 1$  symbols. It seems profitable to exploit the overlap information to *preserve* (as much as possible) the *local* relationship between consecutive  $k$ -mers as to *reduce* the randomness of  $f$ , thus lowering its bit complexity and evaluation time.

We are therefore interested in the design of a *locality-preserving* MPHf, or LP-MPHf, in the following sense: given a query sequence  $Q$ , if  $f(x) = j$  for some  $k$ -mer  $x \in Q$ , we would like  $f$  to hash  $\text{Next}(x)$  to  $j + 1$ ,  $\text{Next}(\text{Next}(x))$  to  $j + 2$ , and so on, where  $\text{Next}(x)$  is the  $k$ -mer following  $x$  in  $Q$  (and assuming these  $k$ -mers are also in  $X$ ). This behavior of  $f$  is very desirable in practice, at least for two important reasons. First, it implies *compression* for satellite values associated to  $k$ -mers. Typical satellite values are abundance counts, reference identifiers (sometimes called “colors”), or contig identifiers (e.g., unitigs) in a de Bruijn graph. Consecutive  $k$ -mers tend to have very similar – if not identical – satellite values, hence hashing consecutive  $k$ -mers to consecutive identifiers induce a natural clustering of the associated satellite values which is amenable to effective compression. The second important reason is, clearly, faster evaluation time when querying for consecutive  $k$ -mers in a sequence. This *streaming* query modality is the query modality employed by  $k$ -mer-based applications [1, 7, 31, 42, 34].

We formalize the notion of locality-preserving MPHf along with other preliminary definitions in Section 2. In Section 3 we show a lower bound to the bit complexity of any  $(1 - \varepsilon)$ -locality-preserving MPHf that depends on a parameter  $0 < \varepsilon < 1$ . The complexity is below the well-known bound of  $n \log_2 e$  bits for a “classic” MPHf, for sufficiently small  $\varepsilon$ . We then give in Section 4 a construction using *random minimizers* [43, 41] whose space approaches the lower bound by increasing  $k$ . The construction approximates a LP-MPHf for  $\varepsilon = 2/(w + 1)$ , for a sufficiently-large minimizer length  $m \leq k$ , where  $w = k - m + 1$ . The data structure is built in linear time in the size of the input (number of distinct  $k$ -mers). In Section 5 we present experiments across a breadth of datasets to show that the construction is practical too: the functions can be several times smaller and even faster to query than the most efficient, albeit general-purpose, minimal perfect hash functions. We conclude in Section 6 where we also sketch some future directions and open questions. Our C++ implementation of the method is publicly available at <https://github.com/jermp/lphash>.

## 2 Locality-Preserving Minimal Perfect Hashing of $k$ -mers

Let  $\mathcal{X}$  be a set of strings over an alphabet  $\Sigma$ . Throughout the paper we focus on the DNA alphabet  $\Sigma = \{A, C, G, T\}$  to better highlight the connection with our concrete application but our algorithms can be generalized to work for arbitrary alphabets. A sub-string of length  $k$  of a string  $S \in \mathcal{X}$  is called a  $k$ -mer of  $S$ .

**Definition 1** (Spectrum). *The  $k$ -mer spectrum of  $\mathcal{X}$  is the set of all distinct  $k$ -mers of the strings in  $\mathcal{X}$ . Formally:  $\text{spectrum}_k(\mathcal{X}) := \{x \in \Sigma^k \mid \exists S \in \mathcal{X} \text{ such that } x \text{ is a } k\text{-mer of } S\}$ .*

**Definition 2** (Spectrum-Preserving String Set). *A spectrum-preserving string set (or SPSS)  $\mathcal{S}$  of  $\mathcal{X}$  is a set of strings such that (i) each string of  $\mathcal{S}$  has length at least  $k$ , and (ii)  $\text{spectrum}_k(\mathcal{S}) = \text{spectrum}_k(\mathcal{X})$ .*

In particular, we are interested in a SPSS  $\mathcal{S}$  where each  $k$ -mer is seen only once, i.e., for each  $k$ -mer  $x \in \text{spectrum}_k(\mathcal{S})$  there is only one string of  $\mathcal{S}$  where  $x$  appears once. We assume that no  $k$ -mer appearing at the end of a string shares and overlap of  $k - 1$  symbols with the first  $k$ -mer of another string, otherwise we could reduce the number of strings in  $\mathcal{S}$  and obtain a smaller SPSS. In the following, we make use of this form of SPSS which is suitable for the minimal perfect hashing problem. We remark that efficient algorithms exist to compute such SPSSs (see, e.g., [40, 9, 25, 24]).

The input for our problem is therefore a SPSS  $\mathcal{S}$  for  $\mathcal{X}$  with  $|\mathcal{S}|$  strings and  $n > 1$  distinct  $k$ -mers. Without loss of generality we assume an order  $S_1, S_2, S_3, \dots$  of the strings of  $\mathcal{S}$  is fixed, so that also an order between the  $k$ -mers is fixed and we indicate with  $x_i$  the  $i$ -th  $k$ -mer in  $\mathcal{S}$  for  $i = 1, \dots, n$ .

**Definition 3** (Fragmentation Factor). *Given a SPSS  $\mathcal{S}$  with  $|\mathcal{S}|$  strings and  $n = |\text{spectrum}_k(\mathcal{S})|$  distinct  $k$ -mers, we define the fragmentation factor of  $\mathcal{S}$  as  $\alpha := \frac{|\mathcal{S}| - 1}{n}$ .*

The fragmentation factor of  $\mathcal{S}$  is a measure of how much contiguous the  $k$ -mers in  $\mathcal{S}$  are. The *minimum* fragmentation  $\alpha = 0$  is achieved for  $|\mathcal{S}| = 1$  and, in this case,  $x_i$  shares and overlap of  $k - 1$  symbols with  $x_{i+1}$  for *all*  $i = 1, \dots, n - 1$ . This ideal scenario is, however, unlikely to happen in practice. On the other hand, the worst-case scenario of maximum fragmentation  $\alpha = 1 - 1/n$  is achieved when  $|\mathcal{S}| = n$  and  $k$ -mers do not share any overlap. This is also unlikely to happen given that  $k$ -mers are extracted consecutively from the strings of  $\mathcal{X}$  and, as a result, many overlaps are expected. A more realistic scenario happens, instead, when  $|\mathcal{S}| \ll n$ , resulting in  $\varepsilon \gg \alpha$ . For the rest of the paper, we focus on this latter scenario to make our analysis meaningful.

We want to build a MPHf  $f : \Sigma^k \rightarrow [n]$  for  $\mathcal{S}$ , i.e., more precisely, for the  $n$  distinct  $k$ -mers in  $\text{spectrum}_k(\mathcal{S})$ . We remark again that our objective is to exploit the overlap of  $k - 1$  symbols between consecutive  $k$ -mers from a string of  $\mathcal{S}$  to preserve their locality, and *hence* reduce the bit complexity of  $f$  as well as its evaluation time when querying  $k$ -mers in sequence. We define a *locality-preserving* MPHf, or LP-MPHf, for  $\mathcal{S}$  as follows.

**Definition 4** (LP-MPHf). *Let  $f : \Sigma^k \rightarrow [n]$  be a MPHf for  $\mathcal{S}$ . If*

$$\mathbb{P}[f(x_{i+1}) = f(x_i) + 1 \text{ and } x_i, x_{i+1} \text{ are from the same string of } \mathcal{S}] = 1 - \varepsilon,$$

for  $i = 1, \dots, n - 1$ , then  $f$  is a  $(1 - \varepsilon)$ -locality-preserving MPHf for  $\mathcal{S}$ .

Let  $E_1$  be the event “ $f(x_{i+1}) = f(x_i) + 1$ ” and  $E_2$  be the event “ $x_i, x_{i+1}$  are from the same string of  $\mathcal{S}$ ”. Then  $\mathbb{P}[E_1 \text{ and } E_2] = 1 - \varepsilon \iff \varepsilon = 1 - \mathbb{P}[E_1 \text{ and } E_2] = \mathbb{P}[\neg E_1 \text{ or } \neg E_2]$  by De Morgan’s laws. Therefore we have

$$\varepsilon = \mathbb{P}[\neg E_1 \text{ or } \neg E_2] = \mathbb{P}[f(x_{i+1}) \neq f(x_i) + 1 \text{ or } x_i, x_{i+1} \text{ are not from the same string of } \mathcal{S}].$$

**Lemma 1.** *Let  $\mathcal{S}$  be a SPSS with fragmentation factor  $\alpha$  and  $f$  be a  $(1 - \varepsilon)$ -locality-preserving MPHf for  $\mathcal{S}$ . Then  $\varepsilon \geq \alpha$ .*

*Proof.*  $\varepsilon = \mathbb{P}[\neg E_1 \text{ or } \neg E_2] \geq \mathbb{P}[\neg E_2] = \mathbb{P}[x_i, x_{i+1} \text{ are not from the same string of } \mathcal{S}] = (|\mathcal{S}| - 1)/n = \alpha$ , since there are exactly  $|\mathcal{S}| - 1$  consecutive pairs  $(x_i, x_{i+1})$  of  $k$ -mers in  $\mathcal{S}$  that are not in the same string ( $x_i$  at the end of some string  $S_j$  and  $x_{i+1}$  at the beginning of the string  $S_{j+1}$ ).  $\square$

Lemma 1 says that any LP-MPHf for  $\mathcal{S}$  is at best  $(1 - \alpha)$ -locality-preserving<sup>1</sup> and how small  $\varepsilon$  can be actually depends on the input SPSS (and on the strategy used to implement  $f$  in practice, as we are going to illustrate in Section 4). In fact, for the worst-case input – a SPSS with maximum fragmentation factor – we immediately obtain the following Corollary.

**Corollary 1.** *Let  $\mathcal{S}$  be a SPSS with maximum fragmentation factor. Then any LP-MPHf for  $\mathcal{S}$  is 0-locality-preserving.*

Intuitively, the “best” LP-MPHf for  $\mathcal{S}$  is the one having the smallest probability  $\varepsilon$ , so we look for practical constructions with small  $\varepsilon$ .

