1. Introduction

Perfect hashing is an elementary problem in computer science. The goal is to find a collision free hash function for a given static key set. Perfect hash functions are used for memory efficient storage and fast retrieval of items from static sets, such as words in natural languages, reserved words in programming languages or interactive systems, item sets in data mining techniques \cite{13,14}, routing tables \cite{43}, sparse spatial data \cite{35}, and large web maps \cite{18}. Perfect hashing methods can be used to construct a data structure to compactly store a static key set that supports queries to locate keys in one probe. For applications with only successful searches, a key is simply represented by the value of a perfect hash function and the key set is not needed to locate information related with the key. For applications with unsuccessful searches, the key set has to be represented somehow to handle collisions.

There are many applications where the search space is restricted to keys with successful searches. One good example can be found in the deduplication of objects in a file system, which maintains an index that maps each unique object to a disk location of a block that holds it. At a given point in time, the file system knows all object
Throughout this paper we denote \( \log_2 \) size \( u \). A proof was later given in [44]. Consider Mehlhorn’s Theorem III.2.3.6 (a) presented in [37] as a starting point to derive theoretical lower bounds for the space consumption of the PHFs and MPHF’s description.

**Theorem 1.1** (Mehlhorn [37] Theorem III.2.3.6 (a)). Let \( u, m, n \) be non-negative integers. Given a key universe \( U \) of size \( u \), a class \( H \) of functions \( h : U \rightarrow \{0, \ldots, m-1\} \) is called \((u,m,n)\)-perfect if for every \( S \subseteq U \), \( |S| = n \), there is \( h \in H \) such that \( h \) is perfect for \( S \). Then

\[
|H| \geq \frac{u}{n!} \binom{m}{n}^{-1}.
\]

Our focus in this paper is the case where \( m < 3n \). For this constraint, applying Stirling’s approximation \( x! = x^e e^{-x} \sqrt{2\pi x} \) to \( |H| \) yields an information theoretical lower bound for a PHF \((m = 1.23n)\) of \( (m-n+\frac{1}{2}) \log (1-n/m) - (u-n+\frac{1}{2}) \log (1-u/u) \) and for an MPHF \((m = n)\) of \( (n-u-\frac{1}{2}) \log (1-n/u) - \frac{1}{2} \log (2\pi n) \). Con-considering \( u > n \), this gives a value of approximately 0.89n bits for PHFs and approximately 1.44n bits for MPHF.

### 1.2. Contributions

In our algorithms we use the well-known idea of partitioning the input key set into small buckets. When the key set fits entirely in the internal memory there is no need for partitioning and we treat it as a single bucket. This leads to an algorithm that operates on internal random access memory, which is referred to as RAM algorithm from now on. When the key set does not fit in the internal memory we have to do the partitioning and optimize our algorithm for I/O operations. This leads to an external memory algorithm, which is referred to as EM algorithm from now on.

The RAM and EM algorithms combine practical performance, expected linear construction time and nearly optimal space consumption for the resulting data structure. The engineering to combine several theoretical results into a practical solution has turned perfect hashing into a very compact data structure to solve the membership problem when the key universe is static and known in advance. Perfect hashing is the data structure that provides the best trade-off between space usage and lookup time when compared with other open addressing and chaining hash schemes too index static key sets [7].

The space consumption of our algorithms to store the resulting functions depends on the relation between \( m \) and \( n \). For \( m = 1.23n \), the space consumption is approximately \( 1.95n + o(n) \) bits for the RAM algorithm and \( 2.7n + o(n) \) bits for the EM algorithm. For \( m = n \), the space consumption is approximately \( 2.62n + o(n) \) bits for the RAM algorithm and \( 3.3n + o(n) \) bits for the EM algorithm. We remark that although the EM algorithm generates functions whose space consumption is \( O(n) \) bits, the hidden constant in the asymptotic notation requires that \( n \) be in the order of hundreds of millions to achieve the space consumption described above. In practice this is not a limitation because for smaller sets the RAM algorithm should be used rather than the EM algorithm which is designed for large sets that cannot be processed in internal memory.
The RAM algorithm works on acyclic random graphs given by function values of uniform hash functions on the keys of an input set \( S \) (see Section 2 for the definition of uniform hashing). The idea of basing perfect hashing on acyclic random graphs was used by Majewski, Wormald, Havas and Czech [36] to solve a different problem, i.e., to construct order-preserving (minimal) perfect hash functions—given any two arbitrary keys \( x,y \in S \) such that \( x < y \), a perfect hash function \( h \) is order-preserving if \( h(x) < h(y) \). Due to the order-preserving property the resulting functions require \( \Omega(\log n) \) bits per key of space. Most of the cases where a hash function is used the order-preserving property is not required. The algorithms presented in this paper construct functions that achieve a space usage of \( O(1) \) bits per key rather than \( O(\log n) \) bits per key.

The EM algorithm uses a number of techniques from the literature to allow the construction of PHFs or MPHFs for sets on the order of billions of keys. There are many different hash functions from the literature that may be used in the constructions. The important insight here is that we split the problem in small buckets using the split-and-share technique [21,22]. This has both practical and theoretical implications. From the theoretical point of view we show that by using a technique to simulate fully random hash functions on the small buckets we are able to prove that the EM algorithm works for every key set with high probability. From the practical point of view, we create buckets that are small enough to fit in the CPU cache, resulting in a significant speedup (in processing time per element) compared to other methods.

We demonstrate the scalability of the EM algorithm by reporting how it is set to work in an implementation that can construct an MPHF for over a billion keys in 46 min on a commodity PC with a 1.86 gigahertz Intel Core 2 processor with 1 gigabytes of main memory and a L2 cache of 4 megabytes, running Linux operating system version 2.6. The popularity of the C Minimal Perfect Hashing Library (http://cmph.sf.net), which is an open source implementation of the algorithms described herein, indicates how useful the results are in practice. The library has been downloaded more than 8600 times by May 2012, and is part of Ubuntu and Debian—two popular Linux distributions.

Preliminary partial results of this paper appeared in [8,10]. In [8] we describe the RAM algorithm, but both the description and the analysis of the algorithm are sketchy and incomplete. In [10] we describe the EM algorithm. We present in this paper significant improvements and extensions on those results. For the RAM algorithm, we now provide a full description with enough details to easily derive an efficient implementation and present a more detailed analysis of time and space complexities of the important phases of the algorithm. For the EM algorithm, we have (i) redesigned the algorithm to make it 40% faster and to construct functions that are 15% more compact. Both are a direct consequence of using a random acyclic hypergraph with edges connecting three vertices instead of a random acyclic graph with edges connecting two vertices; (ii) showed how to engineer a family of hash functions that efficiently simulates uniform hash functions in terms of space usage on small buckets of keys; and (iii) done a new set of experiments to show the efficiency of this new version of the EM algorithm.

1.3. Road map

In Section 2 we discuss the related work. In Section 3 we present the RAM algorithm. In Section 4 we describe the hash functions used in the EM algorithm. In Section 5 we present the EM algorithm. In Section 6 we present a heuristic version of the EM algorithm (HEM algorithm). In Section 7 we show the experimental results for the RAM and EM algorithms. Finally, in Section 8 we present the final remarks and conclusions.

2. Related work

In this section we review some of the most important theoretical, practical and heuristic results on perfect hashing. Czech et al. [17] provide a more comprehensive survey until 1997. There is a gap between theory and practice among theoretical and practical minimal perfect hashing methods. The aim of this section is to discuss the existent gap among the types of algorithms available in the literature. For this we need the concepts of uniform and universal hash functions.

2.1. Uniform versus universal hash functions

The construction of minimal perfect hash functions usually uses functions chosen uniformly from a fixed family \( \mathcal{H} \) of hash functions. To analyze our results we use two popular families that have also been used to analyze many other hashing schemes in the past: (i) uniform hash functions and (ii) universal classes of hash functions. A uniform hash function is a function that is uniformly chosen at random from the set of all \( m^m \) possible hash functions mapping from \( U \) to \( \{0,\ldots,m-1\} \) and distributes all keys from the universe \( U \) independently and uniformly over \( \{0,\ldots,m-1\} \). Various adaptive hashing schemes presume that a hash function with certain prescribed properties can be found in constant expected time. This holds if the function is chosen uniformly at random from all possible functions until a suitable one is found, but not necessarily if the search is limited to a smaller set of functions. The amount of space to represent a uniform hash family is at least \( u \log m \) bits, which usually exceeds the available storage in practice. This situation led Carter and Wegman [12] to the concept of universal classes of hash functions.

**Definition 2.1.** A universal class of hash functions is a set \( \mathcal{H} \) of hash functions with the property that if \( h \colon U \to \{0,\ldots,m-1\} \) is chosen at random from \( \mathcal{H} \) then for all \( x,y \in U \) with \( x \neq y \) we have \( \Pr(h(x) = h(y)) \leq 1/m \).

**Definition 2.2.** A strongly universal class of hash functions or pairwise independent class of hash functions is a set \( \mathcal{H} \) of hash functions with the following property. For all \( x_1, x_2 \in U \) such that \( x_1 \neq x_2 \) and \( a_1, a_2 \in \{0,\ldots,m-1\} \), if one
randomly chooses a function \( h: U \rightarrow \{0, \ldots, m-1\} \) from \( \mathcal{H} \), then the following holds: \( \Pr[h(x_1) = a_1 \text{ and } h(x_2) = a_2] = 1/m^2 \).

Fortunately, for many applications weaker randomness properties (such as pairwise independence) suffice [2]. The split-and-share approach presented in [21,22] allows the construction of hash functions that behave as uniform hash functions on fewer than \( n \) keys (e.g., those in one bin) with high probability. In this paper we make use of the split-and-share approach to simulate a class of uniform hash functions on small buckets so we can show that the EM algorithm can be set to work with high probability on any key set.

### 2.2. Theoretical results

In this section we review some of the most important theoretical results on minimal perfect hashing, which do not assume that uniform hash functions are available without any extra cost of space. Our algorithms, as well as all other algorithms mentioned in this paper, adopt the Word RAM model of computation [28], in which an element of the universe \( U \) fits into one machine word and arithmetic operations and memory accesses have unit costs.

Fredman and Komlós [27] prove that at least \( n \log e + \log \log u - O(\log n) \) bits are required to represent an MPHF, provided that \( u \geq n^x \) for some \( x > 2 \). Mehlhorn [37] shows that the Fredman and Komlós bound is almost tight by providing an algorithm that constructs an MPHF that can be represented with at most \( \log e + \log \log u + O(\log n) \) bits. However, Mehlhorn’s algorithm is far from practical because its construction and evaluation time are exponential in \( n \). Schmidt and Siegel [46] propose the first algorithm to construct an MPHF with constant evaluation time and description size \( O(n + \log \log u) \) bits. From a practical point of view, Schmidt and Siegel’s algorithm is not attractive. The scheme is complicated to implement and the constant of the space bound is large: for a set of \( n \) keys, it needs at least \( 29n \) bits to store the hash function, which means a space usage similar in practice to the best schemes using \( O(n \log n) \) bits. Albeit it seems that Schmidt and Siegel aim to describe their algorithmic ideas in the clearest possible way, not trying to optimize the constant, it seems hard to improve the space usage significantly.

More recently, Hagerup and Tholey [29] came up with the best theoretical result we know of. Their MPHF can be evaluated in \( O(1) \) time and stored in \( n \log e + \log \log u + O(n(\log \log n)^2/\log n + \log \log \log u) \) bits. The construction time is \( O(n + \log \log u) \) using \( O(n) \) words of space. In spite of its theoretical importance, Hagerup and Tholey’s algorithm is not practical either, as it only works when \( n \) is large. For \( n < 2^{150} \) the scheme is not well-defined, as it relies on splitting the key set into buckets of size \( n \leq \log n/2 \)(\( 2\log \log n \)). If we fix this by letting the bucket size be at least 1, then buckets of size one will be used for \( n < 2^{200} \), which means that the space usage will be at least \( 3 \log n + \log 7 \) bits. For a set of a billion keys, this is more than 17 bits per element. Thus, the Hagerup–Tholey MPHF is not space efficient in practical situations. While we believe that their algorithm has been optimized for simplicity of exposition, rather than constant factors, it seems difficult to significantly reduce the space usage based on their approach.

### 2.3. Practical results

We now describe some of the main practical results upon which our work is based. They are characterized by simplicity and provably low constant factors.

The first two results assume uniform random hash functions to be available without any extra cost of space (i.e., the results assume uniform hashing; in practice, universal classes of hash functions are used instead, as a heuristic). Majewski et al. [36] propose a family of algorithms to construct MPHFs based on \( r \)-uniform hypergraphs (i.e., with edges of size \( r \)). The resulting functions can be evaluated in \( O(1) \) time and stored in \( O(n \log n) \) bits. Belazzougui et al. [4] improve the constant involved in the space consumption of the algorithm presented by Czech et al. [16], but the space consumption is still \( O(n \log n) \) bits. In both cases, the MPHF can be constructed in expected \( O(n) \) time. The authors of [6] gave experimental evidence that their construction procedure works well in practice.

The principle of our RAM algorithm was described by Chazelle et al. [15] in a work on “Bloomier Filters”, somewhat hidden, without reference to perfect hashing and with no connection to acyclic hypergraphs. The main difference to the RAM algorithm is that we do recognize the connection to acyclic random hypergraphs, which allows us to provide a neat solution and a tight analysis to optimize the constant of the space usage considering implementation aspects, as well as a way of constructing MPHFs from those PHFs.

Pagh [41] proposes an algorithm to construct MPHFs of the form \( h(x) = (f(x) + d\log(x)) \mod n \), where \( f \) and \( g \) are randomly chosen from a family of universal hash functions, and \( d \) is a vector of “displacement values” used to resolve collisions caused by the function \( f \). The evaluation of the function is optimal in terms of random memory accesses once it does only one, but the space usage is \( (2+\epsilon)n \log n \) bits for a real value \( \epsilon > 0 \). Dietzfelbinger and Hagerup [20] improve Pagh’s result by reducing the space usage to \( (1+\epsilon)n \log n \) bits. Woelfel [48] shows how to decrease the space usage to \( O(n \log \log n) \) bits asymptotically, still with a quite simple algorithm. However, there is no empirical evidence on the practicality of this scheme. Belazzougui et al. [4] show how to decrease the space usage further, to \( O(n) \) bits. For certain parameter settings the algorithm constructs PHFs and MPHFs slightly more compact than the ones presented in this paper. However, the algorithm trades space for evaluation time. The fastest function constructed by the algorithm in [4] carries out at least four random memory accesses whereas the RAM algorithm constructs functions that require up to three. Therefore, due to the increasing gap between CPU speed and memory speed, the functions constructed with the RAM algorithm are at least 25% faster. This can be clearly seen in [4, Section 4.1] where a thorough comparison between the algorithms is given.
2.4. Heuristics

In this section we consider algorithms designed for specific applications where, in general, just experimental evidences of the behavior of the algorithms are provided.

Fox et al. [25,26] present several algorithms for constructing MPHF that in experiments require between 2 and 8 bits per key to be stored. However, it is theoretically shown in [17, Section 6.7] that the construction algorithms have exponential running times in expectation. Also, lookup times are constant but there is no guarantee that the number of bits per key to store the function is constant as n increases.

The work by Lefebvre and Hoppe [35] has the same issue of not providing any guarantee that the storage space of the resulting functions will be a constant number of bits per key. The authors designed a method to specifically represent sparse spatial data, and the resulting PHFs require more than 3 bits per key to be stored. In the same trend, Chang and Lin [13] and Chang et al. [14] design MPHFs tailored for mining association rules and traversal patterns in data mining techniques.

3. RAM algorithm

The RAM algorithm is a randomized algorithm of Las Vegas type. The RAM algorithm works on a random acyclic hypergraph given by uniform hash function values on an input key set \( S \subset U \), \( |S| = n \). A hypergraph is the generalization of a standard undirected graph where each edge connects \( r \geq 2 \) vertices. A hypergraph is acyclic if and only if some sequence of repeated deletions of edges containing at least 1 vertex of degree 1 yields a hypergraph without edges [17, p. 103].

