Algorithms and data structures for hyperedge queries

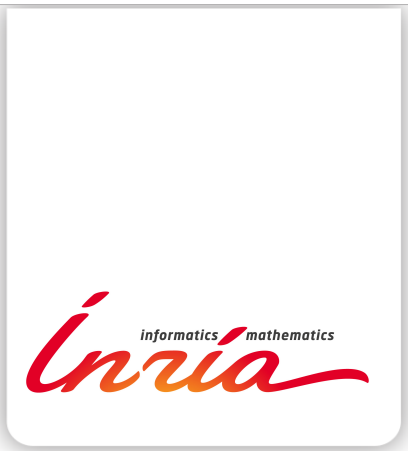
Jules Bertrand, Fanny Dufossé, Bora Uçar

Algorithms and data structures for hyperedge queries

Jules Bertrand, Fanny Dufossé, Bora Uçar

RESEARCH REPORT
N° 9390
February 2021
Project-Teams ROMA, DataMove

ISSN 0249-6399 ISRN INRIA/RR--9390--FR+ENG



Algorithms and data structures for hyperedge queries

Jules Bertrand*, Fanny Dufossé†, Bora Uçar‡

Project-Teams ROMA, DataMove

Research Report n° 9390 — version 2 — initial version February 2021 — revised version May 2021 — 25 pages

Abstract: We consider the problem of querying the existence of hyperedges in hypergraphs. More formally, we are given a hypergraph, and we need to answer queries of the form "does the following set of vertices form a hyperedge in the given hypergraph?". Our aim is to set up data structures based on hashing to answer these queries as fast as possible. We propose an adaptation of a well-known perfect hashing approach for the problem at hand. We analyze the space and run time complexity of the proposed approach, and experimentally compare it with the state of the art hashing-based solutions. Experiments demonstrate that the proposed approach has the shortest query response time than the other considered alternatives, while having a larger construction time than some of the alternatives.

Key-words: Hashing, perfect hashing, hypergraphs

* ENS Lyon, 46, allée d'Italie, ENS Lyon, Lyon F-69364, France.
† Inria Grenoble, Rhône Alpes, 38330, Montbonnot-Saint-Martin, France.
‡ CNRS and LIP (UMR5668 Université de Lyon - CNRS - ENS Lyon - Inria - UCBL 1), 46 allée d'Italie, ENS Lyon, Lyon F-69364, France.

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Algorithmes et structures de données pour le problème de requêtes d'existences d'hyperarêtes dans des hypergraphe

Résumé : Nous considérons le problème de requêtes d'existences d'hyperarêtes dans des hypergraphes. Plus formellement, pour un hypergraphe donné, nous devons répondre à des requêtes de la forme "est-ce que l'ensemble de sommets suivant forme une hyperarête de l'hypergraphe?". Notre objectif est de mettre en place une structure de donnée basée sur du hachage pour répondre à ces requêtes le plus rapidement possible. Nous proposons une adaptation d'une approche bien connue de hachage parfait pour notre problème. Nous analysons la complexité en temps et en espace de cette approche, et nous la comparons expérimentalement à des solutions de l'état de l'art basées sur le hachage. Les expériences démontrent que cette approche a un temps de réponse aux requêtes plus court que les alternatives considérées, pour un temps de construction plus long que certaines des alternatives.

Mots-clés : hachage, hachage parfait, hypergraphes

1 Introduction

Let $H = (V, E)$ be a hypergraph, where V is the set of vertices, and E is the set of hyperedges. We are given a set of queries of the form "is $h \subseteq V$ a member of E?" one by one, and we want to answer these queries. We are interested in data structures and algorithms enabling constant time per query in the worst-case with small memory requirements and construction/preprocess time. We focus on d-uniform, d-partite hypergraphs, where the vertex set is a union of d disjoint parts $V = \bigcup_{i=0}^{d-1} V^{(i)}$, and each hyperedge has exactly one vertex from each part $V^{(i)}$.