On the other hand, note that a “classic” MPHf is obtained when the locality-preserving property is almost always not satisfied and, as a consequence,  $\mathbb{P}[f(x_{i+1}) = j + 1 | f(x_i) = j] = \mathbb{P}[f(x_{i+1}) = j + 1] \approx \frac{1}{n}$  for any  $j \in [n]$ , *regardless of* the overlap between  $x_i$  and  $x_{i+1}$ .

<sup>1</sup>For example, SShash [34, 33] is a  $(1 - \alpha)$ -locality-preserving MPHf. It is also able of detecting alien  $k$ -mers, so its space usage is larger than that of the functions described in this work.

**Corollary 2.** *Let  $\mathcal{S}$  be a SPSS with fragmentation factor  $\alpha$ . Then any “classic” MPHf for  $\mathcal{S}$  is  $\epsilon(1-\alpha)$ -locality-preserving for a very small  $\epsilon \geq 0$ .*

### 3 Information-Theoretic Lower Bound

In this section we address the following central question: What is the information-theoretic lower bound for  $f$ ? As we already motivated, we expect  $f$  to require *less* space than  $\log_2 e$  bits/ $k$ -mer as for a “classic” MPHf – at least for sufficiently small  $\epsilon$  – thanks to the overlaps between consecutive  $k$ -mers on  $\mathcal{S}$ .

**Theorem 1.** *Any MPHf  $f : \Sigma^k \rightarrow [n]$  that is  $(1-\epsilon)$ -locality-preserving for a SPSS  $\mathcal{S}$  needs at least*

$$n \cdot \left( \epsilon \log_2(e/\epsilon) - (1-\epsilon) \log_2(1-\epsilon) \right) \text{ bits} \quad (1)$$

to be described, with  $|\text{spectrum}_k(\mathcal{S})| = n$ .

*Proof.* We first sketch our proof strategy. Let  $I$  be the number of choices for  $\mathcal{S}$ . Suppose that  $f$  is a LP-MPHf for at most  $Z$  distinct SPSSs. Then the number of distinct functions is at most  $I/Z$  and we need at least  $\log_2(I/Z)$  bits to distinguish them. In the remainder of the proof we therefore focus on determining the ratio  $I/Z$ .

By Definition 4, any function  $f$  induces a partition of the  $k$ -mers of  $\mathcal{S}$  into  $\epsilon n$  sub-strings and the  $k$ -mers  $x_1, \dots, x_{|s|-k+1}$  of each sub-string  $s$  are such that  $f(x_{i+1}) = f(x_i) + 1$  for  $i = 1, \dots, |s| - k$ . By binning these  $\epsilon n$  sub-strings with respect to their first  $k$ -mers, we have that the probability of hashing the  $n$   $k$ -mers of  $\mathcal{S}$  without collisions is  $P_f = \frac{(\epsilon n)!}{(\epsilon n)^{(\epsilon n)}}$ . Now, taking into account that any  $f$  induces a sub-string length distribution, it then follows that the number of SPSSs  $\mathcal{S}$  whose  $n$   $k$ -mers are hashed without collisions is  $Z = (P_f \cdot I) / (\epsilon \cdot \binom{n}{\epsilon n})$  where  $\epsilon \cdot \binom{n}{\epsilon n}$  is the number of possible ways of partitioning  $\mathcal{S}$  into  $\epsilon n$  sub-strings. In fact, the total number of  $k$ -mers in the  $\epsilon n$  sub-strings is  $n$  and each sub-string contains at least 1  $k$ -mer. Hence, the number of ways of selecting  $\epsilon n$  non-zero integers whose sum is  $n$  is equivalent to the number of distinct bit-vectors of length  $n$  with  $\epsilon n$  bits set where the last bit (that in position  $n$ ) is always set (because the sum must be  $n$ ) and where each integer is given by the difference between the position of two consecutive bits set. So there are  $\binom{n-1}{\epsilon n-1} = \epsilon \cdot \binom{n}{\epsilon n}$  such bit-vectors.

In conclusion, the space lower bound is  $\log_2(I/Z) = \log_2(\epsilon \cdot \binom{n}{\epsilon n} / P_f)$  bits plus a global redundancy of  $\log_2 n$  bits to encode the value of  $n$ . Using Stirling’s approximation of factorial and simplifying, we obtain

$$\begin{aligned} \log_2(I/Z) + \log_2 n &= \log_2 \left( \epsilon \cdot \binom{n}{\epsilon n} / P_f \right) + \log_2 n \\ &= \log_2 \epsilon + n \log_2 \left( \frac{1}{1-\epsilon} \right) - \epsilon n \log_2 \left( \frac{\epsilon}{1-\epsilon} \right) + \frac{1}{2} \left( \log_2 \epsilon - \log_2 n - \log_2(1-\epsilon) + \log_2(2\pi) \right) \\ &\quad + \epsilon n \log_2 e - \frac{1}{2} \left( \log_2 \epsilon + \log_2 n + \log_2(2\pi) \right) + \log_2 n \\ &= n \cdot \left( \epsilon \log_2(e/\epsilon) - (1-\epsilon) \log_2(1-\epsilon) \right) + \log_2 \epsilon - \frac{1}{2} \log_2(1-\epsilon) \\ &\approx n \cdot \left( \epsilon \log_2(e/\epsilon) - (1-\epsilon) \log_2(1-\epsilon) \right) \text{ bits.} \end{aligned}$$

□

Let  $b(\epsilon) = \epsilon \log_2(e/\epsilon) - (1-\epsilon) \log_2(1-\epsilon)$ , for  $0 < \epsilon < 1$ , so that the lower bound of Theorem 1 can be written as  $n \cdot b(\epsilon)$  bits. It is easy to see that  $\lim_{\epsilon \rightarrow 1} b(\epsilon) = \log_2 e$ , so one would essentially obtain a classic MPHf for large values of  $\epsilon$ . In Figure 1 we plot the function  $b(\epsilon)$ . By taking

$$\frac{\partial b(\epsilon)}{\partial \epsilon} = 0 \iff \log_2(e/\epsilon \cdot (1-\epsilon)) = 0 \implies \epsilon = \frac{e}{1+e}$$

we see that the function is maximum in  $\epsilon = \frac{e}{1+e}$  and the value is  $b\left(\frac{e}{1+e}\right) = \frac{e}{e+1} \log_2(e+1) - \frac{1}{e+1} \log_2\left(\frac{1}{e+1}\right) \approx 1.895$  which is larger than  $\log_2 e$ . This is no surprise because part of the complexity of the function is due to the distribution of sub-string lengths which vanishes for  $\epsilon \rightarrow 1$ . Instead, it can be derived that  $b(\epsilon) < \log_2 e$  for any  $0 < \epsilon < 0.3516$ .

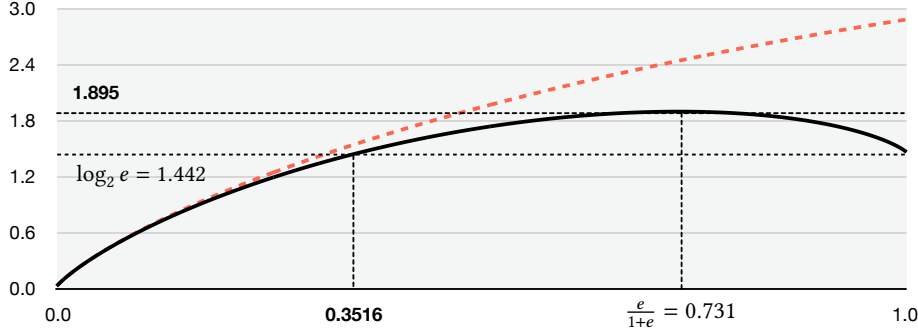


Figure 1: The function  $b(\varepsilon) = \varepsilon \log_2(e/\varepsilon) - (1 - \varepsilon) \log_2(1 - \varepsilon)$ , for  $0 < \varepsilon < 1$ , representing the average number of bits/ $k$ -mer spent by the lower bound in Theorem 1. The dashed red line is the function  $b^*(\varepsilon) = \varepsilon \log_2(e^2/\varepsilon)$ , which is a tight approximation of  $b(\varepsilon)$  for small values of  $\varepsilon$ , e.g., when  $\varepsilon < 0.3516$ .

## 4 Construction

In this section we describe an algorithm to find  $(1 - \varepsilon)$ -locality-preserving MPHFs whose space approaches the theoretic minimum (Theorem 1) for growing  $k$ .

We first need the following definitions.

**Definition 5** (Random Minimizer). *Given a  $k$ -mer  $x$  and a random hash function  $h$ , the minimizer of  $x$  is the  $m$ -mer  $\mu$  such that  $h(\mu) \leq h(y)$  for any other  $m$ -mer  $y$  of  $x$ , for some  $m \leq k$ .*

For convenience, we indicate with  $w = k - m + 1$  the number of  $m$ -mers in a  $k$ -mer. Since  $h$  is a random hash function<sup>2</sup>, each  $m$ -mer in a  $k$ -mer has probability  $\frac{1}{w}$  of being the minimizer of the  $k$ -mer. We say that the triple  $(k, m, h)$  defines a random minimizer scheme. The *density* of a minimizer scheme is the expected number of selected minimizers from the input.

**Definition 6** (Super- $k$ -mer). *Given a string  $S$ , a super- $k$ -mer is a maximal sub-string of  $S$  where each  $k$ -mer has the same minimizer.*

The construction algorithm builds upon the following main insight. Let  $g$  be a super- $k$ -mer and assume  $g$  is the only super- $k$ -mer whose minimizer is  $\mu$ . By definition of super- $k$ -mer, all the  $k$ -mers  $x_{g,1}, \dots, x_{g,|g|-k+1}$  in  $g$  contain the minimizer  $\mu$  as a sub-string. If  $p_{g,1}$  is the start position of  $\mu$  in the first  $k$ -mer  $x_{g,1}$  of  $g$ , then

$$p_{g,i} = p_{g,1} - i + 1 \quad (2)$$

is the start position of  $\mu$  in  $x_{g,i}$  (the  $i$ -th  $k$ -mer of  $g$ ) for  $1 \leq i \leq |g| - k + 1$ . Figure 2 gives a practical example for a super- $k$ -mer  $g$  of length 16 and  $k = 13$ . The next property illustrates the relation between the size  $|g| - k + 1$  and the position  $p_{g,1}$  (we will come later on the implications of this Property).