We now give the intuition behind the RAM algorithm. The algorithm has three steps:

1. We start with \( V = \{0, \ldots, m-1\} \), regarded as vertices of an \( r \)-partite hypergraph where \( m = \lceil c(r)m \rceil \) for certain numbers \( c(2), c(3), c(4) \), ... defined later on, and \( r \) hash functions \( h_0, \ldots, h_{r-1} \) defined as follows. Given an integer \( \eta = \lceil m/r \rceil \) and functions \( h_i: U \mapsto \{0, \ldots, \eta-1\}, \) \( 0 \leq i < r \), that are fully random or uniform on \( S \subset U \) and can be evaluated in O(1) time, we denote each \( h_i: U \mapsto (i \times \eta, \ldots, (i+1) \times \eta-1) \) as

\[
h_i(x) = h_i'(x) + i \times \eta.
\]  

The \( r \) hash functions map \( S \) into \( r \) disjoint partitions of the vertex set. This way each key \( x \in S \) gives rise to an edge \( e(x) = (h_0(x), \ldots, h_{r-1}(x)) \). Vertex and edge sets form the random hypergraph \( G_r \). This part of the algorithm is called Mapping step.

2. If \( G_r \) is acyclic, one can proceed as follows:

- Use a linear equation to calculate an index \( i(x) \in \{0, \ldots, r-1\} \) from \( x \) defined as follows:

\[
i(x) = \left( \sum_{0 \leq j < r} g(h_j(x)) \right) \mod r,
\]

where a function \( g: V \mapsto \{0, \ldots, r-1\} \) has to be found to satisfy the conditions in the next steps. Function \( g \) is implemented as an array containing the values \( g(v), \ v \in V \) and hence we denote \( g(v) = g[v] \) from now on.

- To each key \( x \in S \), assign an element \( h_{i(x)}(x) \) of \( e(x) \) such that the assignment \( x \mapsto h_{i(x)}(x) \) is one-to-one on \( S \). For this reason we call this step Assigning step.

- Associate each key \( x \in S \) with a position \( h(x) \) in the hash table by an one-to-one mapping \( h: S \mapsto \{0, \ldots, m-1\} \) (so \( h \) is a perfect hash function), with the additional property that \( h(x) \in e(x) \). This is possible as a direct consequence of the definition of acyclicity. To calculate \( h(x) \) from \( x \) one has to find the index \( i(x) \in \{0, \ldots, r-1\} \) with \( h(x) = h_{i(x)}(x) \).

3. We compress the range of function \( h \) from \( \{0, \ldots, m-1\} \) to \( \{0, \ldots, n-1\} \) to obtain a minimal perfect hash function. The compression technique, referred to as ranking, uses a well-studied primitive in succinct data structures that can be implemented in O(1) time [40,42,45]. At the beginning of the assigning step we consider \( g[i] = r \), for \( 0 \leq i \leq m-1 \). A value \( g[i] \) is assigned if \( g[i] \neq r \), for \( 0 \leq i \leq m-1 \).

**Definition 3.1.** The function \( \text{rank} : V \mapsto \{0, \ldots, n-1\} \) computes the number of values assigned before a given vertex \( v \) in \( g \), which is uniquely associated with a key \( x \in S \). That is, \( \text{rank}(v) = |\{i \in V : i < v, g[i] \neq r\}| \).

We use the rank function to obtain a minimal perfect hash function \( \mu \) as follows:

\[
\mu(x) = \text{rank}(h(x)).
\]

The data structure to store the PHF \( h \) consists of \( h_0, \ldots, h_{r-1} \) and an array that contains the values \( g(v), \ v \in V \). We show in Section 3.5 that it is possible to get by with \( c(r)[\log (r)] \) bits per key to store the array \( g \). The value that minimizes the cost per key is \( r = 3 \). The MPHF \( \mu \) needs \( \epsilon m \) additional bits of space, \( 0 < \epsilon < 1 \), for the function rank.

We now illustrate the three steps of the RAM algorithm. Fig. 1 gives an overview to construct a PHF for a key set \( S \subset U \) containing three English words, i.e., \( S = \{\text{jan, feb, mar}\} \), based on an \( r \)-partite hypergraph with \( r = 3 \).

The Mapping step in Fig. 1(a) carries out two important tasks:

1. It assumes that it is possible to find three fully random functions, \( h_0, h_1, h_2 \), with ranges \( \{0, 1\}, \{2, 3\} \) and \( \{4, 5\} \), respectively, which corresponds to the three...
acyclic 3-partite hypergraph with
(b) The assigning step builds a perfect hash function from list $L$.

```
Fig. 1. (a) The mapping step builds from $S = \{\text{jan, feb, mar}\}$ a random acyclic 3-partite hypergraph with $m = 6$ vertices and $n = 3$ edges, and a list $\mathcal{L}$ of edges obtained when we test whether the hypergraph is acyclic. (b) The assigning step builds a perfect hash function from $S$ to \{0, 1, 2, 3, 4, 5\}, being represented by an array $g : \{0, 1, 2, 3, 4, 5\} \to \{0, 1, 2, 3\}$ to uniquely assign an edge to a vertex. (c) The ranking step builds the data structure used to compute function $\text{rank} : \{0, 1, 2, 3, 4, 5\} \to \{0, 1, 2\}$ in $O(1)$ time.
```

The assigning step places a one-to-one mapping of the key set $S$ to the edge set $E$ of a random acyclic 3-partite hypergraph $G = (V,E)$, where $|V| = m = 6$ and $|E| = n = 3$. To illustrate the mapping, the key "jan" is assigned to edge $(h_0("jan"), h_1("feb")) = (1, 3)$. Key "feb" is assigned to edge $(h_0("feb"), h_1("feb")) = (1, 2)$, and key "mar" is assigned to edge $(h_0("mar"), h_1("mar")) = (0, 2, 5)$.

We show later in this section that it is possible to obtain such a hypergraph with probability tending to 1 as $n$ tends to infinity whenever $m = \Omega(n)$ and $c \geq c(3)$. The value of $c$ that minimizes the hypergraph size (and thereby the amount of bits to represent the resulting functions) is $c = c(3) \approx 1.23$. If an acyclic hypergraph is not obtained, then a new set of three fully random hash functions is chosen to construct another hypergraph until an acyclic one is obtained.

2. It tests whether the resulting random 3-partite hypergraph contains cycles by iteratively deleting edges connecting vertices of degree 1. The deleted edges are stored in the order of deletion in a list $\mathcal{L}$ to be used in the assigning step. The first deleted edge in Fig. 1(a) was (1, 2, 4), the second one was (1, 3, 5) and the third one was (0, 2, 5). If it ends with an empty graph, then the test succeeds, otherwise it fails.

The assigning step in Fig. 1(b) outputs a PHF $h$ represented by $h_0$, $h_1$, and the array $g$ storing values from the range $\{0, 1, 2, 3\}$. Given a key $x$, the index $i(x) \in \{0, 1, 2\}$ is obtained by $i(x) = (g[h_0(x)] + g[h_1(x)] + g[h_2(x)]) \mod 3$. Next, $h_{i(x)}$ gives the position of key $x$ in the table. For instance, consider key "jan" in Fig. 1. The index $i("jan") = (g[1] + g[3] + g[5]) \mod 3 = 0$. Then, $h("jan") = h_0("jan") = 1$. Similarly, key "feb" is in position 4 of the table because $i("feb") = (g[1] + g[2] + g[4]) \mod 3 = 2$ and $h("feb") = h_2("feb") = 4$, and so on.

The ranking step in Fig. 1(c) outputs an MPHF. It uses the function rank. For instance, $\text{rank}(4) = 2$ because the positions 0 and 1 are assigned since the values of $g[0] \neq r$ and $g[1] \neq r$ for $r = 3$.

Before showing the mapping, assigning and ranking steps in details we need the following definitions.

**Definition 3.2.** The class $C = C_{U,S,m}$ consists of all functions $h : U \to \{0, \ldots, m-1\}$ that can be written as $h(x) = h_{i(x)}(x)$, where $i(x) = \left(\sum_{0 \leq i < r} g(h_i(x))\right) \mod r$,

where each $h_i$ is as in Eq. (1), and $g : V \to \{0, \ldots, r-1\}$ is some function that makes a function $h$ a PHF for $S$.

The class $C$ can be referred to as a "class of perfect hash functions" (for $U, S$ and $m$) in the sense that for every set $S$ of size $n$ (not too large) there is a function $h$ in $C$ that is one-to-one on $S$.

**Definition 3.3.** The class $C_\mu$ consists of all functions $\mu : U \to \{0, \ldots, n-1\}$ that can be written as $\mu(x) = \text{rank}(h(x))$,

where $h \in C_{U,S,m}$ is a PHF and rank is as in Definition 3.1.

The class $C_\mu$ can be referred to as a "class of minimal perfect hash functions" (for $U$, $S$ and $m$) in the sense that for every set $S$ of size $n$ (not too large) there is a function $\mu$ in $C_\mu$ that is one-to-one on $S$.

3.1. Construction of a PHF from class $C$

We detail now the mapping and assigning steps of the RAM algorithm to construct a PHF. Fig. 2 presents a pseudo-code for the mapping step. It takes a key set $S$ and a set $\mathcal{H}$ of hash functions that map $U$ into disjoint partitions of the vertex set of a hypergraph $G_S$, and returns an acyclic hypergraph $G_S$ and the list of edges $\mathcal{L}$. We use an edge-oriented data structure proposed in [24] to represent hypergraphs, where each edge is explicitly represented as an array of $r$ vertices and, for each vertex $v$, there is a list of edges that are incident to $v$. In line 2 the set of edges of the hypergraph $G_S$ is initialized as empty. The list of edges $\mathcal{L}$ is obtained in line 7 when we test whether $G_S$ is acyclic by iteratively deleting edges connecting vertices of degree 1. The list $\mathcal{L}$ stores the deleted edges in the order of deletions (i.e., the first edge in $\mathcal{L}$ was the first deleted edge, the second edge in $\mathcal{L}$ was the second deleted edge, and so on.) The following algorithm can do this test:

1. Traverse $G_S$ and store in a queue $Q$ every edge that has at least one of its vertices with degree one.
2. Until $Q$ is not empty, dequeue one edge from $Q$, remove it from $G_S$, store it in $\mathcal{L}$, and check if any of its vertices is now of degree one. If it is the case, enqueue the only edge that contains that vertex.

Fig. 3 presents a pseudo-code for the assigning step. It takes the hypergraph $G_S$ and the list of edges $\mathcal{L}$ as input, and returns the values in $g$. We first initialize $g[i] = r$ (i.e., each vertex is unassigned) and $\text{Visited}[i] = \text{false}$, for $0 \leq i \leq m-1$. Next, for each edge $e \in \mathcal{L}$ from tail to head, we look for the first vertex $u$ belonging to $e$ not yet visited and keep this information in the index $j$ of $u$ in $e$. Next, we set $g[u] = j - \sum_{v \in e} \text{Visited}[v] + r \mod r$. Whenever we visit a vertex $u$ from $e$ we set $\text{Visited}[u] = \text{true}$ if it has not been visited yet.
procedure Mapping ($S$, $H$, $G_r$, $\mathcal{L}$)
1. repeat
2. $E(G_r) = \emptyset$;
3. select $h_0, h_1, \ldots, h_{r-1}$ uniformly at random from $H$;
4. for each $x \in S$ do
5. \hspace{1em} $e = (h_0(x), h_1(x), \ldots, h_{r-1}(x))$;
6. \hspace{1em} addEdge $(G_r, e)$;
7. $\mathcal{L} = isAcyclic(G_r)$;
8. until $E(G_r)$ is empty

Fig. 2. Mapping step.

procedure Assigning ($G_r$, $\mathcal{L}$, $\eta$)
1. for $u = 0$ to $m-1$ do
2. \hspace{1em} Visited[$u$] = false;
3. \hspace{1em} $g[u] = r$;
4. for $i = |Z| \rightarrow 1$ to 0 do
5. \hspace{1em} $e = e[k]$;
6. \hspace{1em} sum = 0;
7. \hspace{1em} for $k = r \rightarrow 1$ to 0 do
8. \hspace{2em} if (not Visited[e[k]])
9. \hspace{3em} $u = e[k]$;
10. \hspace{3em} Visited[u] = true;
11. \hspace{3em} $j = k$;
12. \hspace{3em} else sum += g[e[k]];
13. \hspace{1em} $g[u] = (j - \text{sum}) \mod r$;

Fig. 3. Assigning step.

3.1.1. Amount of CPU time for the mapping step
In the mapping step presented in Fig. 2, line 2 has cost $O(1)$ and line 3 has cost $O(r)$ by assuming that each function $h_i \in H$ can be selected in $O(1)$ time. It is easy to see that line 5 has cost $O(r)$. In line 6 a given edge $e$ is inserted in $G_r$ with cost $O(r)$ (edge $e$ is inserted in $r$ lists of incident edges, one for each vertex in $e$). Considering that $|S| = n$ thus line 4 has cost $O(n)$ for $r = O(1)$. It is well known that the test to check whether $G_r$ is acyclic in line 7 can be implemented in $O(n)$ time (see, e.g., [17,36]). Therefore, each iteration of the mapping step takes $O(n)$ time. In the following section we show that the expected number of iterations of the repeat-until loop that starts in line 1 is $O(1)$.

3.1.2. Expected number of iterations for the mapping step
In this section we show that the expected number of iterations of the repeat-until loop that starts in line 1 of the mapping step in Fig. 2 is bounded by a constant. Since the construction of each hypergraph $G_r$ takes linear time and it is possible to choose $h_0, h_1, \ldots, h_{r-1}$ repeatedly (see Section 4.2), then the pseudo-code presented in Fig. 2 has expected linear time to construct an acyclic hypergraph $G_r$.

We now present the analysis of the algorithm to obtain a random acyclic $r$-partite hypergraph $G_r = G_r(h_0, h_1, \ldots, h_{r-1})$ with $n$ edges and $m$ vertices with high probability for $r \geq 2$. We will first analyze the case $r = 2$ and next the case $r \geq 3$.

**Theorem 3.4.** Consider $G_2 = (V, E)$ a bipartite random graph with $n$ edges and $m$ vertices. Then, if $m = \lfloor cn \rfloor$ holds for $c > 2$, in the limit when $m \rightarrow \infty$, the probability that $G_2$ is a forest (acyclic) tends to $Pr_2 = \sqrt{1 - (2/c)^2}$.

**Proof.** In this proof, every time we use the word “graph” we mean “bipartite random graph”. Consider $G_2(V, E) = G_{n, \eta}(V, E)$ a graph with $|V| = 2n = m$, and $|E| = d\eta = n$, where $d = n/\eta$ is the average degree of $G_{n, \eta}$. A random graph is obtained by a stochastic process where each graph starts with a set of $m = 2\eta$ vertices and at each step one edge is added between two vertices (one from each partition) at random. Different random graph models produce different probability distributions on graphs. Let $G_{2, \eta, p}$, $0 \leq p \leq 1$, be the model of all bipartite random graphs with $m = 2\eta$ vertices and the $\eta^2$ possible edges occur independently of each other, each with probability $p$. Another closely related model is the $G_{2, \eta, n}$ model which assigns equal probability to all bipartite graphs with exactly $m = 2\eta$ vertices and $n$ edges. It is well known in the random graph theory that results for $G_{2, \eta, n}$ are equivalent to results for $G_{2, \eta, p}$ whenever $p = d/\eta$ and $\eta \rightarrow \infty$ (this is equivalent to $m \rightarrow \infty$ and they can be interchangeable), because the expected number of edges of the graphs in $G_{2, \eta, n}$ would be $\eta^2 p = n$. The $n$ edges are almost surely distinct because there will be no multiple edges with probability $\eta^2/\eta^2 = 1$, where $\eta^2 = \eta(\eta - 1) \cdots (\eta - n + 1)$. In the limit, when $\eta \rightarrow \infty$, this probability tends to $e^{-d/2}$. To get this we used standard calculus to approximate $f(x) = 1 - x \times y(x) = e^{-x}$ for some real $x \in (0, 1)$.

Graph $G_{2, \eta, n}$ has no cycles when neither multiple edges nor cycles of even length larger than or equal to 4 occur. Let $c$ denote the event that the bipartite random graph has no cycles and $M$ denote the event that it has no multiple edges. Thus, to finalize the proof we need to show that: $Pr_2 = Pr(c \cap M) = Pr(c) Pr(M)$.

As mentioned above, no multiple edges occur with probability $e^{-d/2}$ as $\eta \rightarrow \infty$. We now need to determine the probability of constructing $G_{2, \eta}$ with no cycles of even length larger than or equal to 4 given that there is no multiple edges as $n \rightarrow \infty$. To build $G_{2, \eta}$, each edge is independently taken at random with probability $p$ from all $\eta^2$ possible edges. As there are $m = 2\eta$ vertices, and each vertex is connected to an average of $d$ edges, we can conclude that $p = d/\eta = 2d/m$.