We are motivated by a tensor decomposition method recently proposed by Kolda and Hong [11]. This is a stochastic, iterative method targeting efficient decomposition of both dense and sparse tensors, or multidimensional arrays. Our focus is on the sparse case. For this case, Kolda and Hong propose a sampling approach, called stratified sampling, in which the nonzeros and the zeros of a sparse tensor are sampled separately at each iteration for accelerating the convergence of the decomposition method. The stratified sampling approach works as follows. The nonzeros of the input tensor are sampled uniformly at random. For sampling zeros in a d-dimensional tensor, a d-tuple of indices is generated randomly and tested to see if the input tensor contains a nonzero at that position. If the position is nonzero, the d-tuple is rejected and a new one is generated, until a desired number of indices corresponding to zeros of the input tensor are sampled. Sampling nonzeros is a straightforward task, as the nonzeros of a tensor are available, usually, in an array. To sample from the zeros of a tensor, Kolda and Hong propose a method based on sorting the nonzeros as a preprocess and then using binary search during query time to see if the tuple exists or not. They report that this approach is more efficient than other alternatives based on hashing in their tests—which are carried out in Matlab. Since the above implementation of sampling for zeros can be time consuming, Kolda and Hong propose and investigate other sampling approaches for their stochastic tensor decomposition method. Among the alternatives, the stratified sampling approach is demonstrated to be more useful numerically. That is why we are motivated to increase efficiency of the stratified sampling approach by developing data structures and algorithms for quickly detecting whether a given position in a tensor is zero or not.

Let $\mathcal{T}$ be a d-dimensional tensor of size $s_0 \times \cdots \times s_{d-1}$, where s_i is the size of the corresponding dimension. An entry in the tensor is indexed by a d-tuple, e.g., $\mathcal{T}[i_0, \ldots, i_{d-1}]$. One can associate a d-uniform, d-partite hypergraph $H = (V, E)$ with a tensor $\mathcal{T}$ as follows. In H, the vertex set is $V = \bigcup_{i=0}^{d-1} V^{(i)}$ where $V^{(i)} = \{v_0^{(i)}, \ldots, v_{s_i-1}^{(i)}\}$. Furthermore, there is a hyperedge $h \in E$ of the form $h = [v_{i_0}^{(0)}, \ldots, v_{i_{d-1}}^{(d-1)}]$ for each nonzero $\mathcal{T}[i_0, \ldots, i_{d-1}]$. From this correspondence, we see that the problem of testing if a given position in a tensor is zero can be cast as the problem of testing the existence of hyperedges in the associated d-uniform, d-partite hypergraph.

We design and implement a perfect hashing based approach by building on the celebrated method by Fredman, Komlós, and Szemerédi [8] (FKS method). The FKS method stores a given set S of cardinality n with $O(n)$ space in such a way that it takes constant time to answer a membership query in the worst-case. The FKS approach thus promises an asymptotically optimal solution to our problem of answering hyperedge queries. After reviewing the original FKS method in Section 2, we discuss the necessary changes to adapt it to answer hyperedge queries in Section 3. We first list some theoretical properties of the proposed method that are inherited from FKS in Section 3.1, for which the proofs are given in Appendix. We then present an approach to improve space utilization in Section 3.2; while our approach can also be used in the original FKS method, its effects are much more tangible in our use case. We note that since each element in our case is of size d, a look-up takes $O(d)$ time—which is obviously not constant if d is part of the input. Since the queries are of size d, a query response time of $O(d)$ is optimal.

We restrict our attention to d-uniform, d-partite hypergraphs both in describing the proposed

method and experimenting with it. This is so, as it covers the tensor decomposition application. Furthermore, as we discuss in Section 3.3, this is without loss of generality—the method is applicable to general hypergraphs.

To the best of our knowledge, the hyperedge query problem is first addressed by Kolda and Hong. The underlying problem is that of static hashing and hence existing perfect hashing methods can be used. Minimal perfect hash functions (MPHFs) are static data structures that map a given set with n elements to $\{0, \ldots, n-1\}$. By using MPHFs, one can store the ids of hyperedges in a space of size n and answer queries in constant time in the worst case. There exists a number of publicly available MPHF implementations [7, 9, 12, 13, 15]. While these are highly efficient with practically efficient implementations, the hyperedge query problem should be addressed on its own. This is so, as the d dimensional structure of the hyperedges in our target application can enable special hashing methods, and converting the tensor's data into input for hashing methods on other structures can affect the run time. We compare our approach experimentally (Section 4) with a number of methods that use the current state-of-the-art minimal perfect hashing methods, and report faster query response times than all methods considered, while being slower than half of them. We note that there are recent work which use random hypergraph models to build variants of Cuckoo hashing methods [1, 5, 9, 14, 18, 19]. Our work is not in the same domain: we seek static hashing methods for querying the existence of hyperedges in a given hypergraph.