**Property 1.**  $|g| - k + 1 \leq p_{g,1} \leq w$  for any super- $k$ -mer  $g$ .

*Proof.* Since  $p_{g,1}$  is the start position of the minimizer in the first  $k$ -mer of  $g$ , there are at most  $p_{g,1}$   $k$ -mers that contain the minimizer as a sub-string, hence  $|g| - k + 1 \leq p_{g,1}$ . However,  $g$  cannot contain more than  $w$   $k$ -mers.  $\square$

Now, suppose we are given a positive query  $k$ -mer  $x$  whose minimizer is  $\mu$ . The  $k$ -mer must appear as a sub-string of  $g$ , i.e., it must be one among  $x_{g,1}, \dots, x_{g,|g|-k+1}$ . We want to compute the rank of  $x$  among the  $k$ -mers  $x_{g,1}, \dots, x_{g,|g|-k+1}$  of  $g$ , which we indicate by  $\text{Rank}(x)$  (assuming that it is clear from the context that  $\text{Rank}$  is relative to  $g$ ). We can use the positional information given by  $\mu$  to compute  $\text{Rank}(x)$  as follows. Let  $p$  be the start position of  $\mu$  in  $x$ : if  $p_{g,1} \geq p$  and  $1 \leq p_{g,1} - p + 1 \leq |g| - k + 1$ , then

$$\text{Rank}(x) = p_{g,1} - p + 1 \quad (3)$$

otherwise ( $p_{g,1} < p$  or  $p_{g,1} - p + 1 > |g| - k + 1$ ),  $x$  cannot possibly be in  $g$  and, hence, indexed by  $f$ . Our strategy is to compute  $f(x_{g,i})$  as

$$f(x_{g,i}) = f(x_{g,1}) + \text{Rank}(x_{g,i}) - 1 = f(x_{g,1}) + p_{g,1} - p_{g,i} \quad (4)$$

<sup>2</sup>In practice,  $h$  will be an instance of MurmurHash [2] or xxHash [13].

$g$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$x_{g,1}$	1	2	3	4	5	6	7	8	9	10	11	12	13	$p_{g,1} = 6$		
$x_{g,2}$	1	2	3	4	5	6	7	8	9	10	11	12	13	$p_{g,2} = 5$		
$x_{g,3}$	1	2	3	4	5	6	7	8	9	10	11	12	13	$p_{g,3} = 4$		
$x_{g,4}$	1	2	3	4	5	6	7	8	9	10	11	12	13	$p_{g,4} = 3$		

Figure 2: A super- $k$ -mer  $g$  of length 16 with  $|g| - k + 1 = 16 - 13 + 1 = 4$   $k$ -mers  $x_{g,1}, x_{g,2}, x_{g,3}, x_{g,4}$  for  $k = 13$  and minimizer length  $m = 7$ . The shaded boxes highlight the minimizer whose start position is  $p_{g,i}$  in  $k$ -mer  $x_{g,i}$ . It is easy to see that  $i = p_{g,1} - p_{g,i} + 1$  for any  $1 \leq i \leq |g| - k + 1$ .

for any  $k$ -mer  $x_{g,1}, \dots, x_{g,|g|-k+1}$  of  $g$ . Next, we show in Lemma 2 that this strategy maps the  $k$ -mers  $x_{g,1}, \dots, x_{g,|g|-k+1}$  bijectively in  $\{(f(x_{g,1}) - 1) + 1, \dots, (f(x_{g,1}) - 1) + |g| - k + 1\}$  and preserves their locality.

**Lemma 2.** *The strategy in Equation 4 guarantees  $f(x_{g,i+1}) = f(x_{g,i}) + 1$  for any  $i = 1, \dots, |g| - k$ .*

*Proof.* For Equation 4,  $f(x_{g,i}) = f(x_{g,1}) + p_{g,1} - p_{g,i}$ . Therefore  $f(x_{g,i+1}) = f(x_{g,1}) + p_{g,1} - p_{g,i+1}$ . Since  $p_{g,i+1} = p_{g,i} - 1$  for Equation 2, then  $f(x_{g,i+1}) = f(x_{g,1}) + p_{g,1} - p_{g,i+1} = f(x_{g,1}) + p_{g,1} - p_{g,i} + 1 = f(x_{g,i}) + 1$ .  $\square$

## 4.1 Data Structure

From Equation 4 is evident that  $f(x_{g,1})$  acts as a “global” component in the calculation of  $f(x_{g,i})$ , which must be added to a “local” component represented by  $\text{Rank}(x_{g,i})$ . We have shown how to compute  $\text{Rank}(x_{g,i})$  (Equation 3). Moreover, Lemma 2 guarantees that the local rank bijectively maps the  $k$ -mers of  $g$  into  $[1..|g| - k + 1]$ . We are therefore left to show how to compute  $f(x_{g,1})$  for each super- $k$ -mer  $g$ . We proceed as follows.

Let  $\mathcal{M}$  be the set of all the distinct minimizers of  $\mathcal{S}$ . We build a MPHf for  $\mathcal{M}$ ,  $f_m : \Sigma^m \rightarrow [|\mathcal{M}|]$ . Assume, for ease of exposition, that each super- $k$ -mer  $g$  is the only super- $k$ -mer having minimizer  $\mu$ . (We explain how to handle the case where more super- $k$ -mers have the same minimizer in Section 4.3.) We allocate an array  $L[1..|\mathcal{M}| + 1]$  where  $L[1] = 0$  and  $L[f_m(\mu) + 1] = |g| - k + 1$ . We then take the prefix-sums of  $L$ , that is, we replace  $L[i]$  with  $\sum_{j=1}^i L[j]$  for all  $i = 2, \dots, |\mathcal{M}| + 1$ . After this transformation,  $L[f_m(\mu)]$  indicates that there are  $L[f_m(\mu)]$   $k$ -mers before those in  $g$  (whose minimizer is  $\mu$ ) in the order given by  $f_m$ . The size of  $g$  can be recovered as  $L[f_m(\mu) + 1] - L[f_m(\mu)] = |g| - k + 1$ . In conclusion, we compute  $f(x_{g,1})$  as  $f(x_{g,1}) = L[f_m(\mu)]$ . The positions  $p_1$  of each super- $k$ -mer  $g$  are instead written in another array  $P[1..|\mathcal{M}|]$  where  $P[f_m(\mu)] = p_1$ . It follows that the data structure is built in  $O(n)$  time, since a scan over the input suffices to compute all super- $k$ -mers and  $f_m$  can be built in  $O(|\mathcal{M}|)$  expected time.

With these three components –  $f_m$ , and the two arrays  $L$  and  $P$  – it is easy to evaluate  $f(x)$  as shown in Algorithm 1. The complexity of the algorithm is  $O(w)$  since that is the time for computing the minimizer<sup>3</sup> and the evaluation of  $f_m$ , as well as accessing the arrays, takes  $O(1)$ .

The data structure for  $f$  is itself a compressed representation for  $f_m$ ,  $L$ , and  $P$ . To compute the space taken by the data structure we first need to know  $|\mathcal{M}|$ , the expected number of distinct minimizers seen in the input. If  $d$  indicates the density of a random minimizer scheme, then (i)  $|\mathcal{M}| = dn$ , and (ii)  $\varepsilon = d$  for Lemma 2. In particular, a result due to Zheng et al. [47, Theorem 3] allows us to compute  $d$  for a random minimizer scheme as  $d = \frac{2}{w+1} + o(1/w)$  if  $m > (3 + \varepsilon) \log_4(w + 1)$  for any  $\varepsilon > 0$ . (We will always operate under the condition that  $m$  is sufficiently large compared to  $k$  otherwise minimizers are meaningless.)

Therefore any random minimizer scheme gives us a  $(1 - \varepsilon)$ -LP MPHf with  $\varepsilon = \frac{2}{w+1}$ , where  $w = k - m + 1$  (we omit lower order terms for simplicity). Replacing this value of  $\varepsilon$  into Theorem 1, we derive a theoretic minimum space for  $f$  of  $n \cdot \left(\frac{2}{w+1} \cdot \log_2\left(e \cdot \frac{w+1}{2}\right) - \frac{w-1}{w+1} \log_2\left(\frac{w-1}{w+1}\right)\right)$  bits, which is

$$n \cdot \frac{2}{w+1} \cdot \log_2\left(e^2 \cdot \frac{w+1}{2}\right) \text{ bits} \quad (5)$$

<sup>3</sup>Considering each hash calculation  $h(\cdot)$  as  $O(1)$ .



---

**Algorithm 1:** Evaluation algorithm for  $f$ , given the  $k$ -mer  $x$ . The helper function  $\text{minimizer}(x)$  computes the minimizer  $\mu$  of  $x$  and the starting position  $p$  of  $\mu$  in  $x$ .

---

```

1  $f(x)$  :
2    $(\mu, p) = \text{minimizer}(x)$ 
3    $i = f_m(\mu)$ 
4   return  $L[i] + P[i] - p$ 

```

---



---

**Algorithm 2:** Evaluation algorithm for a partitioned representation of  $f$ . The quantities  $n_{lr}$ ,  $n_l$ ,  $n_r$ , and  $n_n$  are, respectively, the number of left-right-max, left-max, right-max, and non-max super- $k$ -mers of  $\mathcal{S}$ .

---

```

1  $f(x)$  :
2    $(\mu, p) = \text{minimizer}(x)$ 
3    $i = f_m(\mu)$ 
4    $t = R[i]$ 
5    $j = \text{Rank}_t(i)$ 
6    $prefix = 0, offset = 0, p_1 = 0$ 
7   switch( $t$ ):
8     case left-right-max:
9        $prefix = 0, offset = (j - 1)w, p_1 = w$ 
10    case left-max:
11       $prefix = n_{lr}, offset = L_l[j], p_1 = L_l[j + 1] - L_l[j]$ 
12    case right-max:
13       $prefix = n_{lr} + n_l, offset = L_r[j], p_1 = w$ 
14    case non-max:
15       $prefix = n_{lr} + n_l + n_r, offset = L_n[j], p_1 = P_n[j]$ 
16  return  $prefix + offset + p_1 - p$ 

```

---

for small  $\varepsilon$  (see dashed red line in Figure 1). Our construction, instead, achieves the following space usage (see the supplementary material for the proof).