Let $J_{2l}$ be the set of cycles of length $2l$ in the complete bipartite graph $K_{n, \eta}$ and $J = \bigcup_{l=2}^{\infty} J_{2l}$ be the set of all cycles. A cycle in $J_{2l}$ can be represented as a sequence of $2l$ distinct vertices in $K_{n, \eta}$. As each cycle can be represented in $2l$ ways by changing the start point, the cardinality of $J_{2l}$ is: $|J_{2l}| = 1/2l(\eta^2)^l$. As each edge in $G_{n, \eta}$ is selected independently of the others and with probability $p = 2d/m$, then each cycle in $J_{2l}$ occurs with probability $Pr_{2l}(d) = p^{2l}$.

Let $C_{2l}(G_{n, \eta})$ be a random variable that measures the number of cycles of length $2l$ in a graph $G_{n, \eta}$. Let $C_{2l}(G_{n, \eta})$ be a random variable that measures the number of cycles of any even length larger than or equal to 4 in $G_{n, \eta}$. The probability distributions of $C_{2l}(G_{n, \eta})$ and $C(d, G_{n, \eta})$ follow Poisson distributions with parameters $\lambda_{2l}$ and $\lambda_d$, respectively.

As mentioned above, no multiple edges occur with probability $e^{-d/2}$ as $\eta \rightarrow \infty$. We now need to determine the probability of constructing $G_{2, \eta}$ with no cycles of even length larger than or equal to 4 given that there is no multiple edges as $n \rightarrow \infty$. To build $G_{2, \eta}$, each edge is independently taken at random with probability $p$ from all $\eta^2$ possible edges. As there are $m = 2\eta$ vertices, and each vertex is connected to an average of $d$ edges, we can conclude that $p = d/\eta = 2d/m$.

Let $J_{2l}$ be the set of cycles of length $2l$ in the complete bipartite graph $K_{n, \eta}$ and $J = \bigcup_{l=2}^{\infty} J_{2l}$ be the set of all cycles. A cycle in $J_{2l}$ can be represented as a sequence of $2l$ distinct vertices in $K_{n, \eta}$. As each cycle can be represented in $2l$ ways by changing the start point, the cardinality of $J_{2l}$ is: $|J_{2l}| = 1/2l(\eta^2)^l$. As each edge in $G_{n, \eta}$ is selected independently of the others and with probability $p = 2d/m$, then each cycle in $J_{2l}$ occurs with probability $Pr_{2l}(d) = p^{2l}$.

Let $C_{2l}(G_{n, \eta})$ be a random variable that measures the number of cycles of length $2l$ in a graph $G_{n, \eta}$. Let $C_{2l}(G_{n, \eta})$ be a random variable that measures the number of cycles of any even length larger than or equal to 4 in $G_{n, \eta}$. The probability distributions of $C_{2l}(G_{n, \eta})$ and $C(d, G_{n, \eta})$ follow Poisson distributions with parameters $\lambda_{2l}$ and $\lambda_d$, respectively.
restricted to be in the half-open interval $[0, 0.294)$ and very close to 0.294. For example, when $c = 2.09$ we have $\Pr_G = 0.29$. This is very close to 0.294—the value we obtained experimentally that follows a geometric distribution. The probability of obtaining a random acyclic $r$-partite hypergraph with cycles occurs we abort and randomly select a new tuple of hash functions $(h_0, h_1, ..., h_{r-1})$. We can model the number of iterations to obtain a random acyclic $r$-partite hypergraph $G_r = (V, E)$ as a random variable $Z$ that follows a geometric distribution. The probability $Pr_z$ of obtaining a random acyclic $r$-partite hypergraph is $\Omega(1)$ in the limit. Thus, $Pr(Z = i) = Pr_z(1 - Pr_z)^{i-1}$ and the mean of $Z$ is $1/Pr_z$, which corresponds to the expected number of iterations to obtain $G_r$. Therefore, as $Pr_z$ is $\Omega(1)$, the expected number of iterations is $O(1)$.

Finally, it is important to remark that the two values of interest are $r = 2$ and $r = 3$. The use of a 3-graph constructs more compact PHFs and MPHFs at the expense of one more hash function $h_2$ and one more random memory access. Thus, our best result in terms of space consumption to store PHFs and MPHFs is for $r = 3$.

Then, in the limit when $m \to \infty$, the probability that $G_r$ is a forest (acyclic) tends to 1. In the case $c \leq c(r)$ the probability tends to zero.

From Theorems 3.4 and 3.5 we can conclude that, under the right choice of $c$, it is possible to construct a random acyclic $r$-partite hypergraph with probability bounded away from zero. The value of $c(r)$ in Eq. (4) is minimized for $r = 3$ and is in the open interval $(1, 2.22, 1.23)$. This is illustrated in Fig. 4, which was previously reported in [36]. This means that the random acyclic $r$-partite hypergraphs with the smallest number of vertices occur when $r = 3$. For $c = 1.23$ we obtained experimentally that the number of iterations to obtain an acyclic hypergraph $G_3 = (V, E)$ is close to 1 ($Pr_z$ is close to 1). In our experiments, we constructed 1000 random 3-partite $G_3 = (V, E)$ hypergraphs with $n = 10^7$ keys, and of the 1000 hypergraphs 998 were acyclic.

The problems of obtaining random acyclic $r$-partite hypergraphs for $r = 2$ and for $r > 2$ are quite different. For $r = 2$, the probability $Pr_z$ varies continuously with constant $c$. But for $r > 2$, there is a phase transition when $m$ tends to $\infty$: there is a value $c(r)$ such that (i) if $c \leq c(r)$, then $Pr_z$ tends to 0 and (ii) if $c > c(r)$, then $Pr_z$ tends to 1. This phenomenon has also been reported in [36] for random $r$-uniform hypergraphs.

We now show that the expected number of iterations of the mapping step is bounded by a constant under the right choice of $c$, according to Theorems 3.4 and 3.5. When a random $r$-partite hypergraph with cycles occurs we abort and randomly select a new tuple of hash functions $(h_0, h_1, ..., h_{r-1})$. We can model the number of iterations to obtain a random acyclic $r$-partite hypergraph $G_r$ as a random variable $Z$ that follows a geometric distribution. The probability $Pr_z$ of obtaining a random acyclic $r$-partite hypergraph is $\Omega(1)$ in the limit. Thus, $Pr(Z = i) = Pr_z(1 - Pr_z)^{i-1}$ and the mean of $Z$ is $1/Pr_z$, which corresponds to the expected number of iterations to obtain $G_r$. Therefore, as $Pr_z$ is $\Omega(1)$, the expected number of iterations is $O(1)$.

For example, when $c = 2.09$ we have $Pr_z = 0.29$. This is very close to 0.294—the value we obtained experimentally that follows a geometric distribution. The probability of obtaining a random acyclic $r$-partite hypergraph with cycles occurs we abort and randomly select a new tuple of hash functions $(h_0, h_1, ..., h_{r-1})$. We can model the number of iterations to obtain a random acyclic $r$-partite hypergraph $G_r$ as a random variable $Z$ that follows a geometric distribution. The probability $Pr_z$ of obtaining a random acyclic $r$-partite hypergraph is $\Omega(1)$ in the limit. Thus, $Pr(Z = i) = Pr_z(1 - Pr_z)^{i-1}$ and the mean of $Z$ is $1/Pr_z$, which corresponds to the expected number of iterations to obtain $G_r$. Therefore, as $Pr_z$ is $\Omega(1)$, the expected number of iterations is $O(1)$.

We now show that the expected number of iterations of the mapping step is bounded by a constant under the right choice of $c$, according to Theorems 3.4 and 3.5. When a random $r$-partite hypergraph with cycles occurs we abort and randomly select a new tuple of hash functions $(h_0, h_1, ..., h_{r-1})$. We can model the number of iterations to obtain a random acyclic $r$-partite hypergraph $G_r$ as a random variable $Z$ that follows a geometric distribution. The probability $Pr_z$ of obtaining a random acyclic $r$-partite hypergraph is $\Omega(1)$ in the limit. Thus, $Pr(Z = i) = Pr_z(1 - Pr_z)^{i-1}$ and the mean of $Z$ is $1/Pr_z$, which corresponds to the expected number of iterations to obtain $G_r$. Therefore, as $Pr_z$ is $\Omega(1)$, the expected number of iterations is $O(1)$.

We now show that the expected number of iterations of the mapping step is bounded by a constant under the right choice of $c$, according to Theorems 3.4 and 3.5. When a random $r$-partite hypergraph with cycles occurs we abort and randomly select a new tuple of hash functions $(h_0, h_1, ..., h_{r-1})$. We can model the number of iterations to obtain a random acyclic $r$-partite hypergraph $G_r$ as a random variable $Z$ that follows a geometric distribution. The probability $Pr_z$ of obtaining a random acyclic $r$-partite hypergraph is $\Omega(1)$ in the limit. Thus, $Pr(Z = i) = Pr_z(1 - Pr_z)^{i-1}$ and the mean of $Z$ is $1/Pr_z$, which corresponds to the expected number of iterations to obtain $G_r$. Therefore, as $Pr_z$ is $\Omega(1)$, the expected number of iterations is $O(1)$.

We now show that the expected number of iterations of the mapping step is bounded by a constant under the right choice of $c$, according to Theorems 3.4 and 3.5. When a random $r$-partite hypergraph with cycles occurs we abort and randomly select a new tuple of hash functions $(h_0, h_1, ..., h_{r-1})$. We can model the number of iterations to obtain a random acyclic $r$-partite hypergraph $G_r$ as a random variable $Z$ that follows a geometric distribution. The probability $Pr_z$ of obtaining a random acyclic $r$-partite hypergraph is $\Omega(1)$ in the limit. Thus, $Pr(Z = i) = Pr_z(1 - Pr_z)^{i-1}$ and the mean of $Z$ is $1/Pr_z$, which corresponds to the expected number of iterations to obtain $G_r$. Therefore, as $Pr_z$ is $\Omega(1)$, the expected number of iterations is $O(1)$.

We now show that the expected number of iterations of the mapping step is bounded by a constant under the right choice of $c$, according to Theorems 3.4 and 3.5. When a random $r$-partite hypergraph with cycles occurs we abort and randomly select a new tuple of hash functions $(h_0, h_1, ..., h_{r-1})$. We can model the number of iterations to obtain a random acyclic $r$-partite hypergraph $G_r$ as a random variable $Z$ that follows a geometric distribution. The probability $Pr_z$ of obtaining a random acyclic $r$-partite hypergraph is $\Omega(1)$ in the limit. Thus, $Pr(Z = i) = Pr_z(1 - Pr_z)^{i-1}$ and the mean of $Z$ is $1/Pr_z$, which corresponds to the expected number of iterations to obtain $G_r$. Therefore, as $Pr_z$ is $\Omega(1)$, the expected number of iterations is $O(1)$.

We now show that the expected number of iterations of the mapping step is bounded by a constant under the right choice of $c$, according to Theorems 3.4 and 3.5. When a random $r$-partite hypergraph with cycles occurs we abort and randomly select a new tuple of hash functions $(h_0, h_1, ..., h_{r-1})$. We can model the number of iterations to obtain a random acyclic $r$-partite hypergraph $G_r$ as a random variable $Z$ that follows a geometric distribution. The probability $Pr_z$ of obtaining a random acyclic $r$-partite hypergraph is $\Omega(1)$ in the limit. Thus, $Pr(Z = i) = Pr_z(1 - Pr_z)^{i-1}$ and the mean of $Z$ is $1/Pr_z$, which corresponds to the expected number of iterations to obtain $G_r$. Therefore, as $Pr_z$ is $\Omega(1)$, the expected number of iterations is $O(1)$.
3.1.3. Amount of CPU time for the assigning step

In Fig. 3, the for loop that starts in line 1 has cost \( O(m) \) and the for loop that starts in line 4 has cost \( O(n) \). As the number of vertices in \( G_r \) is a linear function of the number of edges (i.e., \( m = \lfloor cn \rfloor \) for some constant \( c \)), then, for \( r = O(1) \), the assigning step runs in \( O(n) \) time.

3.2. PHF evaluation

Fig. 5 presents the pseudo-code to evaluate a PHF.

3.2.1. Amount of CPU time to evaluate a PHF

The cost to evaluate the PHF presented in Fig. 5 is \( O(r) \). The practical instances are for \( r = 2 \) and \( r = 3 \) and then the cost to evaluate the PHF is \( O(1) \).

3.3. Construction of an MPHF from class \( C_\mu \)

We now detail the ranking step of the algorithm to construct an MPHF. The function rank is used to compress the range of the PHFs in class \( C \) from \( [0, \ldots, m-1] \) to \( [0, \ldots, n-1] \) to obtain a class \( C_\mu \) of MPHFs. For the implementation of the function \( \text{rank} \) we use a simple and efficient algorithm from [42]. The function rank can be computed in \( O(1) \) time (see, e.g., [40]). The algorithm computes two tables: rankTable and \( T_r \). The table rankTable explicitly stores the \( \text{rank} \) of every \( k \)th index, where \( k = \lfloor \log(m)/e \rfloor \), using \( cm \) additional bits of space, for \( 0 < e < 1 \). The larger \( k \) is the more compact is the MPHF, and the users can trade off space for evaluation time by setting \( k \) appropriately in the implementation. We use values for \( k \) that are powers of two (i.e., \( k = 2^h \) for some constant \( h_0 \)) in order to replace the expensive division and modulo operations by bit-shift and bitwise “and” operations, respectively. Fig. 6 presents a pseudo-code to construct the rankTable taking as input \( g \) and \( k \).

We use \( k = 256 \) in our implementation. The amount of CPU time for constructing the rankTable is \( O(n) \) because line 2 in Fig. 6 just loops over the \( m = \lfloor cn \rfloor \) entries of the array \( g \), performs operations in \( O(1) \), and \( c \) is a constant.

The table \( T_r \) is a lookup table that allows to count in constant time the number of assigned vertices (\( g[i] \neq r \), for \( 0 \leq i \leq m-1 \)) in \( \rho = e \log m \) bits, where \( 0 < e < 1 \). Thus, the actual evaluation time is \( O(1/e) \). We use the value of \( \rho \) as a multiple of the number of bits \( \beta \) used to encode each entry of \( g \). As the values in \( g \) come from the range \( [0, 1, 2, 3] \), then \( \beta = 2 \) bits and we use \( \rho = 8 \). Each entry of \( T_r \) counts the number of assigned vertices in a single byte.

Fig. 7 presents the pseudo-code to construct table \( T_r \), where \( LSi[i, \beta] \) stands for the value of the \( \beta \) least significant bits of \( i \) and \( \gg \) is the right shift of bits. The table \( T_r \) fits entirely in the CPU cache of a commodity PC because it takes \( 2^8 \) bytes of space. We remark that each \( r \geq 2 \) requires a different lookup table \( T_r \).

Fig. 8 presents the pseudo-code to evaluate an MPHF. The value of \( u \) in \( g \) is given by the perfect hash function \( \text{phf} \) presented in Fig. 5. The rank of \( u \) in \( g \) is obtained in two steps: (i) perform a look up in rankTable to obtain the rank of the largest precomputed index \( v \leq u \) in \( g \), and (ii) count the number of assigned vertices from position \( v \) to \( u-1 \) using table \( T_r \). We use the notation \( g[i,j] \) to represent the values stored in the entries from \( g[i] \) to \( g[j] \) for \( i \leq j \).

3.3.1. Amount of CPU time to evaluate an MPHF

In Fig. 8, the value of \( u \) is given by the perfect hash function \( \text{phf} \) in \( O(1) \) time. The function \( \text{rank} \) can be computed in \( O(1) \) time. Thus, the evaluation of the \( \text{mphf} \) costs \( O(1) \) time.

3.4. Space consumption to construct PHFs and MPHFs

We now show that the RAM algorithm needs \( O(n) \) computer words to construct functions of classes \( C \) or \( C_\mu \). We assume that the key set \( S \) is kept in external memory and just the data structures involved in the construction process are kept in internal memory. We maintain the following data structures in internal memory: (i) \( r \) hash functions \( h_0, h_1, \ldots, h_{r-1} \). Each function can be described in \( o(n) \) bits by using the split-and-share technique [21,22]; (ii) a random acyclic \( r \)-partite hypergraph \( G_r \). As \( m = \lfloor cn \rfloor \), it is possible to store \( G_r \) in \( O(m) \) computer words by using the data structure proposed in [24]; (iii) a list \( L \) of deleted edges obtained when we test whether \( G_r \) is a forest, stored in \( O(m) \) computer words; and (iv) a resulting function \( h \). This corresponds to \( \beta m \) bits if \( h \in C \) and \( (\beta + c)m + o(m) \) bits if \( h \in C_\mu \) (the values of \( \beta \) and \( c \) are presented in
Sections 3.5 and 3.6, respectively). Therefore, for \( r = O(1) \), we need \( O(n) \) computer words to construct the functions.