2 Preliminaries and background

For an event $\mathcal{E}$, we use $\Pr(\mathcal{E})$ to denote the probability that $\mathcal{E}$ holds. For a random variable X, we use $\mathbf{E}(X)$ to denote the expectation of X. Markov's inequality states that for a random variable X that assumes only nonnegative values with expectation $\mathbf{E}(X)$, the probability that $X \geq c$ for a positive c is no larger than $\frac{\mathbf{E}(X)}{c}$, that is, $\Pr(X \geq c) \leq \frac{\mathbf{E}(X)}{c}$.

Recall that a family of hash functions H from U to $\{0, \ldots, n-1\}$ is 2-universal if for any $x \neq y \in U$, the probability that their key values are equal is bounded by $1/n$. In other words, $\Pr(h(x) = h(y)) \leq 1/n$ for a uniform random function $h \in H$. This celebrated result [2] can also be found in more recent treatments [16, Ch. 4].

We now give a brief summary of the hashing method by Fredman et al. [8] for static data sets. This method represents a given set of items using linear space and entertains constant time existence queries. We do not give the proofs, as some of our proofs for the proposed method in Section 3 follow closely that of Fredman et al. adapted to our case.

Let $U = \{0, \ldots, u-1\}$ be the universe and $S \subseteq U$ with $|S| = n$ be the set to be represented. The FKS method [8] relies on a two-level approach for storing the set S. It needs a prime number p greater than $u-1$. First, it chooses an element $k \in U$ uniformly at random, and defines the *first level* hash function $h_k : U \to \{0, \ldots, n-1\}$ as $h_k(x) = (kx \bmod p) \bmod n$. It then assigns each element x of S to the bucket B_i where $i = h_k(x)$. As the outcome of the hashing function ranges from 0 to $n-1$, there are n buckets. Then, for a bucket B_i containing $b_i > 0$ elements, a storage space of size b_i^2 is allocated, a number $k^{(i)} \in U$ is chosen at random, and the *second level* hash function $h_{k^{(i)}}(x) = (k^{(i)}x \bmod p) \bmod b_i^2$ is defined for items in B_i. A first requirement is that $\sum_i b_i^2$ should be $O(n)$ so that the method uses linear space. The second requirement is that each $k^{(i)}$ should be an injection for the respective bucket. In other words, two different elements in B_i should have different key values computed with the function $h_{k^{(i)}}(\cdot)$. If the hash functions are from a 2-universal family, then both of these requirements are met.

The FKS method constructs a representation of the given set by finding the values of k and $k^{(i)}$ respecting the constraints. These values are found with random sampling and trials. That is, the FKS method randomly chooses a $k \in U$ and computes the size of the buckets when the

first level hash function uses k. If the summation $\sum_i b_i^2$ is smaller than $3n$, then k is accepted, and the method proceeds to the second level. Otherwise, another k is randomly sampled and tried. A similar strategy is adopted for the hash functions in the second level. For the bucket B_i, a random $k^{(i)} \in U$ is chosen, and the mapping defined by $(k^{(i)}x \bmod p) \bmod b_i^2$ is tested to see if it is an injection for the set of elements assigned to B_i. If so, that $k^{(i)}$ is accepted, if not, another one is sampled and tried. Fredman et al. note that with the bound $\sum_i b_i^2 < 3n$, and the injection requirement of b_i^2 space per bucket, one may end up testing many k and $k^{(i)}$. In order to have a construction time of $O(n)$ in expectation, they suggest testing with $\sum_i b_i^2 < 5n$ and using $2b_i^2$ space for each bucket B_i. Under these relaxations, at least one half of the potential k and $k^{(i)}$ values guarantee that the two requirements are met. The bounds $3n$ and $5n$ given in the original paper can be shown to be $2n$ and $4n$, when a 2-universal hash function family is used.