**Theorem 2.** *Given a random minimizer scheme  $(k, m, h)$  with  $m > (3 + \varepsilon) \log_4(w + 1)$  for any  $\varepsilon > 0$  and  $w = k - m + 1$ , there exists a  $(1 - \varepsilon)$ -LP MPHf for a SPSS  $\mathcal{S}$  which takes*

$$n \cdot \frac{2}{w+1} \cdot \left( \log_2 \left( 4e \cdot (w+1)^2 \right) + o(1) \right) \text{ bits} \quad (6)$$

with  $\varepsilon = \frac{2}{w+1}$  and  $n = |\text{spectrum}_k(\mathcal{S})|$ .

Comparing the space bound in Theorem 2 with the lower bound in Formula 5 and omitting lower order terms, it is easy to derive that our construction costs  $\frac{2}{w+1} \cdot (3 + \log_2(\frac{w+1}{e}))$  bits/ $k$ -mer more than the theoretic minimum. However this surplus diminishes as  $w$  grows; for example, when  $m$  is fixed and  $k$  grows. Next we show how to improve the result of Theorem 2.

## 4.2 Partitioned Data Structure

Property 2 states that  $|g| - k + 1 \leq p_{g,1} \leq w$  for any super- $k$ -mer  $g$ . As an immediate implication we have that if  $|g| - k + 1 = w$  then also  $p_{g,1} = w$  (and, symmetrically, if  $p_{g,1} = 1$  then  $|g| = k$ ). This suggests that, whenever a super- $k$ -mer contains a *maximal* number of  $k$ -mers,  $|g| - k + 1 = p_{g,1} = w$  can be implicitly derived. We can thus save the space for the entries dedicated to such super- $k$ -mers in the arrays  $L$  and  $P$ . Note that the converse is not true in general, i.e., if  $p_{g,1} = w$  it could be that  $|g| - k + 1 < w$ . Nonetheless, we can still save space for some entries of  $P$  in this case.

Based on the starting position of the minimizer in the *first* and *last*  $k$ -mer of a super- $k$ -mer, we distinguish between four *types* of super- $k$ -mers. See Figure 3 for an example.

**Definition 7** (FL rule). *Let  $g$  be a super- $k$ -mer. The first/last (FL) rule is as follows:*

- if  $p_{g,1} = w$  and  $p_{g,|g|-k+1} = 1$ , then  $g$  is a left-right-max super- $k$ -mer; else
- if  $p_{g,1} < w$  and  $p_{g,|g|-k+1} = 1$ , then  $g$  is a left-max super- $k$ -mer; else
- if  $p_{g,1} = w$  and  $p_{g,|g|-k+1} > 1$ , then  $g$  is a right-max super- $k$ -mer; else
- if  $p_{g,1} < w$  and  $p_{g,|g|-k+1} > 1$ , then  $g$  is a non-max super- $k$ -mer.

We store the type of each super- $k$ -mer in an array  $R[1..|\mathcal{M}|]$ , in the order given by  $f_m$ . We can now exploit this labeling of super- $k$ -mers to improve the space bound of Theorem 2 because: (i) for all left-right-max super- $k$ -mers, we do not store  $L$  nor  $P$ ; (ii) for all left/right-max super- $k$ -mers, we only store  $L$  – precisely, two arrays  $L_l$  and  $L_r$  for left- and right-max super- $k$ -mers respectively; (iii) for all



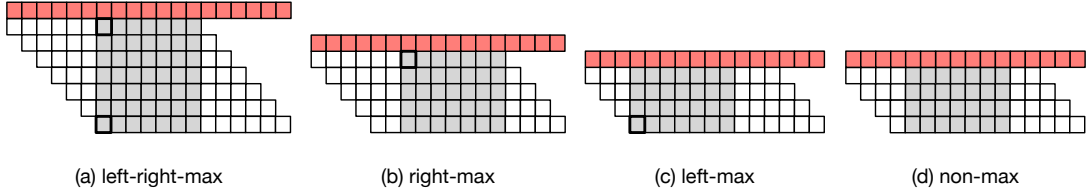


Figure 3: The four different types of super- $k$ -mers. The example is for  $k = 13$  and minimizer length  $m = 7$ , so  $w = k - m + 1 = 13 - 7 + 1 = 7$ . The shaded boxes highlight the minimizer whose start position is marked when it is either max (7), or min (1).

the other super- $k$ -mers, i.e., non-max, we store both  $L$  and  $P$  as explained before – let us indicate them with  $L_n$  and  $P_n$  in the following.

Addressing the arrays  $L_l$ ,  $L_r$ ,  $L_n$  and  $P_n$ , can be achieved by answering  $\text{Rank}_t(i)$  queries on  $R$ : the result of this query is the number of super- $k$ -mers that have type  $t$  in the prefix  $R[1..i]$ . If  $i = f_m(\mu)$ , then we read the type of the super- $k$ -mer associated to  $\mu$  as  $t = R[i]$ . Then we compute  $j = \text{Rank}_t(i)$ . Depending on the type  $t$ , we have to access the  $j$ -th position of either  $L_l$ , or  $L_r$ , or  $L_n$  and  $P_n$ . A succinct representation of  $R$  that also supports  $\text{Rank}_t(i)$  and  $\text{Access}(i)$  queries is the wavelet tree [21]. In our case, we only have four possible types, so the wavelet tree represents  $R$  in  $2|\mathcal{M}| + o(|\mathcal{M}|)$  bits<sup>4</sup> and supports both queries in  $O(1)$  time. The wavelet tree is also built in linear time, so the overall building time remains  $O(n)$ . Algorithm 2 shows the evaluation for this partitioned representation of  $f(x)$ . Also this algorithm executes in  $O(w)$  time.

Intuitively, if the fraction of left-right-max super- $k$ -mers and that of left/right-max super- $k$ -mers is sufficiently high, we can save significant space compared to the previous data structure which stores both  $L$  and  $P$  for all minimizers. Indeed, we obtain the following Theorem (see the supplementary material for the proof).

**Theorem 3.** *Given a random minimizer scheme  $(k, m, h)$  with  $m > (3 + \epsilon) \log_4(w + 1)$  for any  $\epsilon > 0$  and  $w = k - m + 1$ , there exists a  $(1 - \epsilon)$ -LP MPHf for a SPSS  $\mathcal{S}$  which takes*

$$n \cdot \frac{2}{w+1} \cdot \left( \log_2 \left( e \frac{16 \cdot 2^{1/4}}{3} \cdot (w+1) \right) + o(1) \right) \text{ bits} \quad (7)$$

with  $\epsilon = \frac{2}{w+1}$  and  $n = |\text{spectrum}_k(\mathcal{S})|$ .

Comparing this space bound with the lower bound in Formula 5 and omitting lower order terms, we see that this construction costs  $\frac{2}{w+1} \cdot \log_2 \left( \frac{32 \cdot 2^{1/4}}{3e} \right)$  bits/ $k$ -mer more than the theoretic minimum. Also this surplus tends to vanish as  $k$  grows but faster than that from Theorem 2. Theorem 3 saves a factor of  $\frac{2}{5} \cdot (3 - \log_2 e + \log_2(w + 1))$  bits/ $k$ -mer from the space of Theorem 2.

### 4.3 Ambiguous Minimizers

Let  $G_\mu$  be the set of super- $k$ -mers whose minimizer is  $\mu$ . The rank computation in Equation 3 can be used as long as  $|G_\mu| = 1$ , i.e., whenever one single super- $k$ -mer  $g$  has minimizer  $\mu$  and, thus, the single  $p_{g,1}$  unequivocally displace all the  $k$ -mers  $x_{g,1}, \dots, x_{g,|g|-k+1}$ . When  $|G_\mu| > 1$  we say that the minimizer  $\mu$  is “ambiguous”. It is a known fact that the number of such minimizers is small for a sufficiently long minimizer length  $m$  [34, 23, 11] (for example, on the datasets used in Section 5, the fraction of ambiguous minimizers is in between 1% and 4%). However, they must be dealt with in some way.

Let  $\xi$  be the fraction of  $k$ -mers whose minimizers are ambiguous. Our strategy is to build a fall-back MPHf for these  $k$ -mers. This function adds  $\xi \log_2 e$  bits/ $k$ -mer on top of the space of Theorem 2 and 3, making our functions be  $(1 - \xi)(1 - \epsilon)$ -locality-preserving. To detect an ambiguous minimizer  $\mu$ , we use the following trick: we set  $L[f_m(\mu)] = 0$  for the un-partitioned data structure from Section 4.1 and  $L_r[f_m(\mu)] = 0$  for the partitioned variant from Section 4.2. Therefore, with just an extra check on the super- $k$ -mer size we know if the query  $k$ -mer must be looked-up in the fall-back MPHf or not.

<sup>4</sup>The  $o(|\mathcal{M}|)$  term is the redundancy needed to accelerate the binary rank queries. In practice, the term  $o(|\mathcal{M}|)$  can be non-negligible, e.g., can be as high as  $2 \cdot (|\mathcal{M}|/4)$  bits using the Rank9 index [46, Sec. 3], but it is necessary for fast queries in practice (namely,  $O(1)$  time). Looking at Table 1a from [35], we see that the redundancy is in between 3% and 25% of  $2|\mathcal{M}|$ .