3.5. Space consumption to store a PHF

The data structure used to construct a PHF from class \( C \) consists of \( h_0, \ldots, h_{r-1} \) and the values of \( g \). Since \( r \) is a small constant, the number of bits needed to store a PHF is \( O(m) \). Actually, \( \beta = \lceil \log(r+1) \rceil \) bits are sufficient for each value of \( g \). Therefore, \( g \) requires \( \beta m \) bits of storage space. The representation of the \( r \) hash functions in \( o(n) \) bits uses the split-and-share technique (see Section 4 for details). In the following sections we discuss the 2-graph and 3-graph instances used to construct PHFs.

3.5.1. 2-Graph instance

The use of an acyclic bipartite 2-graph yields PHFs in the range \( \{0, \ldots, m-1\} \), where \( m = \lceil cn \rceil \) for \( c > 2 \) (see Section 3.1.2). For \( r=2 \), the values assigned to the vertices are drawn from \( \{0, 1\} \) and \( \beta = 1 \) bit is needed to represent the value assigned to each vertex. Therefore, the resulting PHF requires \( m + o(n) \) bits to be stored. For \( c=2.09 \), the resulting PHFs are stored in approximately \( \lceil 2.09n \rceil + o(n) \) bits and map to the range \( \{0, \ldots, 2.09n \} \).

3.5.2. 3-Graph instance

An acyclic 3-partite random 3-graph yields PHFs in the range \( \{0, \ldots, m-1\} \), where \( m = \lceil cn \rceil \) for \( c \approx 1.23 \). For \( r=3 \), the values assigned to the vertices are drawn from \( \{0, 1, 2\} \) and \( \beta = 2 \) bits are needed to represent the value assigned to each vertex. For \( c=1.23 \), the resulting PHFs are stored in approximately \( \lceil 2.46n \rceil + o(n) \) bits and map to the range \( \{0, \ldots, 1.23n \} \).

If we replace the special value \( r=3 \) by 0 in the array \( g \) (since \( r \equiv 0 \pmod r \), arithmetically \( r \) is the same as 0), then the value of \( \lceil 2.46n \rceil + o(n) \) bits can be compressed to \( \lceil 1.95n \rceil + o(n) \) bits using arithmetic coding. The values assigned to every group of 5 vertices can be packed into one byte because each assigned value comes from a range of size 3 and \( 3^5 = 243 < 256 = 2^8 \). At construction time we use a small lookup table containing: \( \text{pow3_table}[5] = \{1, 3, 9, 27, 81\} \). A value \( x \in \{0, 1, 2\} \) is assigned to a vertex \( u \in V \) as follows:

\[
\begin{align*}
\text{byte} & = g[u/5]; \\
\text{byte}+ & = x \times \text{pow3_table}[u \mod 5]; \\
g[u/5] & = \text{byte};
\end{align*}
\]

A lookup table \( T_{\text{lookup}} \) of size \( 5 \times 256 = 1280 \) bytes is used to speed up the recovery of the value \( x \) assigned to a given vertex \( u \), as follows:

\[
\begin{align*}
\text{byte} & = g[u/5]; \\
x & = T_{\text{lookup}}[u \mod 5][\text{byte}];
\end{align*}
\]

Each entry of table \( T_{\text{lookup}} \) is computed by: \( T_{\text{lookup}}[i] = (i/\text{pow3_table}[i]) \mod 3 \), where \( 0 \leq i < 5 \) and \( 0 \leq j < 256 \). Now each value in \( g \) is compressed to be stored in \( \beta = \log(3) \) bits. Therefore, the space usage comes from the multiplication of \( \beta \) by the number of entries in \( g \), which is \( m = \lceil 1.23n \rceil \).

3.6. Space consumption to store an MPHF

The data structure used to construct an MPHF from class \( C_n \) consists of \( h_0, \ldots, h_{r-1} \), the values of \( g \), the rankTable, and the lookup table \( T_r \). The rankTable is stored in \( \lceil cm \rceil \) bits because it has \( m/k \) entries, each of size \( \log m \) bits and \( k = \lceil \log(m)/c \rceil \) for \( 0 < c < 1 \). The lookup table \( T_r \) is stored in \( o(m) \) bits because it has \( m^r \) entries, each of size \( \log \log m^r \) bits. Putting all together the number of bits required to store the MPHF is \( (\beta + c)m + o(m) \) bits. In the following sections we discuss the 2-graph and 3-graph instances that are used in the construction of MPHFs.

3.6.1. 2-Graph instance

The MPHF requires \( (2+c)m + o(m) \) bits to be stored, which corresponds to \( (2+c)(cn) + o(n) \) bits for any \( c > 0 \) and \( c > 2 \). For \( c = 0.125 \) and \( c = 2.09 \), the MPHF requires approximately 4.44n bits. We note that in this case the values of \( g \) also come from the range \( \{0, 1, 2\} \). Therefore we can use the same packing scheme presented in Section 3.5.2 to yield MPHFs that require \( (\log(3) + c)(cn) + o(n) \) bits to be stored. For \( c = 0.125 \) and \( c = 2.09 \), an MPHF is stored in approximately 3.6n bits.

3.6.2. 3-Graph instance

In this case the values of \( g \) are from the range \( \{0, 1, 2, 3\} \). Thus, we must use \( \beta = 2 \) bits for each entry of \( g \). The MPHF requires then \( (2+c)(cn) + o(n) \) bits to be stored for any \( c > 0 \) and \( c > 1.23 \). For \( c = 0.125 \) and \( c = 1.23 \), an MPHF is stored in approximately 3.62n bits.

4. Using split-and-share to simulate uniform hash functions

In this section we use the split-and-share approach presented in [21,22] to simulate uniform hash functions using \( o(n) \) bits of extra space for a small key set of size \( n \). To “simulate uniform hashing” means to construct for a given parameter \( n \) a random hash function \( h : U \rightarrow (0, 1, \ldots, m-1) \) that has the following properties for any set \( S \subset U \) of size \( n \): all keys \( x \in S \) are uniformly distributed over \( \{0, 1, \ldots, m-1\} \) with high probability. In the EM algorithm presented in Section 5, the input key set is partitioned into small buckets and the RAM algorithm is used to compute an MPHF for each bucket where we use the hash functions designed in this section.

Comparing with the implementation described in [21,22] our construction has two differences. First, it constructs a class of hash functions that are used by the RAM algorithm in each bucket. Second, the split function generates buckets that are provably small—a fact that we exploit in the implementation to take into account the memory hierarchy in an efficient way.

4.1. Splitting

The first ingredient we need is a hash function that maps the keys of \( S \) to \( N_b = 2^g \) buckets, such that all buckets are of approximately the same size. If a uniform hash function is used and \( N_b < n/\log n \), it is well known
that the largest bucket will contain $O(n/N_b)$ keys with high probability [2]. Most explicitly defined classes of hash functions (e.g., universal [12] or polynomial classes of hash functions [19]) have much weaker guarantees. For instance, if a function $h$ is picked uniformly from a class of universal hash functions is used to hash a set $S$ of size $n$ into $n$ buckets, the expected size of the largest bucket is less than $\sqrt{n} + 1/2$, whereas if a uniform hash function $h$ is used the expected size of the largest bucket is exponentially lower: $O(\log n/\log \log n)$ [2]. However, Alon et al. [2] showed that if we fix a concrete class of universal hash functions, it is possible to considerably diminish the loss by using universal hash functions. Let $B = \{B_i \mid B_i = \{x \in S \mid q(x) = i\}\}$ be a set of buckets induced by a function $q : S \rightarrow \{0, 1\}^k$, where $k$ is as defined in Theorem 4.1 (a result presented in [2]).

**Theorem 4.1** (Alon et al. [2]). Let $\mathcal{H}_{L,b}$ be the class of all linear transformations over GF(2), the field of two elements, mapping $\{0,1\}^l$ to $\{0,1\}^k$. Let $N_b = 2^b$ and restrict that $N_b \leq n/\log n$. Let $S \subseteq \{0, 1\}^l$ be a set of size $n$, and choose $q \in \mathcal{H}_{L,b}$ uniformly at random. Then the expected size of the largest bucket when hashing $S$ using $q$ is $O(n \log \log(n)/N_b)$.

The theorem says that the expected size of the largest bucket is within a factor $O(\log \log n)$ of the average bucket size. Hence a function from $\mathcal{H}_{L,b}$ can split the set into $O(n \log \log(n)/\ell)$ buckets of maximum size $\ell$. Thus, for a given constant $\kappa > 0$ we have

$$b \leq \log n + \log \log n - \log \ell + \log \kappa. \quad (5)$$

For the EM algorithm to construct functions with space complexity $O(n)$ bits we have the restriction $N_b \leq n/\log n$ according to Theorem 4.1, and then

$$\ell \geq \kappa \log n \log \log n. \quad (6)$$

Our construction is engineered to work with maximum bucket size $\ell = 256$. However, to keep the space at $O(n)$ bits the values of $b$ and $\ell$ are constrained by Eqs. (5) and (6), respectively. Therefore, for extremely large sets, a larger maximum bucket size is needed. The point where the maximum bucket size has to change depends on both $n$ and the constant $\kappa$.

Let $q : \{0, 1\}^l \rightarrow \{0, 1\}^k$ be a function from the class $\mathcal{H}_{L,b}$ of Theorem 4.1 with the following form: $q(x) = Ax$, where $A$ is a $b \times L$ matrix with entries in GF(2). To represent $q$ we need to store the $bl$ bits of the matrix $A$. A matrix–vector product $Ax$ can be implemented by adding the columns corresponding to values 1 in $x$. Note that addition of vectors over GF(2) corresponds to bit-wise exclusive-or. For example, let us consider $L = 3$ bits, $b = 3$ bits, $x = 110$ and

$$A = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}.$$ 

Then

$$q(x) = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}.$$

The evaluation time for this is $O(L)$, assuming that a column vector can be stored in one machine word. To obtain faster evaluation we use a tabulation idea from [3] that gives evaluation time $O(L/\log \sigma)$ by using space $O(\sigma |L/\log \sigma| \log n)$ for $\sigma > 0$, where the $O(\log n)$ factor is due to the fact that the tables store numbers of $O(\log n)$ bits. Note that if $x$ is short, e.g., 8 bits, we can simply tabulate all the function values and compute $q(x)$ by looking up the value in an array. To make the same thing work for longer keys, split the matrix $A$ into parts of 8 columns each: $A = A_1 \mid A_2 \mid \ldots \mid A_{L/8}$, and create a lookup table $q_i$ for each sub-matrix. Similarly split $x$ into parts of 8 bits, $x = x_1x_2 \ldots x_{L/8}$. Now $q(x)$ is the bit-wise exclusive or of $q_i[x_i]$, for $i = 1, \ldots, [L/8]$. Therefore, we have set $\sigma$ to 256 so that keys of size $L$ can be processed in chunks of $\log \sigma = 8$ bits. Observe that all zero characters in a string can simply be skipped because the corresponding column vectors will all be zeroed after multiplying them with matrix $A$ and therefore their contribution to the matrix–vector product will be zero. This means that the evaluation time is proportional to the number of characters in the input string.

### 4.2. Simulating uniform hash functions

The second ingredient of split-and-share is a single hash function $f$ that, when applied to the keys of a single bucket, behaves like a fully random hash function with high probability. Then, this function can be shared among all buckets. As stated earlier, we construct a class of hash functions such that for any bucket, each function behaves like a fully random function with high probability. Technically, this is done by making $f$ a function of two parameters (see Eq. (7)), where the second parameter $s$ describes which function in the class is used.

#### 4.2.1. Shared function

Let $y_1, \ldots, y_k$ be independently chosen functions from a pairwise independent class of functions (see Definition 2.2) from $\{0, 1\}^k$ to $\{0, 1\}^k$, where $2^k > \ell$ is a parameter to be chosen later. If we randomly choose a function $y$ from this class and two different elements from its domain, say $x, z \in \{0, 1\}^l$, then the probability of collision is: $\Pr[y(x) = y(z)] = 1/2^k$. This assumption is satisfied by the class of functions of Theorem 4.1. Also, let $p$ be a prime number, and $k$ a positive integer. Consider tables $t_1, \ldots, t_k$ and $t_{i_1}, \ldots, t_{i_p}$ contain $2^k$ random values from $\{0, \ldots, p-1\}$. We will use a variation of a class of functions $f : \{0, 1\}^l \times \{1, \ldots, p-1\} \rightarrow \{0, \ldots, p-1\}$ due to [23] that achieves full independence with high probability on small sets

$$f(x, s) = \sum_{j=1}^k (t_j[y_j(x)] + s \times t_j[y_j(x)]) \mod p. \quad (7)$$

The independence property we need is captured by the following lemma.

**Lemma 4.2.** Consider $B_i$ as a set of keys in bucket $i$, $0 \leq i < N_b$. For any $s_i, s'_i \in \{1, \ldots, p-1\}$, $s_i \neq s'_i$, $B_i \subseteq S$ of size $|B_i|$, the following holds: With probability at least $1 - |B_i|/|B_i|^k$ over the choice of $y_1, \ldots, y_k$, the function

$$f(x, s) = \sum_{j=1}^k (t_j[y_j(x)] + s \times t_j[y_j(x)]) \mod p. \quad (7)$$

The independence property we need is captured by the following lemma.
values \( f(x,s), x \in B_i, s \in \{s_i,s'_i\} \) are independent and uniformly distributed in \([0,\ldots,p-1]\).

**Proof.** Consider arbitrary values \( v_{x,s} \in \{0,\ldots,p-1\} \), for \( x \in B_i, s \in \{s_i,s'_i\} \). Independence means that the probability that \( f(x,s) = v_{x,s} \) for all \( x \in B_i, s \in \{s_i,s'_i\} \) is \( p^{-2|B_i|} \). To arrive at a sufficient condition for independence, consider how the entries of \( t_1,\ldots,t_k \) and \( t'_1,\ldots,t'_k \) are accessed when computing \( f(x,s) \) for \( x \in B_i \) and \( s \in \{s_i,s'_i\} \). Assume that a key \( x \in B_i \) has a unique entry \( y_j(x) \) in \( t_j \) and \( t'_j \), which is not accessed when evaluating \( f \) on keys in \( B_i \). Then for any choice of values in other entries, the values \( f(x,s_i) \) and \( f(x,s'_i) \) are independent and uniformly distributed in \([0,\ldots,p-1]\). This is because there is exactly one choice of \( t_j[y_j(x)] \) and \( t'_j[y_j(x)] \) for each value of the pair \( v_{x,s_i}, v_{x,s'_i} \) (two independent linear equations with two variables in GF(p)). In conclusion, a sufficient condition for independence is that we can assign a unique entry \( y_j(x) \) to each \( x \in B_i \).

Since \( y_1,\ldots,y_{k} \) are chosen from a pairwise independent class of hash functions we know that for any \( x \in B_i \), the probability that \( x \) does not have a unique entry is at most \( (|B_i|/2)^k \). By the union bound, the probability that **some** key in \( B_i \) does not have a unique entry is at most \( |B_i|(|B_i|/2)^k \) and the lemma holds. \( \square \)

### 4.2.2. Using the shared function

We want to use the shared function to implement the RAM algorithm on the buckets. In fact, we will use three independent shared functions \( f_0,f_1,f_2 \), one for each hash function needed by the RAM algorithm. However, for reasons explained in the following all three functions will use the same functions \( y_1,\ldots,y_{k} \). Naturally each function \( f_j \) has a different set of tables \( t_j \) and \( t'_j \), where \( 0 \leq i \leq 2 \) and \( 1 \leq j \leq k \).

**Definition 4.3.** Consider \( |B_i| \) the number of keys mapped by \( q \) to bucket \( B_i \) and \( m_i = \lfloor |B_i|/3 \rfloor \), for \( c \geq 1.23 \), then:

\[
h_j(x) = f_j(x,s_i) \mod m_i + j \times m_i, \quad 0 \leq j \leq 2.
\]

The variable \( s_i \) is specific for bucket \( i \). This means it is chosen uniformly at random and stored. This costs storage space of \( \log p = O(\log n) \) bits and therefore we make sure that there are not too many buckets \( (N_b < n/\log n) \). The algorithm randomly selects \( s_i \) from \{1,\ldots,p-1\} until the functions \( h_{0,b}, h_{1,b} \), and \( h_{2,b} \) work with the RAM algorithm, which is used to construct a PHF or an MPHF for each bucket \( i \). We will prove in Section 4.2.3.3 that a constant fraction of the set of all choices of \( s_i \) works with high probability.