In the FKS method, the membership query for an element $q \in U$ can be answered in constant time by following the construction of the data structure. First, the bucket containing q is found by computing $i = (kx \bmod p) \bmod n$. If the bucket B_i is empty, then q is not in S. If B_i is not empty, the value $\ell = (k^{(i)}x \bmod p) \bmod 2b_i^2$ is computed, the element at location ℓ is compared with q, and the result of the comparison is returned as the answer to the query. The comparison between q and the element stored at the location ℓ is required as more than one element from U can map to ℓ—whereas this can happen for only one element from S.

3 A lean variant of FKS

We discuss our adaptation of the FKS method for the hyperedge queries. We first propose a family of hash functions for the two levels and show that the two requirements of having a total space of $O(n)$ and an injection for each bucket are met. We then propose a technique for reducing the space requirements of the proposed method.

3.1 The hash function and its properties

In a d-partite d-uniform hypergraph $H = (V, E)$, the hyperedge set E is a set of d-tuples, which we will represent for hyperedge queries. Let n denote the number of hyperedges, and p be a prime number larger than n. The universe U contains all d-tuples of the form $[x_0, \ldots, x_{d-1}]$ where x_i is between 0 and $p-1$; in other words $U = \{0, \ldots, p-1\}^d$ and $E \subseteq U$. A potential approach to adapt the FKS for hyperedge queries is to convert d-tuples to unique integers by linearizing them. In this approach, in the case $d = 3$ for example, $[x_0, x_1, x_2]$ can be converted to $x_0 + s_0 \times (x_1 + s_1 \times x_2)$, where s_i corresponds to the size of dimension i; and a longer formula for higher d can be generated similarly. Afterwards, the FKS method can be used without any modification. Such an approach has limited applicability—the numbers get quickly too big for sparse tensors, as also noted by Kolda and Hong [11]. That is why we propose to use d-tuples in definoing the hash functions. Furthermore, since storing d-tuples in the buckets makes the storage requirement depend on d, we store the ids of the hyperedges in the buckets which are taken in the given order.

Let $\mathbf{x}, \mathbf{y}$ be two elements of the universe U. We use $\mathbf{x}^T\mathbf{y} = \sum_{i=0}^{d-1} x_i y_i$ to denote the inner product of the vectors corresponding to $\mathbf{x}$ and $\mathbf{y}$. In the proposed approach, as in the FKS method, a $\mathbf{k} \in U$ is chosen for the first level, and the hash function $h : U \to \{0, \ldots, n-1\}$ is defined as $h(\mathbf{x}) = (\mathbf{k}^T\mathbf{x} \bmod p) \bmod n$. Then, each hyperedge $\mathbf{x} \in E$ is assigned to the bucket B_i where $i = h(\mathbf{x})$. We again use b_i to refer to the number of hyperedges from E that are mapped to B_i.

Lemma 1 below ensures that the linear space requirement can be met for any given set of hyperedges. We give the proof in Appendix A for completeness, where we also explain why the

bound is $4n$, instead of $3n$ as in the original FKS method.

Lemma 1. *For a given set $E \subseteq U$ of n hyperedges, there is a $\mathbf{k} \in U$ such that when $(\mathbf{k}^T\mathbf{x} \bmod p) \bmod n$ is used as the first level hash function, we have $\sum_{i=0}^{n-1} b_i^2 < 4n$.*

Lemma 1 ensures the existence of a d-tuple resulting in a linear space, but does not precise how frequent such d-tuples are in the universe. The following corollary, whose proof is in the appendix, expresses a relaxation of the requirement on $\mathbf{k}$ as done by Fredman et al. to yield many candidates.

Corollary 2. *Let $E \subseteq U$ be a given set of n hyperedges. Then, for at least half of the potential $\mathbf{k} \in U$, when $\mathbf{k}$ is used in the first level hash function, we have $\sum_{i=0}^{n-1} b_i^2 < 7n$.*

Thanks to Corollary 2, one needs a constant number of trials, in expectation, to find a $\mathbf{k}$ respecting the space requirement of $\sum_{i=0}^{n-1} b_i^2 < 7n$.

We next show that for each bucket B_i, if we use a space of size b_i^2, we can map each element to a unique position with a function of the form $({\mathbf{k}^{(i)}}^T\mathbf{x} \bmod p) \bmod b_i^2$. The proof is given in Appendix A for completeness.