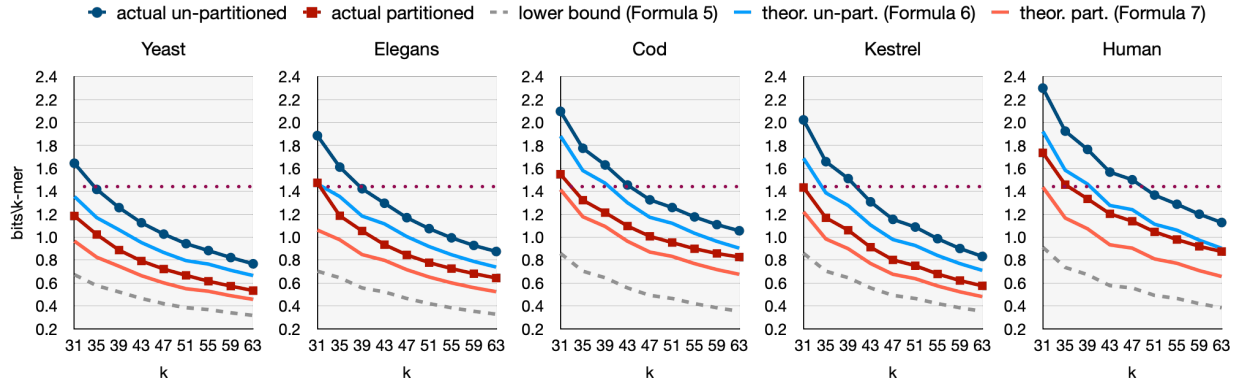


Figure 4: Space in average bits/ $k$ -mer for LPHash by varying  $k$ , for both un-partitioned and partitioned data structures. As reference points, we report that PTHash takes 2.76, 2.68, 2.65, 2.58, and 2.65 bits/ $k$ -mer on Yeast, Elegans, Cod, Kestrel, and Human respectively (2.6-2.8 bits/ $k$ -mer); BBHash takes 3.06 bits/ $k$ -mer across all datasets. The space for PTHash and BBHash does not depend on  $k$ . The dotted line at  $\log_2 e = 1.442$  bits/ $k$ -mer indicates the classic MPHF lower-bound.

We leave the exploration of alternative strategies to handle ambiguous minimizers to future work. For example, one can imagine a recursive data structure where, similarly to [44], each level is an instance of the construction with different minimizer lengths: if level  $i$  has minimizer length  $m_i$ , then level  $i + 1$  is built with length  $m_{i+1} > m_i$  over the  $k$ -mers whose minimizers are ambiguous at level  $i$ .

## 5 Experiments

In this section we show that the data structure described in Section 4 is practical too. Our implementation of the method is in C++ and we refer to it as LPHash in the following. We compare its space usage, query time, and building time against the fastest MPHF, PTHash [36, 37], and the popular BBHash [26]. Both competitors are also written in C++. Following the authors’ recommendations, PTHash is tested with parameters (D-D,  $\alpha = 0.94$ ,  $c = 5.0$ ) and BBHash with parameter  $\gamma = 1.0$  (see the respective papers for an explanation of such parameters).

**Implementation Details.** We report the major implementation details for LPHash. The arrays  $L$  and  $P$  are compressed with Elias-Fano [16, 14]. Both the function  $f_m$  and the fall-back MPHF are implemented with PTHash using parameters (D-D,  $\alpha = 0.94$ ,  $c = 3.0$ ), unless otherwise specified. We do not compress the bit-vectors in the wavelet tree and we add constant-time support for rank queries using the Rank9 index [22, 46].

**Testing Machine.** The experiments were executed on a machine equipped with a Intel i9-9900K CPU (clocked at 3.60GHz), 64 GB of RAM, and running the Linux 5.13.0 operating system. The whole code (LPHash and competitors) was compiled with gcc 11.2.0, using the flags `-O3` and `-march=native`.

**Datasets.** We use datasets of increasing size in terms of number of distinct  $k$ -mers; namely, the whole-genomes of: *Saccharomyces Cerevisiae* (Yeast,  $11.6 \times 10^6$   $k$ -mers), *Caenorhabditis Elegans* (Elegans,  $95 \times 10^6$   $k$ -mers), *Gadus Morhua* (Cod,  $0.56 \times 10^9$   $k$ -mers), *Falco Tinnunculus* (Kestrel,  $1.16 \times 10^9$   $k$ -mers), and *Homo Sapiens* (Human,  $2.77 \times 10^9$   $k$ -mers). For each dataset, we obtain the corresponding SPSS by first building the compacted de Bruijn graph using BCALM2 [12], then running the UST algorithm [40]. At our code repository <https://github.com/jermp/lphash> we provide detailed instructions on how to prepare the datasets for indexing. Also, all processed datasets are available at <https://zenodo.org/record/7239205>.

**Space Effectiveness.** Figure 4 shows the space of LPHash in average bits/ $k$ -mer, by varying  $k$  from 31 to 63 with a step of 4, for both un-partitioned and partitioned data structures. We report the actual space usage achieved by the implementation against the theoretical space usage computed using Formula 6 (un-partitioned) and Formula 7 (partitioned). We also plot the theoretic minimum computed with Formula 5. For each combination of  $k$  and dataset, we choose a suitable value of minimizer length ( $m$ ) as reported in Table 1. For all datasets we use  $c = 3.0$  the PTHash  $f_m$  and fall-back, except on the

Table 1: Minimizer length  $m$  by varying  $k$  on the different datasets.

$k \rightarrow$	31	35	39	43	47	51	55	59	63
Yeast	15	15	16	16	16	16	18	18	18
Elegans	16	18	18	20	20	20	20	20	20
Cod	20	20	22	22	22	24	24	24	24
Kestrel	20	20	22	22	22	24	24	24	24
Human	21	21	23	23	26	26	28	28	28

Table 2: Query time in average nanoseconds per  $k$ -mer.

Method	$k$	Yeast		Elegans		Cod		Kestrel		Human	
		stream	random	stream	random	stream	random	stream	random	stream	random
LPHash	31	29	110	40	115	80	140	90	145	106	160
	35	32	238	42	225	70	248	75	251	96	265
	39	32	245	38	245	65	260	68	262	90	295
	43	30	264	36	264	61	272	65	272	80	303
	47	29	283	34	283	55	289	57	287	78	323
	51	28	300	33	300	51	307	52	302	71	334
	55	28	322	32	322	47	325	48	320	66	335
	59	28	340	31	340	45	340	45	338	64	340
63	27	360	30	360	42	360	43	360	60	370	
PTHash		35		60		100		120		120	
BBHash		50		140		200		220		220	

largest **Human** where we use  $c = 5.0$  to lower construction time at the expense of a larger space usage. As expected, the space lowers for increasing  $k$  and the partitioned data structure is always considerably smaller than the un-partitioned counterpart. The net result is that the achieved space is much better than that of the classic MPHFs traditionally used in the literature and in practice. To make a concrete example, partitioned LPHash for  $k = 63$  achieves 0.54, 0.65, 0.83, 0.58, and 0.87 bits/ $k$ -mer on **Yeast**, **Elegans**, **Cod**, **Kestrel**, and **Human** respectively. These values are  $5.1\times$ ,  $4.1\times$ ,  $3.2\times$ ,  $4.4\times$ , and  $3\times$  smaller than the those achieved by PTHash and reported in the caption of Figure 4 (and even smaller when compared to BBHash). We remark that, however, PTHash and BBHash are general-purpose MPHFs that can work with arbitrary keys, whereas the applicability of LPHash is restricted to  $k$ -mer sets.

Not surprisingly, the actual space usage of LPHash is higher than the theoretical one achieved by Formula 6 and Formula 7. This is due to the fact that, in practice, the MPHf  $f_m$  and the fall-back one take more than the theoretic minimum of  $\log_2 e = 1.44$  bits/key (e.g., 2.3 bits/ $k$ -mer in our implementation).

**Query Time.** Table 2 reports the query time for LPHash in comparison to PTHash and BBHash. Timings were collected using a single core of the processor; we query all  $k$ -mers read from the Human chromosome 13, for a total of  $\approx 100 \times 10^6$  queries. First of all, we noticed that query timings for un-partitioned and partitioned LPHash are the same, so we do not distinguish between the two data structures in Table 2. While the evaluation for the un-partitioned data structure (Algorithm 1) is simpler compared to that of the partitioned variant (Algorithm 2), it always performs two accesses per query to two compressed arrays. On the other hand, partitioning involves an extra rank query over a wavelet tree but it also spares many accesses to the arrays  $L$  and  $P$ .

We distinguish between streaming and random queries (lookups) for LPHash. Given a query string  $Q$ , we query for each  $k$ -mer read *consecutively* from  $Q$ , that is, for  $Q[1..k]$ ,  $Q[2..k + 1]$ ,  $Q[3..k + 2]$ , etc. We refer to this query modality as streaming; anything else different from streaming is a random lookup (i.e., “random” here means “without locality”). LPHash is optimized for streaming lookup queries, whereas PTHash and BBHash do not benefit from any specific query order. In fact, the locality-preserving nature of LPHash makes the calculation of hashes for consecutive  $k$ -mers very cheap, as consecutive  $k$ -mers are likely to be part of the same super- $k$ -mer.

Considering the result in Table 2, we see that LPHash’s streaming query time is in fact much smaller than random query time. Both timings are sensitive to the growth of  $k$ : while the streaming one slightly decreases for the better locality, the random one increases significantly for the more expensive hash calculations. Also note that the random query time is essentially independent from the size of the data structures but depends on  $w = k - m + 1$  as explained in Section 4. LPHash is as fast as PTHash for streaming queries on the smaller **Yeast** dataset, but actually up to  $2 - 2.5\times$  faster on the larger **Elegans**, **Cod**, **Kestrel**, and **Human**. Compared to BBHash, LPHash is  $2\times$  faster on **Yeast** and up to  $4 - 5\times$  faster on the larger datasets.

Random lookup time is, instead, slower for LPHash compared to PTHash and BBHash: this is expected because the evaluation of LPHash is more complex (it involves computing the minimizer, accessing several arrays, and computing a rank using a wavelet tree). However, we do not regard this as a serious limitation since, as we already motivated, the streaming query modality is the one used in Bioinformatics tasks involving  $k$ -mers [1, 7, 31, 42, 34]. We also see that: the slowdown is more evident on the smaller datasets while it tends to diminish on the larger ones; lookup time for  $k = 31$

is substantially better than that for the other values of  $k$  since 31 bases fits into a single 8-byte integer whereas any  $31 < k \leq 63$  requires the double of the size, making hash calculations more expensive.