In the implementation we have focused on ways to make the memory access pattern more local when computing \( h_{0,b}, h_{1,b}, h_{2,b} \). This is to make better use of the CPU cache. The idea is that the tables used for storing the function descriptions are merged, such that all six values looked up using \( y_j(x) \) (two in each function \( f_j \), where \( 0 \leq j \leq 2 \)) are stored in consecutive memory locations, and so on for \( y_2(x),\ldots,y_k(x) \).

### 4.2.3. Analysis of the shared function

By Lemma 4.2 the probability that we fail to get a class of fully random hash functions for all buckets is at most \( \sum_i |B_i|(|B_i|/2)^k \leq n(\ell/2)^k \). If we choose, for example, \( \delta = \lceil \log \sqrt{ri} \rceil \) and \( k \geq 4, \) this probability is \( o(1/n) \). Then, the shared function will succeed with high probability, i.e., with probability \( 1-o(1/n) \).

Finally, we need to show that it is possible to obtain, with high probability, a value of \( s_i \) such that the functions \( h_{0,b}, h_{1,b} \), and \( h_{2,b} \) (see Definition 4.3) make the RAM algorithm work for \( B_i \). There are two issues. First, the functions \( h_{0,b}, h_{1,b} \), and \( h_{2,b} \) do not produce values that are exactly uniformly distributed in \([0,\ldots,m_i-1]\), because \( m_i \) does not divide \( p \). However, it is not hard to see that the probability of a particular set of hash functions values (or, in the analysis of RAM, of a particular 3-graph) is close to the probability in the uniformly distributed case. More specifically, the probability is at most a factor \( e^{B_i/p} \) higher, because the probability of getting a given set of hash values is upper bounded by \( |p/m_i|^3|B_i|/|p|^3|B_i| \leq (1+\varepsilon)^{B_i/|B_i|} \leq e^{C\kappa B_i/|B_i|} \). Since \( 3m_i \approx |B_i| \) and \( p \geq \ell^2 \), the failure probability will be very close to the uniform case.

The second issue is to show that even though any single choice of \( s_i \) makes the RAM algorithm fail with a constant probability \( \Theta < 1 \), with high probability there are many values of \( s_i \) that will make the RAM algorithm work. We may assume that the choice of \( y_1,\ldots,y_{k} \) was successful, i.e., that all functions in Definition 4.3 are fully random on all buckets. Let \( X \) be a random variable that counts the number of choices of \( s_i \) that makes the RAM algorithm to fail (i.e., the three hash functions lead to a 3-graph that contains cycles). Thus, the expectation \( E[X] = \Theta \), since there are \( p \) possible values for \( s_i \). Lemma 4.2 tells us that the events that the hash functions fail, for any two different values of \( s_i \), are independent. This means that \( Var(X) = Var(X) \leq \Theta \). By Chebyshev’s inequality (see e.g. [39]) then says that the probability that more than \( p(1+\Theta) \) hash functions fail is bounded by \( 4\Theta \).

### 4.2.4. Implementation details

The class of linear hash functions over GF(2) enables us to compute the functions \( q, y_1, y_2, y_3,\ldots, y_{k} \) in parallel. The idea is to take a linear function \( h' : \{0,1\}^k \rightarrow \{0,1\}^m \) that is chopped into (disjoint) parts. The first part has \( b \) bits and corresponds to the value of \( q(x) \). The remaining parts have \( \delta \) bits and correspond to the values of \( y_i, 1 \leq i \leq k \). Clearly, these functions will be independent.

We use an one-to-one function \( h' \) to map the keys from \( S \) to a \( \gamma \)-bit fingerprint set \( F \). As the function \( h' \) comes from a class of universal hash functions [2], the probability that there exist two keys that have the same values under all functions is at most \( \gamma^k/2^{\gamma^2+\delta} \), which is a negligible value when we choose \( k \) and \( \delta \) as in the beginning of Section 4.2.3.

The value of \( \gamma \) must be encoded by at least \( b+\delta \) bits so that a single fingerprint will be able to represent the values computed by the functions \( q, y_1, y_2, y_3,\ldots, y_{k} \). In the implementation we used \( \gamma = 96 \) bits. For a fingerprint \( h'(x), x \in S, h'(x)[a,b] \) denotes, from now on, the bits in \( h'(x)\).
from bit a to bit b. The 32 most significant bits are used to compute \( q(x) \), i.e., \( q(x) = h(x)[\text{65, 96}] > > (32 - b) \), where the symbol \( > > \) denotes the right shift of bits. The other 64 bits correspond to the values of \( y_1(x), y_2(x), \ldots, y_k(x) \), for \( k = 4 \), leading to \( \delta = 16 \). However, to save space for storing the tables used for computing \( h_0, h_1, h_2 \), we make the most significant bit of each chunk of 16 bits equal to zero during the computation of the linear hash function \( h' \). Therefore, \( \delta = 15 \). The prime number \( p \) should be chosen as large as possible, and in all cases \( p > \ell^2 \). In the implementation we set it to the largest 32-bit integer that is prime, i.e., \( p = 4294967291 \).

In the experiments we noticed that the constant \( \kappa \) presented in Eq. (5) and in Eq. (6) is in the range \( 0 < \kappa < 1 \). For instance, taking \( n = 1024 \) million keys we got \( b = 23 \) and therefore \( \kappa \approx 0.42 \). This holds for smaller values of \( n \), see Section 7. Therefore, based on these experimental results, it is possible to estimate the largest problem we can solve in 32-bit and 64-bit architectures. The largest problem we can solve in a 32-bit architecture is a key set with 500 billion keys. For larger sets more than 32 bits would be required to address a single bucket, i.e., \( b > 32 \).

But in 64-bit architecture we can deal with sets of sizes up to \( 1.8 \times 10^{21} \) keys with high probability. For larger sets \( b \) would require more than 64 bits. We remark that these estimates are based on the constant \( \kappa \approx 0.42 \) obtained experimentally and this can change for larger values of \( n \).

5. EM algorithm

The EM algorithm is a two-step external memory perfect hashing algorithm. Fig. 9 illustrates the two steps of the algorithm: the partitioning step and the construction step. The partitioning step takes a key set \( S \) of size \( n \) and uses a hash function \( q \in H_{KB} \) to partition \( S \) into \( N_b \) small buckets. The construction step generates an MPHF (or equivalently, a PHF) for each bucket \( i \), \( 0 \leq i \leq N_b - 1 \), and computes an array to store the offset of each function range, where array \( \text{offset}[i] \) stores the total number of keys before bucket \( i \) (i.e., the sum of the number of keys in buckets \( j \), \( 0 \leq j < i \)). The evaluation of the MPHF constructed by the algorithm for a key \( x \) is

\[
\text{MPHF}(x) = \text{MPHF}_i(x) + \text{offset}[i],
\]

where \( i = q(x) \) indicates the bucket where key \( x \) is and \( \text{MPHF}_i(x) \) is the position of \( x \) in bucket \( i \).

The main novelties of the EM algorithm are: (i) it uses external memory to construct PHFs or MPHFs for sets in the order of a billion keys; (ii) it constructs the resulting functions without assuming that uniform hash functions are available for free; (iii) it partitions the input into buckets small enough to fit in the CPU cache. Therefore, it accesses memory in a less random fashion when compared to the RAM algorithm.

We refine and combine a number of existing techniques in the design and implementation of the algorithm, as follows:

1. The RAM algorithm presented in Section 3 is used to compute one PHF or MPHF for each bucket.
2. The split-and-share technique [21,22] is used to split the problem into small buckets, and simulate fully random hash functions on each bucket. In Section 4 we presented a particular engineering of this idea, with a refinement that gives a class of fully random hash functions on each bucket without extra space usage.
3. External memory mergesort (see, e.g., [47,34]) is used to group the keys into the buckets, as illustrated in Fig. 9. Before the merging starts the fingerprints in each file are sorted considering the value of \( q(x) \) as the sorting key. The important insight here is that we split the problem in small buckets and this has both practical and theoretical implications. From the theoretical point of view we show that by refining the split-and-share technique to simulate fully random hash functions on the small buckets we are able to prove that the EM algorithm works for every key set with high probability. From the practical point of view, we create buckets that are small enough to fit in the CPU cache, resulting in a significant speedup (in processing time per element) compared to other methods. This is described in Section 5.1.
4. We use offset tables to put everything together to a single PHF or MPHF. This has been done in several theoretical works (see, e.g., [46,29]). In Section 5.2 we show how to implement this with low space overhead in practice.

We consider the situation in which the key set may not fit in the internal memory and so the first step of the algorithm is necessary to deal with the keys stored on disk to form the buckets. The EM algorithm first scans the list of keys and computes the hash function values (fingerprints) that will be needed later on in the algorithm. These values will (with high probability) distinguish all keys, so we can discard the original keys. From Dietzfelbinger and Weidling [21,22] we know that hash values of at least \( 2 \log n \) bits are required to have no collisions while mapping keys to fingerprints. Thus, for sets of a billion keys or more we cannot expect the list of hash values to fit in the internal memory of a standard PC.

We first use the radix sort algorithm to sort the fingerprints in each file using \( q(x) \) as sorting key. Next, we use an external memory mergesort algorithm [34] to group fingerprints with same values of \( q(x) \) in one bucket. The detailed description of the partitioning and the
function Partitioning ($S$, $\mathcal{H}_{\ell}$, $\text{Files}$)

- Let $\zeta$ be the size in bytes of the fixed-length key set $F$
- Let $M$ be the size in bytes of a priori reserved internal memory area
- Let $N_f = \lceil \zeta/M \rceil$ be the number of key blocks that will be read from disk into an internal memory area

1. Select $h'$ uniformly at random from $\mathcal{H}_{\ell}$
2. for $j = 1$ to $N_f$
   1. Read a subset $S_j$ of the keys from disk (one at a time) and store $h'(x)$, for each $x \in S_j$, into $\mathcal{B}_j$, where $|\mathcal{B}_j| = M$
3. Cluster $\mathcal{B}_j$ into $N_b$ buckets using a radix sort algorithm that takes $g(x)$ for $x \in S_j$ as sorting key
   - (i.e., the $b$ most significant bits of $h'(x)$) and if any bucket $B_i$ has more than $\ell$ keys restart partitioning step
4. Dump $\mathcal{B}_j$ to disk into Files[$j$]

Fig. 10. Partitioning step.

Fig. 11. Construction step.

construction steps are presented in Sections 5.1 and 5.2, respectively.

5.1. Partitioning step

Fig. 10 presents the partitioning step. It performs two important tasks. First, the variable-length keys are mapped to $\gamma$-bit fingerprints by using a linear hash function $h' : S \rightarrow \{0, 1\}^\gamma$ taken uniformly at random from the family $\mathcal{H}_{\ell}$ of linear hash functions presented in Section 4.1, where $\gamma = b+\ell b\ell$ bits. That is, the variable-length key set $S$ is mapped to a fixed-length key set $F$ of fingerprints. Second, the key set $S$ is partitioned into $N_f$ buckets, where $b$ is a suitable parameter chosen to guarantee that each bucket has $\ell = \Omega(\log n \log \log n)$ keys with high probability (see Eq. (6)). It outputs a set of Files containing the buckets, which are merged in the construction step when the buckets are read from disk.

In Fig. 10, the critical point is the internal sorting algorithm to make the partitioning step work in $O(n)$ time. We use radix sort for two reasons. First, it sorts each fingerprint subset $\mathcal{B}_j$ in linear time because the sorting keys are short integers (less than 15 decimal digits). Second, it just needs $O(|\mathcal{B}_j|)$ words of extra memory to control the memory usage independently of the number of keys in $S$.

5.2. Construction step

Fig. 11 presents the construction step. The for-loop that starts in line 1 builds a heap $H$ of size $N_f$, which is well-known to be linear time on $N_f$ [33]. The order relation of $H$ is given by the bucket address $i$ (i.e., the $b$ most significant bits of $x \in F$). The for-loop that starts in line 4 has the following steps. The statement in line 5 reads a bucket from disk, as described in Fig. 12. The statement in line 6 constructs a function $f_i \in \{C \cup C_d\}$ (see Definitions 3.2 and 3.3) for each bucket $B_i$ using the RAM algorithm presented in Section 3, which uses three hash functions $h_{\ell 0}$, $h_{\ell 1}$, and $h_{\ell 2}$ to compute the function $f_i \in \{C \cup C_d\}$ for each bucket $i$.

These hash functions and the hash function $q$ used in the partitioning step are described in Section 4.2. The statement in line 7 computes the next entry of the offset array. Finally, the statement in line 8 writes the description of $f_i$ and offset[i] to disk. Note that to compute offset[i+1] we need $M_i$, (i.e., the maximum value of $f_i$ on bucket $B_i$) and offset[i]. So, just two entries of the offset array are kept in memory all the time.

Fig. 12 presents the algorithm to read bucket $B_i$ from disk. Bucket $B_i$ is distributed among many files and the heap $H$ is used to drive a multi-way merge operation. The line 2 extracts and removes triple $(i,j,x)$ from $H$, where $i$ is a minimum value in $H$. The line 3 inserts $x$ in bucket $B_i$. The line 4 performs a seek operation in Files[$j$] on disk for the first read operation and reads sequentially all $\gamma$-bit fingerprints $x \in F$ that have the same index $i$ and inserts them all in bucket $B_i$. Finally, the line 5 inserts in $H$ the triple $(i',j',x')$, where $x' \in F$ is the first $\gamma$-bit fingerprint read from Files[$j$] (in line 4) that does not have the same bucket address as the previous keys.

5.3. Amount of CPU time to construct the functions

In this section we present the amount of time to construct the functions, considering the partitioning and the construction steps.

5.3.1. Partitioning step

The partitioning step presented in Fig. 10 reads the whole key set $S$ in blocks such that the computed $\gamma$-bit fingerprints fit in a memory area $B_j$, $1 \leq j \leq N_f$, of size $M$ bytes. Then, radix sort is used to cluster the $|\mathcal{B}_j|$ $\gamma$-bit fingerprints in buckets before dumping them to disk. The partitioning step runs in expected $O(n)$ time because $\sum_{j=1}^{N_f} |\mathcal{B}_j| = n$. It is expected $O(n)$ time because the algorithm might fail in line 4 whenever a bucket with more than $\ell$ keys is generated. However, it succeeds with high probability and the expected number of iterations is $O(1)$, as showed in Section 4.2.3.

5.3.2. Construction step

The construction step presented in Fig. 11 uses the RAM algorithm for each bucket. As mentioned before, the RAM algorithm is a randomized algorithm that might fail with small probability for a given bucket when it cannot find appropriate hash functions. When it fails, we restart in the partitioning step. By using the hash functions designed in Section 4.2, it is possible to make the construction step work with high probability and the number of iterations will be bounded by a constant.
The multi-way merge operation driven by a heap $H$ with $N_f$ entries using $N_f = \Omega(n^c)$ computer words can be done in linear time for $0 < \tau < 1$ (see, e.g., [1, Theorem 3.1]). For $\tau = 0.5$ the merge operation can be performed in one pass. We conclude that the construction step runs in expected $O(n)$ time, once it is essentially a multi-way merge operation and the RAM algorithm used to compute the functions $f_i$ of each bucket is also linear on the buckets’ size.

Finally, in the worst case, the $\gamma$-bit fingerprints of bucket $i$ are spread over at most $\ell$ files on disk (recall that $\ell$ is the maximum number of keys found in any bucket). Therefore, the critical step in reading a bucket is in line 4 of Fig. 12, where a seek operation in $\text{Files}[j]$ may be performed by the first read operation. The seek operation problem can be addressed by using buffering techniques [33] to amortize the number of seeks performed.

5.4. Space consumption to store the functions

The description of the resulting functions consists of the function $q$, the offset array and the functions $f_i \in \{C_i \cup C_0\}$, $0 \leq i < N_b$. The function $q$ comes from the set $H_{L_k}$ of linear hash functions over $GF(2)$ and therefore requires $O(L \log n)$ bits to be stored. The offset array has $N_b$ entries of log $n$ bits and, then, requires $O(n)$ bits since $N_b \leq n/\log n$.