Lemma 3. *For each bucket B_i with $b_i > 0$ elements, there is a $\mathbf{k}' \in U$ such that the function $({\mathbf{k}'}^T\mathbf{x} \bmod p) \bmod b_i^2$ is an injection for $p \gg b_i^2$.*

While the bound $\sum_i b_i^2$ from Lemma 1 does not guarantee $p \gg b_i^2$, this must hold in practice for the original FKS and the proposed method to be efficient. A proposition below (Proposition 5) shows that there are many nonempty buckets, suggesting that a large b_i is not very likely. Our experiments also confirm this.

As done by Fredman et al., one can relax the storage requirement of each bucket to have a constant number of trials in expectation to find a $\mathbf{k}'$ defining an injection. This is shown in the following corollary, whose proof is in the appendix.

Corollary 4. *Let B_i be a bucket with $b_i > 0$ elements. For at least half of the d-tuples $\mathbf{k}' \in U$, it holds that the function $({\mathbf{k}'}^T\mathbf{x} \bmod p) \bmod 2b_i^2$ defines an injection for the elements of B_i for $p \gg b_i^2$.*

3.2 Reducing the space requirements

While the previous lemmas show that we can use the FKS method with d-tuples $\mathbf{k}$ and $\mathbf{k}^{(i)}$ for each bucket B_i, there is a catch. For a bucket B_i, we need a space of size d to store $\mathbf{k}^{(i)}$. This results in a space requirement of $O(nd)$ over all buckets. The space requirement will thus be large when the input tensor has a large number n of nonzeros or dimensions d.

An obvious way to reduce the space required to store the $\mathbf{k}^{(i)}$s is to avoid creating such a d-tuple for buckets with at most one element (those buckets B_i with $b_i = 0$ or $b_i = 1$). As we discuss in Proposition 5, one will still need, in expectation over all sets $F \subseteq U$ of n hyperedges, a total of $\Omega(nd)$ space.

Proposition 5. *For a random $\mathbf{k} \neq [0, \ldots, 0]$ and a random set F with n hyperedges, the first level hash function using $\mathbf{k}$ creates at least $n\left(1 - e^{-1+2/p}\right)$ nonempty buckets in expectation, where e is the base of natural logarithm.*

The proof of this proposition can be found in the appendix.

In order to further reduce the memory requirements and obtain a lean variant of FKS for hyperedge queries, we propose to share the second level hash functions among the buckets. This

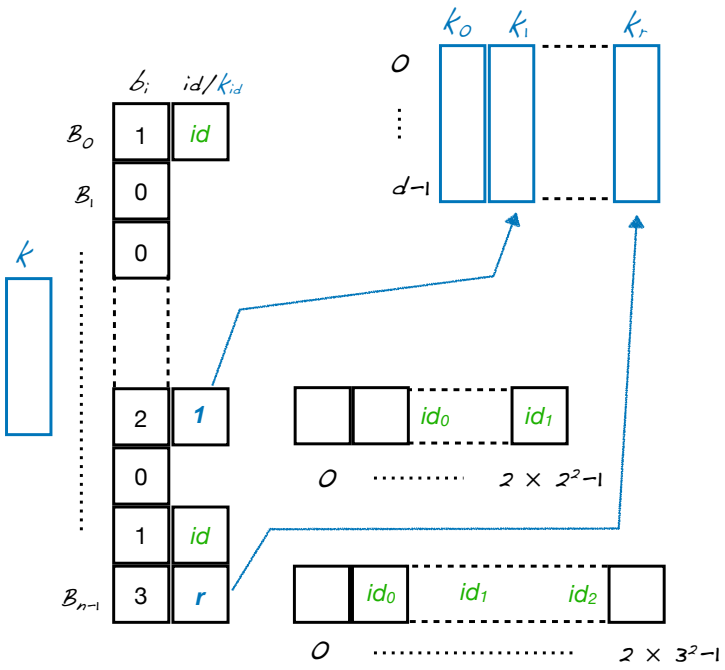


Figure 1 – In the proposed variant of FKS, for each bucket B_i, we store the number b_i of hyperedges assigned to B_i. If b_i is 0, nothing else is stored for B_i. If b_i is 1, the id of the hyperedge mapping to i is stored. If b_i is larger than 1, then the index of a suitable d-tuple from $\mathbf{k}_0$ to $\mathbf{k}_r$ in K is stored, along with a space of size $2b_i^2$ to hold the ids of b_i hyperedges assigned to B_i.