**Building Time.** Lastly in this section, we consider building time. Refer to Table 3. Again, we report that the building time for un-partitioned and partitioned LPHash is the same. LPHash is competitive with the fastest BBHash and significantly faster than PTHash on the larger datasets. Specifically, it is faster than PTHash over the entire set of  $k$ -mers since it builds two smaller PTHash functions ( $f_m$  and fall-back). The slowdown seen for Cod is due to the larger fall-back MPHF, which is built with PTHash under a strict configuration ( $c = 3.0$ ) that privileges space effectiveness (and query efficiency) rather than building time. One could in principle use BBHash instead of PTHash for the fall-back function, hence trading space for better building time. For example, recall that we use  $c = 5.0$  on Human for this reason.

Table 3: Total building time, including the time to read the input and serialize the data structure on disk. All constructions were run with 4 processing threads and within 8GB of RAM.

Method	Yeast	Elegans	Cod	Kestrel	Human
	mm:ss	mm:ss	mm:ss	mm:ss	mm:ss
LPHash	00:03	00:35	11:30	08:50	16:41
PTHash	00:05	00:45	10:30	31:30	83:10
BBHash	00:03	00:25	02:40	05:05	13:45

Table 4: Minimal perfect hashing (MPH) problems for arbitrary and spectrum  $k$ -mer sets of size  $n$ .

Problem	Input set type	Bit-complexity	Reference
order-preserving MPH	arbitrary	$\Omega(n \cdot \log n)$	[17]
monotone MPH	arbitrary	$\Omega(n \cdot \log \log(2k))^5$	[3, Theorem 1]
“classic” MPH	arbitrary	$n \cdot \log_2 e$	[32, Lemma 1]
$r$ -wise MPH	arbitrary	$n \cdot \log_2 r / (2r) \cdot (1 + o(1))$	[29, Theorem 3]
$(1 - \varepsilon)$ LP-MPH	<b>spectrum</b>	$n \cdot (\varepsilon \log_2(e/\varepsilon) - (1 - \varepsilon) \log_2(1 - \varepsilon))$	[this paper, Theorem 1]

## 6 Conclusion and Open Questions

In this paper, we initiate the study of locality-preserving minimal perfect hash functions for  $k$ -mers. We illustrate a theoretic minimum lower bound of the space complexity of these functions and propose a construction that approaches the minimum when  $k$  increases. We show that a concrete implementation of the method is practical. Before this paper, one used to build a BBHash function over the  $k$ -mers and spend (approx.) 3 bits/ $k$ -mer and 100-200 nanoseconds per lookup. This work shows that it is possible to do significantly better than this when the  $k$ -mers come from a spectrum-preserving string set (SPSS): for example, less than 0.9 bits/ $k$ -mer and 30-60 nanoseconds per lookup. Our code is open-source.

As future work, we plan to carefully engineer the current implementation to accelerate construction and streaming queries even more. Other strategies for sampling the strings could be used other than random minimizers [19, 47]. For example, the *Miniception* by Zheng et al. [47] achieving  $\varepsilon = \frac{1.67}{w} + o(1/w)$ . Evaluating the impact of such different sampling schemes is a promising avenue for future research. Lastly, we also plan to investigate other strategies for handling the ambiguous minimizers. A better strategy is likely to lead to improved space effectiveness and faster construction.

We conclude with some questions that remain open from this work, in the hope of spurring further research on the topic. (i) *Given a SPSS with fragmentation factor  $\alpha$ , what is the lowest  $\varepsilon$  that can be achieved in theory? And in practice?* We know for Lemma 1 that  $\varepsilon \geq \alpha$ . *Can we improve this lower bound or prove that it is tight?* (ii) *Does exist a MPHF for a SPSS which is not locality-preserving and, yet, has a lower bit-complexity than a classic MPHF? Or do our results suggest that any MPHF built for a SPSS features the locality-preserving property for some  $\varepsilon \geq \alpha$ ?* (iii) In Table 4 we report the lower bounds for other minimal perfect hashing problems, such as  $r$ -wise, monotone, and order-preserving minimal perfect hashing. All of them are tight as they can be matched, at least in theory. We must be careful in comparing such lower bounds with that from Theorem 1 because those results are valid for *arbitrary*  $k$ -mer sets, not for  $k$ -mer spectra. Thus, a natural question is: *What is the bit-complexity of the other MPH problems in Table 4 when applied to a SPSS?* For example, we know that SSSHash [34, 33] is an order-preserving MPHF for a SPSS (and  $(1 - \alpha)$ -locality-preserving): in fact, its space usage is well below  $\Omega(\log n)$  bits/ $k$ -mer even with the ability of rejecting alien  $k$ -mers.

<sup>5</sup>Assadi et al. [3] recently showed that  $\Omega(n \cdot \log \log \log |U|)$  expected bits are necessary for a universe of size  $|U|$ , while  $O(n \cdot \log \log \log |U|)$  bits were already shown to be sufficient by Belazzougui et al. [5]. In our problem we have  $|U| = 4^k$ .

**Funding.** This work was partially supported by the project MobiDataLab (EU H2020 RIA, grant agreement N°101006879) and by the French ANR AGATE (ANR-21-CE45-0012).

## 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] Austin Appleby. Smhasher. <https://github.com/aappleby/smhasher>, Last accessed September 2022.
- [3] Sepehr Assadi, Martin Farach-Colton, and William Kuszmaul. Tight bounds for monotone minimal perfect hashing. *arXiv preprint arXiv:2207.10556*, 2022.
- [4] Djamel Belazzougui, Paolo Boldi, Giuseppe Ottaviano, Rossano Venturini, and Sebastiano Vigna. Cache-oblivious peeling of random hypergraphs. In *2014 Data Compression Conference*, pages 352–361. IEEE, 2014.
- [5] Djamel Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. Monotone minimal perfect hashing: searching a sorted table with  $O(1)$  accesses. In *Proceedings of the twentieth annual ACM-SIAM symposium on Discrete algorithms*, pages 785–794. SIAM, 2009.
- [6] Djamel Belazzougui, Fabiano C Botelho, and Martin Dietzfelbinger. Hash, displace, and compress. In *European Symposium on Algorithms*, pages 682–693. Springer, 2009.
- [7] Timo Bingmann, Phelim Bradley, Florian Gauger, and Zamin Iqbal. Cobs: a compact bit-sliced signature index. In *International Symposium on String Processing and Information Retrieval*, pages 285–303. Springer, 2019.
- [8] Fabiano C Botelho, Rasmus Pagh, and Nivio Ziviani. Practical perfect hashing in nearly optimal space. *Information Systems*, 38(1):108–131, 2013.
- [9] Karel Břinda, Michael Baym, and Gregory Kucherov. Simplitigs as an efficient and scalable representation of de Bruijn graphs. *Genome biology*, 22(1):1–24, 2021.
- [10] Chin-Chen Chang and Chih-Yang Lin. Perfect hashing schemes for mining association rules. *The Computer Journal*, 48(2):168–179, 2005.
- [11] 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.
- [12] 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.
- [13] Yann Collet. xxhash. <https://cyan4973.github.io/xxHash>, Last accessed September 2022.
- [14] Peter Elias. Efficient storage and retrieval by content and address of static files. *Journal of the ACM*, 21(2):246–260, 1974.
- [15] Emmanuel Esposito, Thomas Mueller Graf, and Sebastiano Vigna. Recsplit: Minimal perfect hashing via recursive splitting. In *2020 Proceedings of the Twenty-Second Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 175–185. SIAM, 2020.
- [16] Robert Mario Fano. On the number of bits required to implement an associative memory. *Memo-randum 61, Computer Structures Group, MIT*, 1971.
- [17] Edward A Fox, Qi Fan Chen, Amjad M Daoud, and Lenwood S Heath. Order-preserving minimal perfect hash functions and information retrieval. *ACM Transactions on Information Systems (TOIS)*, 9(3):281–308, 1991.
- [18] Edward A Fox, Qi Fan Chen, and Lenwood S Heath. A faster algorithm for constructing minimal perfect hash functions. In *Proceedings of the 15th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 266–273, 1992.

- [19] Martin C. Frith, Jim Shaw, and John L. Spouge. How to optimally sample a sequence for rapid analysis. *bioRxiv*, 2022.
- [20] Marco Genuzio, Giuseppe Ottaviano, and Sebastiano Vigna. Fast scalable construction of ([compressed] static— minimal perfect hash) functions. *Information and Computation*, page 104517, 2020.
- [21] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 12-14, 2003, Baltimore, Maryland, USA*, pages 841–850. ACM/SIAM, 2003.
- [22] Guy Jacobson. Space-efficient static trees and graphs. In *30th annual symposium on foundations of computer science*, pages 549–554. IEEE Computer Society, 1989.
- [23] Chirag Jain, Arang Rhie, Haowen Zhang, Claudia Chu, Brian Walenz, Sergey Koren, and Adam M. Phillippy. Weighted minimizer sampling improves long read mapping. *Bioinform.*, 36(Supplement-1):i111–i118, 2020.
- [24] Jamshed Khan, Marek Kokot, Sebastian Deorowicz, and Rob Patro. Scalable, ultra-fast, and low-memory construction of compacted de bruijn graphs with cuttlefish 2. *Genome biology*, 23(1):1–32, 2022.
- [25] Jamshed Khan and Rob Patro. Cuttlefish: fast, parallel and low-memory compaction of de Bruijn graphs from large-scale genome collections. *Bioinformatics*, 37(Supplement\_1):i177–i186, 2021.
- [26] Antoine Limasset, Guillaume Rizk, Rayan Chikhi, and Pierre Peterlongo. Fast and scalable minimal perfect hashing for massive key sets. In *16th International Symposium on Experimental Algorithms*, volume 11, pages 1–11, 2017.
- [27] Yi Lu, Balaji Prabhakar, and Flavio Bonomi. Perfect hashing for network applications. In *2006 IEEE International Symposium on Information Theory*, pages 2774–2778. IEEE, 2006.
- [28] Harry G Mairson. The program complexity of searching a table. In *24th Annual Symposium on Foundations of Computer Science (FOCS 1983)*, pages 40–47. IEEE, 1983.
- [29] Harry G Mairson. The effect of table expansion on the program complexity of perfect hash functions. *BIT Numerical Mathematics*, 32(3):430–440, 1992.
- [30] Bohdan S Majewski, Nicholas C Wormald, George Havas, and Zbigniew J Czech. A family of perfect hashing methods. *The Computer Journal*, 39(6):547–554, 1996.
- [31] Camille Marchet, Mael Kerbirou, and Antoine Limasset. Blight: efficient exact associative structure for k-mers. *Bioinformatics*, 37(18):2858–2865, 04 2021.
- [32] Kurt Mehlhorn. On the program size of perfect and universal hash functions. In *23rd Annual Symposium on Foundations of Computer Science*, pages 170–175. IEEE, 1982.
- [33] Giulio Ermanno Pibiri. On weighted k-mer dictionaries. In *International Workshop on Algorithms in Bioinformatics (WABI)*, pages 9:1–9:20, 2022.
- [34] Giulio Ermanno Pibiri. Sparse and skew hashing of k-mers. *Bioinformatics*, 38(Supplement\_1):i185–i194, 06 2022.
- [35] Giulio Ermanno Pibiri and Shunsuke Kanda. Rank/select queries over mutable bitmaps. *Information Systems*, 99(101756), 2021.
- [36] Giulio Ermanno Pibiri and Roberto Trani. Parallel and external-memory construction of minimal perfect hash functions with PTHash. *CoRR*, abs/2106.02350, 2021.
- [37] Giulio Ermanno Pibiri and Roberto Trani. PTHash: Revisiting FCH Minimal Perfect Hashing. In *The 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 1339–1348, 2021.
- [38] 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.