If $f_i$ is a PHF, then it requires $|f_i| = \log(3)|c \cdot B_i| \cdot \log n$ bits of space, $c \geq 1.23$, to store each function $f_i$. Therefore, $\sum_{i=0}^{N_b-1} |f_i| = \log(3)|c \cdot B_i| \cdot \log n$ bits are necessary to store a PHF. If $f_i$ is an MPHF, then it requires $|f_i| = (2 + \epsilon)\log(n_{f_i}) + o(n_{f_i})$ bits of space, for $c \geq 1.23$ and $\epsilon > 0$. Therefore, $\sum_{i=0}^{N_b-1} |f_i| = (2 + \epsilon)\log(n_{f_i}) + o(n)$ bits are necessary to store an MPHF.

Additionally, we need to store the hash functions $h_{b_0}$, $h_1$, and $h_2$ (see Definition 4.3). For this we need to store 6k tables with $2^n$ entries of log $p$ bits, where $p$ is a large prime number of $O(\log n)$ bits, and the seed numbers $s_i$ of log $p$ bits, where $0 \leq i < N_b$. Considering that $s = \lceil \log(\sqrt{n}) \rceil$ and $k = 4$ are values chosen to make the EM algorithm work with high probability and $N_b \leq n/\log n$, then $h_{b_0}$, $h_1$, and $h_2$ are stored in $O(n)$ bits.

Thus, the number of bits required to store a function constructed by the EM algorithm is $\log(3)cn + O(n)$ bits for a PHF and $(2 + \epsilon)cn + O(n)$ bits, $\epsilon > 0$, for an MPHF. That means $O(n)$ bits for both cases.

5.5. Space consumption to construct the functions

The EM algorithm needs to maintain in internal memory: (i) a fixed a priori working area of size $M$ bytes that depends on the amount of internal memory available to run the algorithm; (ii) $O(|B_i|)$ words required to run the indirect radix sort algorithm; (iii) $O(N_f)$ words used to drive a $N_f$-way merge operation using the heap $H$, which allows the merge operation to be performed in one pass through each file.

Therefore, the EM algorithm requires $O(N_f)$ computer words to construct either a PHF or an MPHF. As shown in [1, Theorem 3.1], to get a linear time complexity we need $N_f = \Omega(n^c)$ computer words for $0 < \tau < 1$. To allow the merge operation to be performed in one pass we need $\tau = 0.5$.

6. HEM algorithm

We designed another version of the EM algorithm that uses faster and more compact pseudo-random hash functions proposed in [32], referred to as heuristic EM algorithm, or simply HEM algorithm from now on.

As opposed to the EM algorithm that uses the hash functions described in Section 4.2, which guarantee that the EM algorithm can be made to work for every key set, the HEM algorithm is not guaranteed that it will work for every key set. However, limited randomness often suffices in practice [2], and the HEM algorithm has worked for all key sets we have applied it to.

The HEM algorithm avoids the space needed for the lookup tables of the EM algorithm. It uses a heuristic hash function that does not impose any upper bound for the key sizes and their description requires just the storage of one 32-bit random seed for a pseudo-random number generator. Therefore, just three 32-bit random seeds are required to describe the functions $h_{b_0}$, $h_1$, and $h_2$ of each bucket. The function just loops over the key doing bitwise operations on blocks of 12 bytes and outputs a 12-byte fingerprint.

7. Experimental results

In this section we present the experimental results for the RAM algorithm with $r = 2$ and $r = 3$, the EM algorithm and the HEM algorithm. We compare them with some of the main practical perfect hashing algorithms we found in the literature. We show that the mathematical basis for minimal perfect hashing presented in Sections 3 and 5 is set to work in an implementation that can construct an MPHF for key sets of size in the billions that use much less space than previously known algorithms, including the one by Hagerup and Tholey [29]. As mentioned in Section 2.3, Belazzougui et al. [4] have shown how to generate slightly more compact functions that are 25% slower to be computed than the ones described herein. We refer the reader to their paper to check the thorough comparison that was done between their algorithm and the RAM algorithm.

The experiments were carried out on a computer with a 1.86 gigahertz Intel Core 2 processor with 1 gigabytes of main memory and a L2 cache of 4 megabytes, running Linux operating system version 2.6. The algorithms were
implemented in the C language and are available at http://cmph.sf.net under the GNU Lesser General Public License (LGPL).

We use the following metrics to compare the algorithms: (i) The amount of time to construct PHFs or MPHFs, referred to as Construction Time. (ii) The space requirement for the description of the resulting PHFs or MPHFs, referred to as Storage Space. (iii) The amount of time required by a PHF or an MPHF for each retrieval, referred to as Evaluation Time.

For the experiments we used the two collections presented in Table 1: (i) a set of 150 million randomly generated 4 byte long integers, referred to as INT4 (the choice of 4 bytes was motivated by IPv4 addresses), and (ii) a set of 1024 million 64-byte long (on average) URLs collected from the Web.

7.1. Performance of the RAM algorithm

In this section we evaluate the performance of the RAM algorithm. Table 2 presents the construction time and storage space to construct MPHFs for \( r = 2 \) and \( r = 3 \), with a confidence level of 95%. The table shows that the algorithm for \( r = 3 \) is the fastest and also constructs the most compact functions. The algorithm for \( r = 3 \) is the fastest because the probability of obtaining a hypergraph with no cycles tends to 1 for \( c = 1.23 \) (see Theorem 3.5). As expected, the construction time is influenced by the key length (INT4 are 4 bytes long and URLs are 64 bytes long on average) and the storage space is not.

Table 2

<table>
<thead>
<tr>
<th>( n )</th>
<th>Collection</th>
<th>( r = 2 )</th>
<th>( r = 3 )</th>
<th>( n = 1 )</th>
<th>( n = 12 )</th>
<th>( n = 24 )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Construction time (s)</td>
<td>Storage space</td>
<td>Bits/Key</td>
<td>Size (MB)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>INT4</td>
<td>URLs</td>
<td>INT4</td>
<td>URLs</td>
<td>INT4</td>
<td>URLs</td>
</tr>
<tr>
<td>( 1 \times 10^6 )</td>
<td></td>
<td></td>
<td>3.09 ± 0.28</td>
<td>4.00 ± 0.34</td>
<td>3.60</td>
<td>0.43</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1.32 ± 0.01</td>
<td>1.61 ± 0.01</td>
<td>2.62</td>
<td>0.31</td>
</tr>
<tr>
<td>( 12 \times 10^6 )</td>
<td></td>
<td></td>
<td>48.30 ± 4.42</td>
<td>59.04 ± 5.47</td>
<td>3.60</td>
<td>5.15</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>23.2 ± 0.02</td>
<td>26.31 ± 0.06</td>
<td>2.62</td>
<td>3.75</td>
</tr>
<tr>
<td>( 24 \times 10^6 )</td>
<td></td>
<td></td>
<td>101.59 ± 9.13</td>
<td>125.65 ± 11.35</td>
<td>3.60</td>
<td>10.30</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>51.19 ± 0.03</td>
<td>57.39 ± 0.04</td>
<td>2.62</td>
<td>7.50</td>
</tr>
</tbody>
</table>

7.2. Performance of the EM and HEM algorithms

In this section we evaluate the performance of the EM and the HEM algorithms. Fig. 13 presents the runtime of the EM and HEM algorithms for the INT4 and URL collections, for values of \( n \) equals to \( 2^m \) million keys, where \( 0 \leq j \leq 10 \). The size \( M \) of the a priori reserved internal memory area was set to 250 megabytes—later in this section we show how this parameter affects the algorithms’ runtime. The parameter \( b \) (see Eq. (5) in Section 4) was set to the minimum value that gives us a maximum bucket size lower than \( \ell = 256 \). For each value chosen for \( n \), the respective values for \( b \) are \( i \) bits for \( 13 \leq i \leq 23 \). The solid line corresponds to a linear regression model. As we were expecting, the HEM algorithm is slightly faster than the EM algorithm because it uses a faster pseudo-random hash function.

The runtime of both EM and HEM algorithms does not vary much as we explain in the following. The runtime of the algorithm that constructs the buckets is a random variable that follows a geometric distribution with mean \( 1/Pr_a \approx 1 \), because \( Pr_a \rightarrow 1 \) as \( n \rightarrow \infty \) for the RAM algorithm with \( r = 3 \). Thus we define \( X_i \), \( 0 \leq i \leq N_b \), random variables and let \( Y = \sum_{0 \leq i < N_b} X_i \) denote the runtime of the construction step. Under the hypothesis that the \( X_i \) are independent and bounded, the law of large numbers (see, e.g., [16]) implies that the random variable \( Y/N_b \) converges to the expected value of each \( X_i \) as \( n \rightarrow \infty \). This and the fact that the partitioning step was never restarted (because the parameter \( b \) is chosen so that the maximum bucket size \( \ell \) is lower than or equal to 256 with high probability) explain why the runtime does not vary much.

Table 3 presents the space required to store the functions for both EM and HEM algorithms. It shows that the space required to store the PHFs and MPHFs for the
EM algorithm is on average 2.6 and 3.21 bits per key, respectively, and for the HEM algorithm is on average 2.51 and 3.1 bits per key, respectively. Since the EM algorithm is supposed to be used for key sets that cannot be handled in internal memory where \( n \) is in the order of billions, we did not consider in the aforementioned analyses the cost to represent the lookup tables used by the hash functions of the EM algorithm described in Section 4. Those lookup tables require a storage cost of 3,345,409 bytes to implement truly random hash functions on the buckets. It is a nonsense to use the EM algorithm for small sets since the cost for the lookup tables would dominate the space required to store the functions’ description.

As mentioned in Section 6, the HEM algorithm avoids the space needed for the lookup tables of the EM algorithm. It uses a heuristic hash function that requires just the storage of one 32-bit random seed for a pseudo-random number generator. Therefore, just three 32-bit random seeds are required to describe the functions \( h_{00}, h_{10}, \) and \( h_{20} \) of each bucket. Thus, in the partitioning step, the key set \( S \) is mapped to a set \( F \) containing 12-byte long fingerprints (recall that \( \gamma = 96 \) bits). As there are no lookup tables to cause cache misses the construction time for a set of 1024 million URLs has dropped from 49.3 for the EM algorithm down to 46.2 min for the HEM algorithm in the same setup. In fact this improvement is not much and has the disadvantage of using the Jenkins hash function, for which there is no formal proof that it works much and has the disadvantage of using the Jenkins hash function, for which there is no formal proof that it works.

7.2.1. Controlling disk accesses
In this section we evaluate how much the parameter \( M \) affects the runtime of both versions of the EM algorithm. For that we fixed \( n \) in 1024 million URLs and used \( M \) equal to 100, 200, 300, 400, 500, and 600 megabytes.

In the worst case the \( \gamma \)-bit fingerprints of a bucket \( i, 0 \leq i < N_f \), are spread in at most \( \ell \) files on disk. Therefore, we need to take into account that the critical step in reading a bucket is in line 4 of Fig. 12, where a seek operation in Files[j] may be performed by the first read operation.

In order to lower the number of seek operations on disk we benefit from the fact that both versions of the EM algorithm leave almost all main memory available to be used as disk I/O buffer during the construction step. We then use a buffering technique from [33] to amortize the number of seeks.

We create a buffer \( j \) of size \( S = M / N_f \) bytes for each file \( j \), where \( 1 \leq j < N_f \). Every time a read operation is requested to file \( j \) and the data is not found in the \( j \)-th buffer, \( S \) bytes are read from file \( j \) to buffer \( j \). Hence, the number of seeks in the worst case is given by \( \zeta / S \), where \( \zeta = [\gamma / 8] = 12n \) bytes for both the EM and HEM algorithms. For that we have made the pessimistic assumption that one seek happens every time buffer \( j \) is filled in. Therefore, the number of seek is linear on \( n \) and amortized by \( S \).

Table 4 presents the number of files \( N_f \), the buffer size \( S \) used for all files, the number of seeks \( \zeta / S \) in the worst case, and the time to construct a PHF or an MPHF for 1024 million URLs as a function of the amount of internal memory available. Observing Table 4 we noticed that the time spent in the construction decreases as the value of \( M \) increases. However, for \( M > 400 \), the time variation is not as significant as for \( M \leq 400 \). This can be explained by the fact that the kernel 2.6 I/O scheduler of Linux has smart policies to avoid seeks and to diminish the average seek time (see http://www.linuxjournal.com/article/6931).

7.3. Comparison with practical results from the literature
In this section we compare the RAM, EM and HEM algorithms with the following practical algorithms from the literature: Botelho et al. [6] (referred to as BKZ), Fox et al. [25] (referred to as FCH), Majewski et al. [36] (referred to as MWHC), and Pagh [41] (referred to as PAGH). For the MWHC algorithm we used the version based on random hypergraphs with \( r = 3 \). We did not consider the one that uses random graphs with \( r = 2 \) because it is shown in [6] that the BKZ algorithm outperforms it. It is also shown therein that the BKZ algorithm outperforms the algorithm by Dietzfelbinger and Hagerup [20], which generates functions that require approximately half of the space of the ones generated by Pagh’s algorithm—the space usage is \((1+\epsilon)n \log n\) bits for \( \epsilon \in [1.13,1.15] \). The algorithm by Woelfel [48] was not considered because its implementation would look like the implementation of the EM and HEM algorithms, and therefore it is fair to say that the algorithm would be as efficient as the EM and HEM algorithms as to construction.

### Table 3
Space usage to respectively store the resulting PHFs and MPHFs of the EM algorithm and the HEM algorithm.

<table>
<thead>
<tr>
<th>( n )</th>
<th>( b )</th>
<th>EM NOT considering lookup table cost (bits/key)</th>
<th>EM considering lookup table cost (bits/key)</th>
<th>HEM (bits/key)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 10^5 )</td>
<td>9</td>
<td>2.41</td>
<td>3.00</td>
<td>270.04</td>
</tr>
<tr>
<td>( 10^6 )</td>
<td>13</td>
<td>2.67</td>
<td>3.29</td>
<td>29.43</td>
</tr>
<tr>
<td>( 10^7 )</td>
<td>16</td>
<td>2.53</td>
<td>3.13</td>
<td>5.21</td>
</tr>
<tr>
<td>( 10^8 )</td>
<td>20</td>
<td>2.74</td>
<td>3.34</td>
<td>3.00</td>
</tr>
<tr>
<td>( 10^9 )</td>
<td>23</td>
<td>2.67</td>
<td>3.29</td>
<td>2.70</td>
</tr>
</tbody>
</table>

### Table 4
Influence of the internal memory area size \( (M) \) in the runtime of both versions of the EM algorithm to construct PHFs or MPHFs for 1024 million URLs (time in min).

<table>
<thead>
<tr>
<th>( M ) (MB)</th>
<th>EM</th>
<th>HEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>( )</td>
<td>( N_f )</td>
<td>( S ) (KB)</td>
</tr>
<tr>
<td>100</td>
<td>301</td>
<td>340</td>
</tr>
<tr>
<td>200</td>
<td>119</td>
<td>1721</td>
</tr>
<tr>
<td>300</td>
<td>74</td>
<td>4151</td>
</tr>
<tr>
<td>400</td>
<td>54</td>
<td>7585</td>
</tr>
<tr>
<td>500</td>
<td>43</td>
<td>11,906</td>
</tr>
<tr>
<td>600</td>
<td>35</td>
<td>17,554</td>
</tr>
</tbody>
</table>
time. However, it requires at least $2n \log \log n$ bits to store its resulting function description. For instance, even for a small $n=1024$ keys, the function description would take at least 6.6 bits/key. Therefore the algorithm is not as practical as the EM and HEM algorithms to store space.

For all the experiments we used $n=3,541,615$ keys for the two collections presented in Table 1. The reason to choose a small value for $n$ is because the FCH algorithm has exponential time on $n$ for the construction phase, and the times explode even when the number of keys is a little larger. The following conclusions do not change if one wants to vary $n$.

Table 5 shows that the RAM (for $r=3$), EM, HEM and MWHC algorithms are faster than the others to construct MPHFs. The reason why both EM and HEM algorithms perform well is due to two main factors. First, as the key set is stored in external memory, all the other algorithms scan the whole key set every time a failure occurs, whereas both EM and HEM algorithms simply scan the whole key set once and map it to a set of fixed length fingerprints. This is intrinsically part of their design and do not introduce overhead at evaluation time. We could use the same trick for the other algorithms but this would introduce overhead to evaluate the resulting functions due to the extra level of indirection. We cannot assume that the resulting fingerprint set fits in memory. Therefore the extra level of indirection would not improve the other algorithms performance anyway. Second, as the whole key set is broken into buckets with at most 256 keys and the memory is accessed in a less random fashion the EM and HEM algorithms result in fewer cache misses.