can be achieved by storing a set K of d-tuples and keeping a reference to one suitable element of K for each bucket. The proposed variant of FKS is explained in Figure 1. We keep the first level the same as in the original FKS method and obtain a $\mathbf{k}$ in expected linear time that results in a suitable bound on the memory utilization as in Corollary 2. On the second level, rather than keeping a d-tuple per bucket we keep a set K of tuples with cardinality $|K|$ much smaller than n. The set K is such that for each bucket B_i with $b_i > 1$, there is at least one d-tuple in K defining an injection for the hyperedges in B_i, and B_i keeps a reference id to this tuple. As seen in the figure, for a bucket with a single hyperedge, we just store that hyperedge's id rather than a reference to a tuple of K. For an empty bucket B_i, we only store b_i, which is 0.

We now discuss how to create the set K of tuples and bound the number of d-tuples we need to store in K. As the buckets independently need d-tuples for hashing, a random sequence of d-tuples generated for a bucket is also a random sequence of d-tuples for another one. With this observation in mind let us fix a random sequence of the d-tuples in the universe U. When we need a random d-tuple for a bucket, we try the d-tuples in the fixed random order, until we find one. We then include those d-tuples which were used by at least one bucket in K. Theorem 6 below shows that the number of d-tuples in K is $O(\log_2 n)$ in the expectation.

Theorem 6. *The size of K generated with the above technique will be smaller than $1 + \log_2 n$ in expectation for a given set E of n hyperedges.*

Proof. For each bucket, we know from Corollary 4 that at least half of the tuples from the universe U defines an injection. For a given bucket B_i, let X_i be a random variable counting the number of trials we did to find a suitable $\mathbf{k}^{(i)}$, where at the jth trial we consider the jth tuple from the fixed random sequence. Then,

$$\Pr(X_i = t) \leq \frac{1}{2^t} \text{ for all } t \text{ and } i = 0, \ldots, n-1 \; .$$

Let $X_{i,t}$ be a random variable taking on value 1 if $X_i = t$, and 0 otherwise. Then, the expected number of buckets for which we did t trials to find an injection is

$$\mathbf{E}\left(\sum_{i=0}^{n-1} X_{i,t}\right) \leq \frac{n}{2^t} \; ,$$

by the linearity of expectation. For a given t, the probability that there is a bucket for which we have found a second level hash at the trial t is

$$\Pr(X_i = t \text{ for some } i) = \Pr\left(\sum_{i=0}^{n-1} X_{i,t} \geq 1\right) \leq \mathbf{E}\left(\sum_{i=0}^{n-1} X_{i,t}\right) \leq \frac{n}{2^t} \; ,$$

by Markov's inequality.

Let us define another random variable

$$R_K = \max_i (X_i) \; ,$$

which corresponds to the maximum number of trials required until a second level hash function has been defined for all buckets, and therefore describes the number of d-tuples in K.

We will bound the expectation of R_K to obtain the bound stated in the theorem. We first note that

$$\begin{aligned} \Pr(R_K \geq r) = \Pr(X_i \geq r \text{ for some } i) \leq \sum_{t=r}^{\infty} \Pr(X_i = t \text{ for some } i) &\leq \sum_{t=r}^{\infty} \frac{n}{2^t} \\ &= \frac{n}{2^{r-1}} \; . \end{aligned} \tag{1}$$

The bound obtained in (1) is very large for small values of r. Indeed, when r is smaller than $\log_2 n$, 1 is a better bound on the probability. Therefore, we define a new random variable Y which is equal to $\log_2 n$ if $R_K \leq \log_2 n$ and R_K otherwise. We will bound the expectation of Y. Since $\mathbf{E}(Y) \geq \mathbf{E}(R_K)$, the bound will also apply to R_K and hence to the number of tuples in K. We have

$$\mathbf{E}(Y) = \log_2 n \Pr(R_K \leq \log_2 n) + \sum_{r=1+\log_2 n}^{\infty} \Pr(R_K \geq r) \leq \log_2 n + \sum_{r=1+\log_2 n}^{\infty} \frac{n}{2^{r-1}} ,$$

by using the bound (1) and the fact that $\Pr(R_K \leq \log_2 n) \leq 1$. Since

$$\sum_{r=1+\log_2 n}^{\infty} \frac{n}{2^{r-1}} = \frac{n}{2^{\log_2 n}} = 1$$

we obtain the result $\mathbf{E}(R_K) \leq \mathbf{E}(Y) \leq 1 + \log_2 n$. □

We note that Theorem 6 can also be useful to understand how far from its average value the number of elements in K can be. Indeed, we immediately deduce the following corollary from the proof above.