- [39] Giulio Ermanno Pibiri and Rossano Venturini. Handling massive  $N$ -gram datasets efficiently. *ACM Transactions on Information Systems*, 37(2):25:1–25:41, 2019.
- [40] 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.
- [41] 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.
- [42] Lucas Robidou and Pierre Peterlongo. findere: Fast and precise approximate membership query. In *String Processing and Information Retrieval*, pages 151–163, Cham, 2021. Springer International Publishing.
- [43] Saul Schleimer, Daniel S Wilkerson, and Alex Aiken. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 76–85, 2003.
- [44] Yoshihiro Shibuya, Djamel Belazzougui, and Gregory Kucherov. Space-efficient representation of genomic  $k$ -mer count tables. *Algorithms for Molecular Biology*, 17(1):1–15, 2022.
- [45] Grant P. Strimel, Ariya Rastrow, Gautam Tiwari, Adrien Piérard, and Jon Webb. Rescore in a Flash: Compact, Cache Efficient Hashing Data Structures for  $n$ -Gram Language Models. In *Proceedings of the 21st Annual Conference of the International Speech Communication Association*, pages 3386–3390, 2020.
- [46] Sebastiano Vigna. Broadword implementation of rank/select queries. In *International Workshop on Experimental and Efficient Algorithms*, pages 154–168. Springer, 2008.
- [47] Hongyu Zheng, Carl Kingsford, and Guillaume Marçais. Improved design and analysis of practical minimizers. *Bioinformatics*, 36(Supplement\_1):i119–i127, 2020.

## Supplementary Material

### Proofs of Corollary 1 and 2

**Corollary 1.** *Let  $\mathcal{S}$  be a SPSS with maximum fragmentation factor. Then any LP-MPHF for  $\mathcal{S}$  is 0-locality-preserving.*

*Proof.* By definition,  $\varepsilon = \mathbb{P}[\neg E_1 \text{ or } \neg E_2] = \mathbb{P}[\neg E_1] + \mathbb{P}[\neg E_2] - \mathbb{P}[\neg E_1 \text{ and } \neg E_2]$  because  $\neg E_1$  and  $\neg E_2$  are not mutually exclusive (there could obviously be  $k$ -mers  $x_i$  and  $x_{i+1}$  that do not belong to the same string and for which  $f(x_{i+1}) \neq f(x_i) + 1$ ). But  $\mathbb{P}[\neg E_1 \text{ and } \neg E_2] = \mathbb{P}[\neg E_1]$  because  $\neg E_1 \subseteq \neg E_2$  since  $\mathcal{S}$  has maximum fragmentation factor and there are no  $k$ -mers that belong to the same string ( $m = n$ ). In conclusion:  $\varepsilon = \mathbb{P}[\neg E_1] + \mathbb{P}[\neg E_2] - \mathbb{P}[\neg E_1 \text{ and } \neg E_2] = \mathbb{P}[\neg E_1] + \mathbb{P}[\neg E_2] - \mathbb{P}[\neg E_1] = \mathbb{P}[\neg E_2] = 1$ .  $\square$

**Corollary 2.** *Let  $\mathcal{S}$  be a SPSS with fragmentation factor  $\alpha$ . Then any “classic” MPHf for  $\mathcal{S}$  is  $\epsilon(1-\alpha)$ -locality-preserving for a very small  $\epsilon \geq 0$ .*

*Proof.* A “classic” MPHf will hash  $x_i$  and  $x_{i+1}$  independently, thus  $\mathbb{P}[\neg E_1] = 1 - \epsilon$  for a very small  $\epsilon \geq 0$ . In this case the two events  $\neg E_1$  and  $\neg E_2$  are independent, thus:  $\varepsilon = \mathbb{P}[\neg E_1] + \mathbb{P}[\neg E_2] - \mathbb{P}[\neg E_1] \cdot \mathbb{P}[\neg E_2] = 1 - \epsilon + \alpha - \alpha(1 - \epsilon) = 1 - \epsilon(1 - \alpha)$ .  $\square$

### Proof of Theorem 2

**Property 2.**  $|g| - k + 1 \leq p_{g,1} \leq w$  for any super- $k$ -mer  $g$ .

**Theorem 2.** *Given a random minimizer scheme  $(k, m, h)$  with  $m > (3 + \epsilon) \log_4(w + 1)$  for any  $\epsilon > 0$  and  $w = k - m + 1$ , there exists a  $(1 - \epsilon)$ -LP MPHf for a SPSS  $\mathcal{S}$  which takes*

$$n \cdot \frac{2}{w+1} \cdot \left( \log_2 \left( 4e \cdot (w+1)^2 \right) + o(1) \right) \text{ bits} \quad (6)$$

with  $\varepsilon = \frac{2}{w+1}$  and  $n = |\text{spectrum}_k(\mathcal{S})|$ .



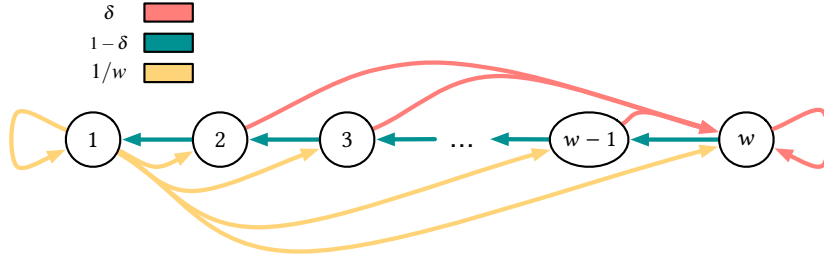


Figure 5: The chain is in state  $1 \leq p \leq w$  if the minimizer starts at position  $p$  in the  $k$ -mer. Different edge colors represent different probabilities.

*Proof.* We have to represent the three components  $f_m$ ,  $L$ , and  $P$ . The MPHf  $f_m$  can be realized using  $2n/(w+1) \cdot \log_2 e$  bits. The array  $L$  stores  $2n/(w+1)$  integers whose sum is  $n$  since each element represents the number of  $k$ -mers in a super- $k$ -mer. Therefore  $L$  can be represented with Elias-Fano [16, 14] using at most  $\frac{2n}{w+1} \cdot (\log_2(\frac{w+1}{2}) + 2 + o(1))$  bits. We can also represent  $P$  using this space bound. In fact, note that since  $|g| - k + 1 \leq p_{g,1} \leq w$  for Property 2, we can store the quantities  $(p_{g,1} - (|g| - k + 1))$  in  $P$  for any super- $k$ -mer  $g$ . If we compute the prefix-sums for the array  $P$ , we can encode it with Elias-Fano. The last element in the prefix-summed  $P$  is at most  $\sum_g (w - (|g| - k + 1)) = w \cdot \frac{2n}{w+1} - n < 2n - n = n$ . Summing these spaces, the claimed space bound follows.  $\square$

### Proof of Theorem 3

**Theorem 3.** *Given a random minimizer scheme  $(k, m, h)$  with  $m > (3 + \epsilon) \log_4(w + 1)$  for any  $\epsilon > 0$  and  $w = k - m + 1$ , there exists a  $(1 - \epsilon)$ -LP MPHf for a SPSS  $\mathcal{S}$  which takes*

$$n \cdot \frac{2}{w+1} \cdot \left( \log_2 \left( e \frac{16 \cdot 2^{1/4}}{3} \cdot (w+1) \right) + o(1) \right) \text{ bits} \quad (7)$$

with  $\epsilon = \frac{2}{w+1}$  and  $n = |\text{spectrum}_k(\mathcal{S})|$ .

Before proving Theorem 3, we need to compute the proportions of the different types of super- $k$ -mers as given by the FL rule. For ease of notation, let  $P_{lr} = \mathbb{P}[g \text{ is left-right-max}]$ ,  $P_l = \mathbb{P}[g \text{ is left-max}]$ ,  $P_r = \mathbb{P}[g \text{ is right-max}]$ ,  $P_n = \mathbb{P}[g \text{ is non-max}]$ , for any super- $k$ -mer  $g$ .