Table 5 also shows that the RAM (for $r=3$), EM and HEM algorithms present the most compact functions. The storage space requirements in bits per key for the two versions of the RAM algorithm are 3.6 when $r=2$, and 2.62 when $r=3$. For the EM and HEM algorithms the storage space requirements are 3.21 and 3.17 bits per key, respectively. For the BKZ, MWHC and PAGH algorithms they are $\log n$, 1.23 $\log n$ and 2.03 $\log n$ bits per key, respectively. It is possible to build a more compact function with Pagh’s algorithm. For instance, we know of implementations that require 0.4 $\log n$ bits per key in practice. However, in the worst case we must force the space up to 2.03 $\log n$ bits to get the algorithm to work. The algorithm also runs slowly to create more compact functions.

Table 6 shows the evaluation time of the algorithms for a random permutation of the $n$ keys. Although the number of memory probes at retrieval time of the MPHF constructed by the PAGH algorithm is optimal [41] (it performs only 1 memory probe), it is important to note in this experiment that the evaluation time is smaller for the FCH and the RAM algorithms because the constructed functions fit entirely in the machine’s L2 cache (see the storage space size for the RAM algorithm and the FCH algorithm in Table 5). For example, for sets of size up to 13 million keys the resulting functions constructed by the RAM algorithm with $r=3$ will fit entirely in a 4-megabyte L2 cache. In a converse situation, where the functions do not fit in the cache, the MPHFs constructed by the PAGH algorithm are the most efficient.

### 7.4. Comparison of PHFs and MPHFs

In this section we compare the two types of functions constructed by the RAM (with $r=2$ and $r=3$), the EM and the HEM algorithms: PHFs ($m=n$) and MPHFs ($m=n$). Table 7 presents the following results:

- Construction time: there is no significant differences between PHFs and MPHFs constructed by any of the four algorithms. Among them, the RAM algorithm with $r=2$ is slower than the other three because the probability of obtaining an acyclic 2-graph for $c=2.09$ tends to 0.29, whereas the probability of obtaining a 3-graph for $c=1.23$ tends to one.
- Evaluation time: the PHFs for $m=[2.09n]$ and $m=[1.23n]$ are faster than MPHFs because MPHFs need to compute function rank.
- Storage space: the space for PHFs with $r=3$ and $m=[1.23n]$ is in the range 1.95–2.7 bits per key, whereas for MPHFs with $m=n$ it is in the range 2.62–3.3 bits per key.

Finally, we compare PHFs and MPHFs when they are used to index a table storing small values, say 1,2,3,4,… bits
The use of a PHF (see Table 8 presents total space values for PHFs and MPHFs. Table 7 considers construction time, evaluation time and storage space metrics using n = 3,541,615 for the two collections.)

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Type</th>
<th>m</th>
<th>Construction time (s) INT4 URLs</th>
<th>Evaluation time (s) INT4 URLs</th>
<th>Storage space INT4 or URLs</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAM (r=2)</td>
<td>PHF</td>
<td>2.00n</td>
<td>10.50 ± 1.24 14.79 ± 1.58</td>
<td>0.68 1.63</td>
<td>2.09</td>
</tr>
<tr>
<td></td>
<td>MPHF</td>
<td>n</td>
<td>11.39 ± 1.33 16.73 ± 1.89</td>
<td>1.19 2.12</td>
<td>3.60</td>
</tr>
<tr>
<td>RAM (r=3)</td>
<td>PHF</td>
<td>1.23n</td>
<td>5.54 ± 0.01 6.78 ± 0.02</td>
<td>0.79 1.71</td>
<td>1.95</td>
</tr>
<tr>
<td></td>
<td>MPHF</td>
<td>n</td>
<td>5.46 ± 0.01 6.74 ± 0.01</td>
<td>1.16 2.11</td>
<td>2.62</td>
</tr>
<tr>
<td>EM</td>
<td>PHF</td>
<td>1.23n</td>
<td>5.82 ± 0.17 7.34 ± 0.05</td>
<td>2.27 3.97</td>
<td>2.76</td>
</tr>
<tr>
<td></td>
<td>MPHF</td>
<td>n</td>
<td>5.86 ± 0.17 7.68 ± 0.22</td>
<td>2.72 4.36</td>
<td>3.31</td>
</tr>
<tr>
<td>HEM</td>
<td>PHF</td>
<td>1.23n</td>
<td>5.47 ± 0.16 5.97 ± 0.09</td>
<td>1.44 2.43</td>
<td>2.62</td>
</tr>
<tr>
<td></td>
<td>MPHF</td>
<td>n</td>
<td>5.56 ± 0.16 6.27 ± 0.11</td>
<td>1.75 2.73</td>
<td>3.17</td>
</tr>
</tbody>
</table>

Table 8 presents total space values (in bits) for PHFs and MPHFs considering the space per key to store the function (x) and the space per key to store small values (y) in a table indexed by the function. (see in Section 1 an example of a garbage collector system that uses a PHF to index a bitmap storing values of one bit). Consider x the space per key to store the function and y the space per key to store the values. The equation to compute the total space per key (in bits) is x + y + m/n. Table 8 presents total space values for PHFs and MPHFs. The use of a PHF (m = 1.23n) is always better while storing values that have less than three bits whereas the use of an MPHF (m = n) is preferable for values that have 3 bits or more. Another aspect to consider is that evaluation time for PHFs is faster than evaluation time for MPHFs because MPHFs need to compute function rank.

8. Conclusions

This paper has presented a time efficient, highly scalable and nearly optimal space perfect hashing algorithm. The basic idea to obtain scalability is the well-known idea of partitioning the input key set into small buckets. The main contribution is the way we engineer many theoretical results into an implementation that scales for billions of keys in practice. The dominating phase in the construction of the functions consists of external sorting n fingerprints of O(log n) bits in O(n) time. The construction algorithm is highly scalable because it uses only a little amount of internal memory to work, basically the space necessary to accommodate a heap that drives a multi-way merge operation, which is $O(n^2)$ computer words to have linear time complexity, where $0 < \tau < 1$. In our case, as we want to perform the merge operation in one pass, we need $\tau = 0.5$ (see, e.g., [1, Theorem 3.1]). As discussed in Section 4.2.4, in a 64-bit architecture our algorithm is able to deal with key sets of size $n = 1.8 \times 10^9$.

The resulting functions are evaluated in $O(1)$ time and take a constant number of bits per key of storage space. The space usage depends on the relation between the size $m$ of the hash table and the size $n$ of the input. For $m = n$, the space usage is in the range $2.62n + o(n)$ to $3.3n + o(n)$ bits, depending on the constants involved in the construction and evaluation phases. For $m = \left\lceil 1.23n \right\rceil$ the space usage is in the range $1.95n + o(n)$ to $2.7n + o(n)$ bits. In all cases, this is within a small constant factor from the information theoretical minimum of approximately $1.44n$ bits for MPHFs and $0.89n$ bits for PHFs.

The algorithm is theoretically well understood. We have illustrated the scalability of our algorithm by constructing an MPHF for a set of 1024 million URLs from the World Wide Web of average length 64 characters in approximately 46 min, using a commodity PC.

Finally, the algorithm is suitable for a distributed and parallel implementation. For instance, in [5] was presented a distributed and parallel version of the EM algorithm. In the distributed algorithm, the keys to be processed are distributed among several machines. Further, both the buckets and the construction of the hash functions for each bucket are also distributed among the participating machines. Two versions of the distributed algorithm were presented: one where both the description and the evaluation of the resulting MPHFs are centralized in one machine, and another version where both the description and the evaluation of the resulting MPHFs are distributed among the participating machines. In the centralized evaluation algorithm, the task of writing the final MPHFs to disk corresponds to the sequential part and represents approximately 0.5% of the execution time. In the distributed evaluation algorithm, the MPHFs are written in parallel in each participating machine. Therefore, in this case, the fraction of parallelism that can be potentially exploited corresponds to 100% of the execution time. That is why both versions of the parallel
algorithm are considered embarrassingly parallel. Considering the construction phase of both algorithms, an MPHF for a set of 14.336 billion 16-byte integer keys can be constructed in 50 minutes using 14 commodity PCs, achieving an almost linear speedup. Considering the MPHFs fed by a key stream of one billion 16-byte integers taken at random, the time spent by both sequential and centralized algorithms was 24.54 min whereas the time taken at random, the time spent by both sequential and centralized algorithms was 11.47 min, an improvement of approximately 214%.

Acknowledgments

We thank Djamal Belazzougui for suggesting a method to construct PHFs that map to the range \(\{0, \ldots, m-1\}\) based on random 3-graphs. The resulting functions are stored in 2.46 bits per key and this space usage was further improved to 1.95 bits per key by using arithmetic coding. We also thank the anonymous referees of prior submissions and the partial support given by the Brazilian National Institute of Science and Technology for the Web (Grant MCT/CNPq 573871/2008-6), Project InfoWeb (Grant MCT/CNPq/CT-INFO 550874/2007-0) and CNPq Grant (Nivio Ziviani).

Appendix A. Symbol table

Symbols and acronyms used throughout the paper are given in Table A1.

Appendix B. Probability distribution of cycles in bipartite random graphs

In this section we show that the probability distribution of cycles in bipartite random graphs can be approximated by a Poisson distribution. For that we are going to follow the same reasoning used by Janson [31], which has applied the technique on Poisson convergence and Poisson processes to random graphs without the bipartite assumption.

Table A1

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\beta)</td>
<td>Number of bits used to encode each entry of (g)</td>
</tr>
<tr>
<td>(b)</td>
<td>Parameter chosen to guarantee that each bucket has at most (\ell = \Omega(\log n \log \log n)) keys</td>
</tr>
<tr>
<td>(B_i)</td>
<td>Set of fingerprints in bucket (i)</td>
</tr>
<tr>
<td>(\tilde{B})</td>
<td>Set of buckets induced by a function (h_0 : \mathcal{S} \rightarrow {0, 1}^b)</td>
</tr>
<tr>
<td>(c)</td>
<td>Ratio between number of edges and number of vertices in an acyclic hypergraph (G), such that (c &gt; c(\ell))</td>
</tr>
<tr>
<td>(c(\ell))</td>
<td>Minimum ratio between number of edges and number of vertices in a hypergraph (G), so it is acyclic with high probability</td>
</tr>
<tr>
<td>(C)</td>
<td>Class of perfect hash functions</td>
</tr>
<tr>
<td>(C_p)</td>
<td>Class of minimal perfect hash functions</td>
</tr>
<tr>
<td>CMPH</td>
<td>C Minimal Perfect Hashing Library (<a href="http://cmph.sf.net">http://cmph.sf.net</a>)</td>
</tr>
<tr>
<td>(\epsilon)</td>
<td>Real constant. Its value is restricted to be either (\epsilon &gt; 0) or (0 &lt; \epsilon &lt; 1) depending on the context</td>
</tr>
<tr>
<td>(e)</td>
<td>Edge or hyperedge of a hypergraph</td>
</tr>
<tr>
<td>(E)</td>
<td>Set of edges of a hypergraph</td>
</tr>
<tr>
<td>EM</td>
<td>External memory algorithm</td>
</tr>
<tr>
<td>(F)</td>
<td>Set of fixed-length (\gamma)-bit fingerprints</td>
</tr>
<tr>
<td>(g)</td>
<td>Array containing the values (g(v), v \in V)</td>
</tr>
<tr>
<td>(\gamma)</td>
<td>Fingerprint length in bits, which is obtained from a linear hash function (h : \mathcal{S} \rightarrow {0, 1}^\gamma)</td>
</tr>
<tr>
<td>(G_{(V,E)})</td>
<td>Hypergraph with a vertex set (V) and an edge set (E), each edge connecting (r) vertices</td>
</tr>
<tr>
<td>(h)</td>
<td>Hash function</td>
</tr>
<tr>
<td>(\mathcal{H})</td>
<td>Set of hash functions</td>
</tr>
<tr>
<td>(\mathcal{H}_{(r,b)})</td>
<td>Set of linear hash functions mapping from ({0, 1}^d) to ({0, 1}^b)</td>
</tr>
<tr>
<td>(\ell)</td>
<td>Maximum number of keys in any bucket</td>
</tr>
<tr>
<td>(\mathcal{L})</td>
<td>List of edges of a hypergraph</td>
</tr>
<tr>
<td>(L)</td>
<td>Maximum key length in bits</td>
</tr>
<tr>
<td>(m)</td>
<td>Size of a hash function range</td>
</tr>
<tr>
<td>(\mu)</td>
<td>Minimal perfect hash function</td>
</tr>
<tr>
<td>MPHF</td>
<td>Minimal perfect hash function</td>
</tr>
<tr>
<td>(n)</td>
<td>Number of keys in (S)</td>
</tr>
<tr>
<td>(N_b)</td>
<td>Number of buckets</td>
</tr>
<tr>
<td>(\text{offset}[i])</td>
<td>Total number of keys before bucket([i])</td>
</tr>
<tr>
<td>PHF</td>
<td>Perfect hash function</td>
</tr>
<tr>
<td>(P_{\text{sa}})</td>
<td>Probability that a hypergraph is a forest</td>
</tr>
<tr>
<td>(r)</td>
<td>Number of vertices of a hyperedge</td>
</tr>
<tr>
<td>(\text{rankTable})</td>
<td>Table storing the rank of every k-th index in (g), where (k = \lfloor \log(m)/c \rfloor), using (cm) additional bits of space, for (0 &lt; \epsilon &lt; 1)</td>
</tr>
<tr>
<td>(\text{RAM})</td>
<td>Random access memory algorithm</td>
</tr>
<tr>
<td>(\text{rank})</td>
<td>Function returning the number of (g)-values assigned before a given vertex (v \in V) in (g)</td>
</tr>
<tr>
<td>(S)</td>
<td>Subset of a key universe of size (</td>
</tr>
<tr>
<td>(T_i)</td>
<td>Lookup table where each entry gives the number of assigned vertices (i.e., (g[i] \neq r), for (0 \leq i \leq m - 1)) in a byte from (g)</td>
</tr>
<tr>
<td>(u)</td>
<td>Size of a key universe</td>
</tr>
<tr>
<td>(U)</td>
<td>Key universe</td>
</tr>
<tr>
<td>(V)</td>
<td>Set of vertices of a hypergraph</td>
</tr>
</tbody>
</table>
A point process takes place in some set \( \mathcal{Y} \) that is assumed to be a locally compact second countable Hausdorff topological space (e.g., \( \mathcal{Y} \) may be a closed or open subset of \( \mathbb{R}^d \)). A Radon measure on \( \mathcal{Y} \) is a Borel measure \( \mu \) such that \( \mu(K) < \infty \) for every compact set \( K \subset \mathcal{Y} \). Point processes are defined as random integer valued Radon measures that can be written as

\[
\xi = \sum_{i=1}^{N} \delta_{X_i},
\]

(B.1)

where \( X_i \) are random variables with values in \( \mathcal{Y} \), \( N \) is a finite or infinite random variable and \( \delta_x \) is the Dirac measure \( \delta_x(A) = 1(x \in A), A \subset \mathcal{Y} \).

One can think of \( \xi \) as the random multiset \( \{ X_i \} \) such that \( \xi(A) = \sum_{i} 1_{X_i \in A} \) is the number of points of this multiset that fall in \( A \).

Let \( \lambda \) be a Radon measure in \( \mathcal{Y} \). The Poisson process with intensity \( \lambda \) is the unique point process \( \xi \) such that the random variable \( \xi(A) \) is Poisson distributed with parameter \( \lambda(A) \) for every Borel set \( A \subset \mathcal{Y} \), and \( \xi(A_1,\ldots, A_k) \) are independent for any disjoint Borel sets \( A_1,\ldots,A_k \). A simple example is when \( \mathcal{Y} \) is a finite or infinite discrete set and a Poisson process on \( \mathcal{Y} \) is a collection of independent Poisson variables. If \( \xi \) is a point process, \( A \) and \( B \) are two Borel sets in \( \mathcal{Y} \), then \( A \xi \) will denote the restriction of \( A \xi \) to \( A \) defined by \( A \xi(B) = \xi(A \cap B) \) for \( A \xi(B) \).