Corollary 7. *The probability that the number of d-tuples in K exceeds $t \log_2 n + 1$ is bounded by n^{1-t}.*

Another more intuitive way of seeing the $O(\log_2 n)$ bound of Theorem 6 is as follows. A randomly sampled $\mathbf{k}' \in U$ defines an injection with probability more than $1/2$ for a given bucket by Corollary 4. If we try a randomly sampled $\mathbf{k}'$ on all buckets, we will have an injection for half of the buckets in expectation. We can then randomly sample another $\mathbf{k}'$, which will again define an injection for half of the remaining buckets in expectation. By continuing this way, we see that $O(\log_2 n)$ tuples will thus be enough to define all injections, in expectation. This way of seeing the bound of Theorem 6 also leads to an efficient implementation, which we discuss next.

We start with an empty set K and build it incrementally. We examine all non-empty buckets of size greater than 1 one-by-one. While examining the bucket B_i, we first test whether any of the existing tuples in K defines an injection for the hyperedges of B_i. If one such tuple is found, then B_i points to that tuple, and we proceed to examining the next bucket without modifying K. If none of the tuples in K defines an injection for B_i, a new tuple is randomly sampled from the universe until one defining an injection for B_i is found— as before the number of trials per bucket is constant in expectation. Then, the d-tuple found for B_i is inserted into the set K, and B_i points to this newly added d-tuple. We note that this way of incrementally building K is likely to result in less than $1 + \log_2 n$ elements in K, as only d-tuples that define an injection are stored, and we find one such d-tuple for a bucket after a constant number of trials in expectation. The run time complexity of this approach of incrementally building K can be loosely bound as $O(\sum db_i \log_2 n) = O(nd \log_2 n)$, as we have, in expectation, $O(\log_2 n)$ tuples in K, and $O(\log_2 n)$ is a clear upper bound on the number of trials for each bucket. This is so, as each trial for a bucket B_i costs $O(db_i)$ time, and for a bucket one can try $\log_2 n$ tuples. As the first level hash function is found in expected linear time (by Corollary 2), and the second level is built in $O(nd \log_2 n)$ expected time, the proposed method's construction time is $O(nd \log_2 n)$ in expectation.

With the above construction, we store (i) only a special flag for each empty bucket; (ii) the number of elements assigned to each nonempty bucket; (iii) if the number of elements in a bucket is 1, then an index to a hyperedge; (iv) for the others an index to a d-tuple in K. All these add

up to $2n - m + \sum_{i,b_i>1} 2b_i^2$ space where m is the number of empty buckets if we store the number of elements, the ids, and the references to a $\mathbf{k} \in K$ with the same data type. As we saw earlier, the set K adds another $O(d \log_2 n)$ space.

A query for the existence of a hyperedge $\mathbf{q} \in U$ can be answered by first checking the size of the bucket B_i where $i = (\mathbf{k}^T\mathbf{q} \bmod p) \bmod n$. If $b_i = 0$, then $\mathbf{q}$ is not a hyperedge of the given hypergraph. If $b_i = 1$, the query is answered by comparing $\mathbf{q}$ with the hyperedge whose id is stored for B_i. If $b_i > 1$, then the associated d-tuple from K is retrieved as $\mathbf{k}^{(i)}$, and $\ell = (\mathbf{q}^T\mathbf{k}^{(i)} \bmod p) \bmod 2b_i^2$ is computed. If there is an id stored at the location ℓ, then the query is answered by comparing that hyperedge with $\mathbf{q}$. If there is no id at the location ℓ, then $\mathbf{q}$ is not in the hypergraph. As seen here, a query can be read and answered in $O(d)$ time.