**Remark 1.** *The FL rule is a partitioning rule, i.e.,  $P_{lr} + P_l + P_r + P_n = 1$  for any super- $k$ -mer.*

We now want to derive the expression for the probabilities  $P_{lr}$ ,  $P_l$ ,  $P_r$ , and  $P_n$ , parametric in  $k$  and  $m$ . Let  $X : \Sigma^k \rightarrow \{1, \dots, w\}$  be a discrete random variable, modelling the starting position of the minimizer in a  $k$ -mer. The corresponding Markov chain is illustrated in Figure 5. Each state of the chain is labelled with the corresponding value assumed by  $X$ , i.e., with each value in  $\{1, \dots, w\}$ . Clearly, we have a left-right-max super- $k$ -mer if, from state  $w$  we transition to state  $w - 1$ , then to  $w - 2, \dots$ , down to state 1. Each state has a “fall-back” probability to go to state  $w$  which corresponds to the event that the right-most  $m$ -mer (that coming next to the right) is the new minimizer. If the chain reaches state 1, instead, we know that we are always going to see a new minimizer next. If  $c \in [1, \Delta]$  is the code assigned to the current minimizer by the coding function  $h$  used by  $\mu$ , for some  $\Delta$  (e.g,  $c$  is a 64-bit hash code, so  $\Delta = 2^{64}$ ), the probability for any  $m$ -mer to become the new minimizer is equal to  $\delta = \frac{c-1}{\Delta}$ . Vice versa, the probability of keeping the same minimizer when sliding one position to the right, is  $1 - \delta$ . Whenever we change minimizer, we generate a new code  $c$  and, hence, the probability  $\delta$  changes with every formed super- $k$ -mer. Nonetheless, the following Theorem shows that the probabilities  $P_{lr}$ ,  $P_l$ ,  $P_r$ , and  $P_n$ , do not depend on  $\delta$ .

**Theorem 4.** *For any random minimizer scheme  $(k, m, h)$  we have*

$$\begin{aligned} P_{lr} &= \mathbb{P}[g \text{ is left-right-max}] = W^2 + 1/w \\ P_l &= \mathbb{P}[g \text{ is left-max}] = W(1 - W) \\ P_r &= \mathbb{P}[g \text{ is right-max}] = W(1 - W) \\ P_n &= \mathbb{P}[g \text{ is non-max}] = W^2 \end{aligned}$$

where  $W = \frac{1}{2} \cdot (1 - \frac{1}{w})$  and  $w = k - m + 1$ .

We give the following Lemma to prove Theorem 4. (When we write “first”/“last”  $k$ -mer we are going to silently assume “of a super- $k$ -mer”.)

**Lemma 3.**  $\mathbb{P}[X = 1] = \frac{1}{2}$  and  $\mathbb{P}[X = w] = \frac{1}{2} \cdot (1 + \frac{1}{w})$ .

*Proof.* First note that

$$\mathbb{P}[X = p \text{ in the first } k\text{-mer}] = \mathbb{P}[X = 1] \cdot \frac{1}{w}, \text{ for any } 1 \leq p \leq w - 1. \quad (8)$$

Then we have the following equivalences.

$$\begin{aligned} \sum_{p=1}^w \mathbb{P}[X = p \text{ in the first } k\text{-mer}] &= 1 \iff \\ \mathbb{P}[X = w] + \sum_{p=1}^{w-1} \mathbb{P}[X = p \text{ in the first } k\text{-mer}] &= 1 \iff \\ \mathbb{P}[X = w] + \mathbb{P}[X = 1] \cdot \left(1 - \frac{1}{w}\right) &= 1, \text{ using Equation 8.} \end{aligned} \quad (9)$$

Now note that

$$\mathbb{P}[X = w] = P_{lr} + P_r \quad (10)$$

because the starting position of the minimizer of the **first**  $k$ -mer of any left-right-max and of any right-max super- $k$ -mer is  $w$ . In a similar way, we have that

$$\begin{aligned} \mathbb{P}[X = 1 \text{ in the last } k\text{-mer}] &= \\ \mathbb{P}[X = 1] + \mathbb{P}[X = 1 \text{ in the first } k\text{-mer}] &= \\ \mathbb{P}[X = 1] \cdot \left(1 + \frac{1}{w}\right) &= P_{lr} + P_l \end{aligned} \quad (11)$$

because the starting position of the minimizer of the **last**  $k$ -mer of any left-right-max and of any left-max super- $k$ -mer is 1. Now we prove that

$$P_l = P_r. \quad (12)$$

In fact, we have

$$\begin{aligned} P_l &= \mathbb{P}[X = w \text{ in first } k\text{-mer}] \cdot \mathbb{P}[X \neq 1 \text{ in last } k\text{-mer}] = \\ &= (1 - \mathbb{P}[X \neq w \text{ in first } k\text{-mer}]) \cdot (1 - \mathbb{P}[X = 1 \text{ in last } k\text{-mer}]) = \\ &= \left(1 - \mathbb{P}[X = 1] \cdot \left(1 - \frac{1}{w}\right)\right) \cdot \left(1 - \mathbb{P}[X = 1] \cdot \left(1 + \frac{1}{w}\right)\right) = \\ &= (\mathbb{P}[X = 1])^2 \cdot \left(1 - \frac{1}{w^2}\right), \text{ and similarly} \\ P_r &= \mathbb{P}[X \neq w \text{ in first } k\text{-mer}] \cdot \mathbb{P}[X = 1 \text{ in last } k\text{-mer}] = \\ &= \mathbb{P}[X = 1] \cdot \left(1 - \frac{1}{w}\right) \cdot \mathbb{P}[X = 1] \cdot \left(1 + \frac{1}{w}\right) = \\ &= (\mathbb{P}[X = 1])^2 \cdot \left(1 - \frac{1}{w^2}\right). \end{aligned}$$

So from Equation 12 we have  $P_{lr} + P_r = P_{lr} + P_l$  which, using Equation 11 and 10, yields

$$\mathbb{P}[X = w] = \mathbb{P}[X = 1] \cdot \left(1 + \frac{1}{w}\right). \quad (13)$$

Lastly the Lemma follows by using Equation 13 into Equation 9.  $\square$

Now we prove Theorem 4.

*Proof.* Since the FL rule induces a partition:

$$\begin{aligned} P_{lr} + P_r + P_l + P_n &= 1 \iff \\ P_{lr} + P_{lr} + P_r + P_l + P_n &= 1 + P_{lr} \text{ (adding } P_{lr} \text{ to both sides)} \iff \\ 2\mathbb{P}[X = w] + P_n &= 1 + P_{lr} \text{ (knowing that } P_{lr} + P_r = P_{lr} + P_l = \mathbb{P}[X = w]) \iff \\ P_{lr} &= P_n + \frac{1}{w} \text{ (for Lemma 3).} \end{aligned} \quad (14)$$

Again exploiting the fact that  $P_{lr} + P_r = P_{lr} + P_l = \mathbb{P}[X = w]$ , we also have

$$P_l = P_r = \mathbb{P}[X = w] - P_{lr} = \frac{1}{2} \cdot \left(1 + \frac{1}{w}\right) - P_n - \frac{1}{w}. \quad (15)$$

We have therefore to compute  $P_n$  to also determine  $P_{lr}$ ,  $P_l$ , and  $P_r$ .

$$\begin{aligned} P_n &= \mathbb{P}[X \neq w \text{ in first } k\text{-mer}] \cdot \mathbb{P}[X \neq 1 \text{ in last } k\text{-mer}] = \\ &= \mathbb{P}[X = 1] \cdot \left(1 - \frac{1}{w}\right) \cdot (1 - \mathbb{P}[X = 1 \text{ in last } k\text{-mer}]) = \\ &= \mathbb{P}[X = 1] \cdot \left(1 - \frac{1}{w}\right) \cdot \left(1 - \mathbb{P}[X = 1] \cdot \left(1 + \frac{1}{w}\right)\right) = \left(\frac{1}{2} \cdot \left(1 - \frac{1}{w}\right)\right)^2 \text{ for Lemma 3.} \end{aligned} \quad (16)$$

Now letting  $W = \frac{1}{2} \cdot \left(1 - \frac{1}{w}\right)$  and substituting  $P_n = W^2$  (Equation 16) into Equation 14 and 15, the Theorem follows.  $\square$

Lastly in this section, we prove Theorem 3.

*Proof.* We have to represent the following components: the minimizer MPHF  $f_m$ , the array  $R$  storing the types of the super- $k$ -mers, and the arrays  $L_l$ ,  $L_r$ ,  $L_n$ , and  $P_n$ . The MPHF  $f_m$  takes (i)  $2n/(w+1) \cdot \log_2 e$  bits and  $R$  takes (ii)  $2n/(w+1) \cdot (2 + o(1))$  bits as already discussed. For sufficiently large  $w$ , the proportions of super- $k$ -mers are all  $\approx 1/4$  for Theorem 4 so that we have  $\frac{n}{2(w+1)}$  super- $k$ -mers of each type. (This also means that the choice of 2-bit codes for the symbols of  $R$  is essentially optimal.) For the left-right-max super- $k$ -mers we do not store anything and they cover  $w \cdot \frac{n}{2(w+1)} \approx n/2$  of the  $k$ -mers. The other half of the  $k$ -mers is handled by the other three super- $k$ -mer types: we do not know the exact amount of  $k$ -mers per type; yet, we are sure that the worst case for the space happens when each partition takes the same amount of  $k$ -mers (uniform partitioning), i.e.,  $n/6$ . The space for  $L_l$  plus that for  $L_r$  is then (iii)  $2 \cdot \frac{n}{2(w+1)} \cdot (\log_2(n/6 \cdot (w+1)/(2n)) + 2 + o(1)) = \frac{n}{w+1} \cdot (\log_2((w+1)/3) + 2 + o(1))$  bits. Similarly, the array  $L_n$  takes  $\frac{n}{2(w+1)} \cdot (\log_2((w+1)/3) + 2 + o(1))$  bits. The space for the array  $P_n$  is instead  $\frac{n}{2(w+1)} \cdot (\log_2(2(w+1)/3) + 2 + o(1))$  bits following a similar argument to that used in the proof of Theorem 2: the last element in the prefix-summed  $P_n$  is at most  $\sum_{g \text{ is non-max}} (w - (|g| - k + 1)) = w \cdot \frac{n}{2(w+1)} - n/6 < n/2 - n/6 = n/3$ . Therefore, the space for  $L_n$  plus that for  $P_n$  is (iv)  $\frac{n}{w+1} \cdot (\log_2((w+1)/3) + 5/2 + o(1))$  bits. Summing spaces (i), (ii), (iii), (iv) together, the claimed space bound follows.  $\square$