A \( \lambda \)-continuity set is a Borel set \( A \) such that \( \lambda A = 0 \). Likewise, if \( \xi \) is a point process, \( A \) is a \( \xi \)-continuity set if \( \xi(A) = 0 \) almost surely. If \( \xi \) is a Poisson process with intensity \( \lambda \), the \( \xi \)-continuity sets are exactly the \( \lambda \)-continuity sets. Note that the \( \xi \)-continuity sets form a ring.

A DC-semiring \( \mathcal{Q} \) is a semiring of Borel sets such that for any \( \epsilon > 0 \), any compact subset of \( \mathcal{Y} \) may be covered by a finite number of elements of \( \mathcal{Q} \) having diameter less than \( \epsilon \). A DC-ring is a DC-semiring that is a ring. The family of finite disjoint unions of sets in a given DC-semiring is a DC-ring.

Consider a sequence \( \xi_1,\ldots,\xi_m \) of point processes on \( \mathcal{Y} \) where each one is represented by Eq. (B.1). A representation with a non-random (finite or infinite, and possibly depending on \( m \)) number of terms is preferable to operate with rather than one that has random number of terms. Fortunately it is possible to turn the random number of terms into a non-random number of terms by the following device. Let \( \mathcal{Y}' \) be a space that contains \( \mathcal{Y} \) as a subspace and consider that the random variables \( X_i \) from Eq. (B.1) have values in \( \mathcal{Y}' \), but \( \delta_{X_i} \) are measures on \( \mathcal{Y} \). Thus \( \delta_{X_i} = 0 \) if \( X_i \in (\mathcal{Y}' - \mathcal{Y}) \), which means that any number of “ghosts” \( X_i \) with values in \( \mathcal{Y}' - \mathcal{Y} \) may be added. Therefore the total number of terms can be fixed as infinite. Note that the actual values taken by \( X_i \) outside \( \mathcal{Y} \) are irrelevant, because all points in \( \mathcal{Y}' - \mathcal{Y} \) are treated as non-existent.

Convergence of point processes is discussed in two topologies: (i) the vague topology defined on the set of all Radon measures; and (ii) the weak topology defined on the subset of finite measures. It is used \( \xrightarrow{\text{vague}} \) and \( \xrightarrow{\text{weak}} \) to denote convergence in distribution in these topologies, respectively, as \( m \to \infty \) (the phrase “as \( m \to \infty \)” is usually omitted from the formulae). It is also used \( \xrightarrow{\text{weak}} \) to denote convergence in distribution regardless of the topology. In the following it is presented the results we have used to prove the claim in Section Appendix B.2, which were proved by Janson [31].

**Lemma B.1** (Janson [31], Lemma 2.6). Let \( \xi \) be a DC-semiring on \( \mathcal{Y} \). Let \( \xi' \) be a point process on \( \mathcal{Y} \) and \( \mu \) a Borel measure such that \( \xi' \mu(B) \leq \mu(B) \) for every \( B \in \mathcal{Q} \). Then \( \xi' \mu \leq \mu \).

**Theorem B.2** (Janson [31], Theorem 3.2). Let \( \lambda \) be a Radon measure on \( \mathcal{Y} \). Let, for each \( m \), \( \xi_m \) be a point process \( \sum_{j=1}^{J} \delta_{X_j} \) on \( \mathcal{Y} \), where \( \{ X_j \}_{j=1}^{J} \) is a family of random variables with values in \( \mathcal{Y}' - \mathcal{Y} \) and \( X_j \) are independent of \( \lambda \). Assume that, for each \( m \), for every \( j \) there exists a subset \( D_j \) of \( \mathcal{Y} \) with \( j \in \mathcal{D}_j \) such that \( X_j \) is independent of \( \{ X_k : k \notin D_j \} \). Assume further that, for every \( Q \) and \( \xi' \) in a fixed DC-semiring \( \mathcal{Q} \) on \( \mathcal{Y} \) of \( \xi' \)-continuity sets, as \( m \to \infty \)

\[
\sum_{j=1}^{J} \Pr(X_j \in Q) \to \lambda(Q),
\]

(B.2)

\[
\sum_{j=1}^{J} \Pr(X_j \in Q) \Pr(X_k \in \xi') \to 0,
\]

(B.3)

\[
\sum_{j=1}^{J} \Pr(X_j \in Q \text{ and } X_k \in \xi') \to 0.
\]

(B.4)

Then \( \xi_m \xrightarrow{\text{vague}} \xi' \), where \( \xi' \) is a Poisson process on \( \mathcal{Y} \) with intensity \( \lambda \).

**Corollary B.3** (Janson [31], Corollary 3.2). Assume that the conditions of Theorem B.2 are satisfied and, furthermore, that \( \mu \) is a Borel measure such that, for every \( Q \in \mathcal{Q} \) and every \( m \),

\[
\sum_{j=1}^{J} \Pr(X_j \in Q) \leq \mu(Q).
\]

(B.5)

Then, for every \( \xi' \)-continuity set \( A \subset \mathcal{Y} \) with \( \mu(A) < \infty \),

\[
A \xrightarrow{\text{vague}} \xi'(A) \quad \text{with} \quad \xi'(A) \xrightarrow{\text{weak}} \text{Poisson}(\lambda(A)).
\]

**B.2. Probability distribution of cycles in bipartite random graphs converges to a Poisson distribution**

In this section we apply the general results presented in Section Appendix B.1 to show that the probability distribution of cycles in bipartite random graphs converges to a Poisson distribution. From now on, every time we use the word “graph” we mean “bipartite random graph”.

A bipartite random graph \( G_{n,m}(V,E) \), where \( |V| = 2n = m \), \( |E| = dm \), and \( d = n/\eta \) is the average degree of \( G_{n,m} \) obtained by a stochastic process where each graph starts with a set of \( m = 2\eta \) vertices and at each step one edge is added between two vertices (one from
each partition) at random. Different random graph models produce different probability distributions on graphs. Let \( G_{np} \), \( 0 \leq p \leq 1 \), be the model of all bipartite random graphs with \( m = 2n \) vertices and the \( n^2 \) possible edges occur independently of each other, each with probability \( p \). Other closely related model is the \( G_{n, n} \) model which assigns equal probability to all bipartite graphs with exactly \( m = 2n \) vertices and \( n \) edges. It is well known in the random graph theory that results for \( G_{np} \) are equivalent to results for \( G_{n, n} \) whenever \( p = d/n \) and \( n \to \infty \) (this is equivalent to \( m \to \infty \) and they can be interchangeable), because the expected number of edges for the graphs in \( G_{np} \) would be \( np = n \). The \( n \) edges are almost surely distinct because there will be no multiple edges with probability \( (\eta^2)_{n}/\eta^{2n} \), where \( (\eta)_{n} = \eta(\eta-1) \cdots (\eta-n+1) \). In the limit, when \( n \to \infty \), this probability tends to \( e^{-d/2} \). To get this we used standard calculus to approximate \( f(x) = 1-x \) by \( g(x) = e^{-x} \) for a small real \( x \in (0, 1) \).

Consider the evolution of bipartite random graphs when the edges are sequentially added at random. Let \( \{T_e : e \text{ ranges over the set of edges in the complete bipartite graph} \}
\]

\( K_{n, n} \)

be \( n^2 \) random variables with a common continuous distribution on \([0, \infty)\). As in Janson [31] we let each \( T_e \) be uniformly distributed on \([0, n^2]\). Let \( G_{n, n}(t) \) denote this process to generate bipartite random graphs with \( m = 2n \) vertices and all edges \( e \) for which \( T_e \leq t \). Thus one can think of \( T_e \) as the time the edge \( e \) appears. \( G_{n, n}(t) \) generates a random graph in \( G_{np} \) with \( p = \text{Pr}(T_e \leq t) \). The process \( G_{n, n}(t) \) itself graphs in \( G_{np} \) for different values of \( p \). Furthermore, as \( t \) increases, new edges are added at the random times \( (T_{1/2}^i)^n_i \) and they are almost surely distinct as we have seen above. Hence graphs in \( G_{np} \) can be constructed as \( G_{np}(T_n) \). Therefore results for both \( G_{np} \) and \( G_{n, n} \) can be obtained from results for the process \( G_{n, n}(t) \). Let \( T_C = \max \{T_e : e \text{ belongs to the edge set } C \} \) be the time at which an arbitrary subgraph \( G \) arises during the process \( G_{n, n}(t) \). Thus, if \( G \) has \( |G| \) edges the following holds:

\[
\text{Pr}(T_C \leq t) = \left( \frac{1}{n^2} \right)^{|G|}, \quad 0 \leq t \leq n^2.
\] (B.5)

Let, for each \( m, J = [\sum_{l=0}^{\infty} J_{2l}] \), where \( J_{2l} \) is the set of cycles of even length \( 2l \) in the complete bipartite graph \( K_{n, n} \). A cycle in \( J_{2l} \) can be represented as a sequence of \( 2l \) distinct vertices in \( K_{n, n} \). As each cycle can be represented in \( 2l \) ways by changing the start point, the cardinality of \( J_{2l} \) is

\[
|J_{2l}| = \frac{1}{2l} (\eta^2)^l.
\] (B.6)

Let \( Y = [0, \infty) \times \{4, 6, 8, \ldots\} \). Thus \( Y \) is the disjoint union of infinitely many half-lines \( Y_{2l} \), \( l \geq 2 \). For any cycle \( J \in J \), define

\[
X_J = (T_J, 2l)
\] (B.7)

where \( T_J \) is the time when cycle \( J \) arises in a process \( G_{n, n}(t) \). Let \( \xi^i = \sum_{J \in J} \xi^i_J \), where \( \xi^i_m([0, t] \times \{i\}) \). \( \xi^i_m([0, t] \times \{i\}) \) are the number of cycles of even lengths 4, 6, \ldots, in a bipartite random graph obtained in the process \( G_{n, n}(t) \), or equivalently, in \( G_{np} \) with \( p = t/\eta, (t \leq \eta \). The space \( Y \) allows us to consider cycles of all even lengths simultaneously. It is evident from the definitions that, if \( J \in J_{2l} \)

\[
\text{Pr}(X_J \in J, 2l) = (t/\eta)^{2l},
\] (B.8)

and thus, as \( \eta \to \infty \)

\[
\sum_{J \in J} \text{Pr}(X_J \in [0, t] \times \{2l\}) = |J_{2l}| \times \left( \frac{t}{\eta} \right)^{2l} = \frac{1}{2l} t^{2l}.
\] (B.9)

We now define a Randon measure \( \lambda \) on \( Y \) as \( \lambda([0, t] \times \{2l\}) = F_{2l}(t), t \geq 0, l \geq 2 \), where

\[
F_{2l}(t) = \frac{1}{2l} t^{2l}.
\] (B.10)

Therefore \( \lambda \) equals \( F_{2l}(t) \) \( dt \) on \( Y_{2l} \), where

\[
F_{2l}(t) = \frac{d}{dt} F_{2l}(t) = t^{2l-1}.
\] (B.11)

\[
\text{Pr}(X_J \in Q) = \sum_{K \in (D_{2l-2}) \mid J} \text{Pr}(T_J \leq t \quad \text{and} \quad T_K \leq t)
\]

\[
= \sum_{i=1}^{2l-1} \sum_{K \in D_{2l-2}} \text{Pr}(T_J \leq t \quad \text{and} \quad T_K \leq t)
\]

\[
= \sum_{i=1}^{2l-1} |D_{2l-2} \times \{i\}| \left( \frac{t}{\eta} \right)^{2l-1}
\]

\[
\leq \sum_{i=1}^{2l-1} \left( \frac{2l}{i+1} \right) \left( \frac{2l}{2l-i-1} \right)^{2l-1} \left( \frac{t}{\eta} \right)^{2l-1}
\]

\[
= O(\eta^{-2l-1}) \quad \text{as} \quad \eta \to \infty.
\] (B.12)
for every $J \in \mathcal{J}_{2l}$. Therefore Eq. (B.4) holds by combining Eqs. (B.6) and (B.12) as follows:

$$\sum_{j=2}^{\infty} \mathbb{O}(\eta^{2l-1}) \left( \frac{1}{2l!} \right)^2 \rightarrow 0 \quad \text{as} \quad \eta \rightarrow \infty. \quad (B.13)$$

This finishes the proof for the following theorem.

**Theorem B.4.** Let $\tilde{z}$ be a Poisson process on $\mathcal{Y}$ with intensity $\lambda$.

Note that $\tilde{z}$ can be thought of as a collection of independent Poisson processes on $[0,\infty)$ with the intensities $f(t)f(t)\ldots$ given by Eq. (B.11). Furthermore, by Eq. (B.6) (cf. Eq. (B.9))

$$\sum_{j=2}^{\infty} \mathbb{P}(X_j \in [a,b] \times [2l]) = |\mathcal{J}_{2l}|$$

$$\times \left( \left( \frac{b}{\eta} \right)^{2l} - \left( \frac{a}{\eta} \right)^{2l} \right) \leq \lambda([a,b] \times [2l]). \quad (B.14)$$

By Lemma B.1, with $\mu = \lambda$, $E_{z_m} \subseteq \lambda$ for every $m$, and Corollary B.3 yields the following extension of Theorem B.4.

**Theorem B.5.** If $A$ is a $\mathcal{J}_{2l}$-continuity set in $\mathcal{Y}$ with $\lambda(A) < \infty$, then $\bar{A}_{z_m} \rightarrow \lambda(A)$, in particular

$$\bar{z}_{z_m}(A) \xrightarrow{d} \text{Poisson}(\lambda(A)).$$

Let $\mathcal{C}_{2l}(G)$ be a random variable that measures the number of cycles of length $2l$ in a graph $G$ generated through the process $\mathcal{G}_{n,t}(t)$, where $0 \leq t < \infty$ and the average degree of graph $G$ be $d = \text{tr}$. Theorem B.4 (or Theorem B.5) immediately yields

$$\mathcal{C}_{2l}(G) = \bar{z}_{z_m}([0,d] \times [2l]) \xrightarrow{d} \text{Poisson} \left( \frac{1}{2l} d^2 \right). \quad (B.15)$$

More generally, we obtain the following corollary.

**Corollary B.6.** Let $\mathcal{G}_{n,t}$ be a random graph in $\mathcal{G}_{n,t}$, and $\mathcal{G}_{n,n}$ be a random graph in $\mathcal{G}_{n,n}$. Let $0 \leq d < \infty$ and $\eta \rightarrow d$, then

$$\mathcal{C}_{2l}(\mathcal{G}_{n,n}) \xrightarrow{d} \text{Poisson} \left( \frac{1}{2l} d^2 \right). \quad (B.16)$$

If $\eta \rightarrow \infty$ and $n/\eta \rightarrow d$, then

$$\mathcal{C}_{2l}(\mathcal{G}_{n,n}) \xrightarrow{d} \text{Poisson} \left( \frac{1}{2l} d^2 \right). \quad (B.17)$$

**Proof.** Observe that $\eta \rightarrow d$ and $\bar{z}_{z_m} \xrightarrow{d} \tilde{z}$ implies $\bar{z}_{z_m}([0,\eta d] \times [2l]) \xrightarrow{d} \bar{z}_{z_m}([0,d] \times [2l])$, which yields Eq. (B.16). Eq. (B.17) follows similarly because $\mathcal{C}_{2l}(\mathcal{G}_{n,n}) = \bar{z}_{z_m}([0,T] \times [2l])$, and $T \rightarrow d$. □

Since

$$\sum_{i=2}^{\infty} F_i(2l) = \sum_{i=2}^{\infty} \frac{1}{2l} d^2 = -\frac{1}{2} \ln(1-d^2) - \frac{1}{2} d^2 \quad \text{for} \quad 0 \leq d < 1. \quad (B.18)$$

We have used Maclaurin’s expansion $\sum_{j=0}^{\infty} \frac{x^j}{j!} = -\ln(1-x)$ above, where $x = d^2$. **Theorem B.5** yields this section’s claim.

**Corollary B.7.** Let $\mathcal{G}_{n,n,p}$ be a random graph in $\mathcal{G}_{n,n,p}$ and $\mathcal{C}_{n}(\mathcal{G}_{n,n})$ be a random variable that measures the number of cycles of any even length larger than or equal to $4$ in $\mathcal{G}_{n,n,p}$. If $0 \leq d < 1, np \rightarrow d$, and $\eta \rightarrow \infty$ then

$$\mathcal{C}_{n}(\mathcal{G}_{n,n,p}) \xrightarrow{d} \text{Poisson} \left( -\frac{1}{2} \ln(1-d^2) - \frac{1}{2} d^2 \right). \quad (B.19)$$

**References**