3.3 Addressing general hypergraphs

We focus on the d-partite, d-uniform hypergraphs as this is a large class covering the requirements of the tensor decomposition application. The proposed method is not limited to this class. We can apply the algorithms to any hypergraph, without a requirement that the vertices belong to disjoint partitions or that the hyperedges are all of equal size. Let $H = (V, E)$ be a hypergraph and r be the maximum size of a hyperedge in E. Let us assume that 0 is not a member of V. We now define a new hypergraph $H' = (V', E')$ as follows. The vertex set $V' = \bigcup_{i=0}^{r-1} V^{(i)}$ where $V^{(i)} = V \cup \{0\}$ for $i = 0, \ldots, r-1$. The hyperedge set E' contains a hyperedge $\mathbf{e}'$ for each unique hyperedge $\mathbf{e} \in E$ of the original hypergraph. To create $\mathbf{e}'$ from $\mathbf{e}$, we first sort the vertices in $\mathbf{e}$ and use them in this order for the first $|\mathbf{e}|$ dimensions of $\mathbf{e}'$. For the dimensions $|\mathbf{e}|, \ldots, r-1$, we append a 0 to $\mathbf{e}'$ from the corresponding vertex partition $V^{(i)}$. The hypergraph H' is therefore r-uniform and r-partite. Any query hyperedge $\mathbf{q}$ for H can be converted similarly into a query hyperedge $\mathbf{q}'$ for H' by sorting its vertices, and adding the vertex 0 to $\mathbf{q}'$ for all missing dimensions $j = |\mathbf{q}|, \ldots, r-1$ so that $\mathbf{q}'$ becomes of size r as well. A query $\mathbf{q}$ posed on H can therefore be answered equivalently by the query $\mathbf{q}'$ on H'.

The above transformation of padding queries and hyperedges with 0 for missing positions should be done implicitly, otherwise the run time and space requirements can increase prohibitively. In particular, when processing a hyperedge in H', only the original vertices should be used while computing inner products with the $\mathbf{k}$ vectors. If the hyperedges and queries are not sorted at the outset, one needs to sort them, in which case query response time can increase in complexity.

4 Experiments

We compare the proposed algorithm called FKSlean with the following methods which use different hashing schemes:

- HashàlaFKS: the standard average-case constant time hashing method available in `C++ std` as `unordered_map`, with which we propose to use the first level hash function $(\mathbf{k}^T\mathbf{x} \bmod p) \bmod n$ of FKSlean. The tuple $\mathbf{k}$ used for HashàlaFKS enjoys the same properties as that of FKSlean. HashàlaFKS will be used as the baseline.

- BBH [12]: a minimal perfect hash function. It has a parameter γ for which the original paper suggests values 1, 2 and 5, where $\gamma = 2$ strikes a tradeoff between space ($\gamma = 1$) used to store the hash function and the look-up time ($\gamma = 5$). We use BBH with $\gamma = 1$ and $\gamma = 5$.

- RecSplit [7]: another minimal perfect hash function. It has two parameters (`LEAF_SIZE`, `bucket_size`), where the configurations (8,100) and (5,5) are suggested in the original paper. We use RecSplit with these two configurations.
- PTHash [15]: the most recent minimal perfect hash function to the best of our knowledge. The original paper identifies four configurations which we use in our experiments: (1) C-C, $\alpha = 0.99$, $c = 7$, which optimizes the look-up time; (2) D-D, $\alpha = 0.88$, $c = 11$, which optimizes the construction time; (3) EF, $\alpha = 0.99$, $c = 6$, which optimizes the space effectiveness of the function; (4) D-D, $\alpha = 0.94$, $c = 7$, which optimizes the general trade-off.

In a preliminary set of experiments, we also tried RadixSort, as it was used before in the original tensor decomposition application [11]. This method sorts the hyperedges in linear time using radix sort [3, Section 8.3], and then uses a binary search scheme to answer queries. We observed that the query time with RadixSort is about 10 times larger than FKSlean on a large set of instances, and hence deemed it too slow. We thus do not give explicit results with it. In order to use the hashing methods BBH, RecSplit, and PTHash in our context, we created the corresponding MPHFs on the n hyperedges of a given hypergraph, and then used the resulting mapping function to uniquely store the id of each hyperedge in an array of size n. The resulting approaches are called uBBH, uRecSplit, and uPTHash, respectively, where the configurations are specified with (1) and (5) for BBH, (8, 100) and (5,5) for RecSplit, and (1)–(4) for PTHash. While BBH and PTHash have multithreaded construction algorithms, we used one thread to have a fair comparison with the rest of the methods. All codes are compiled with `g++ version 9.2` with options `-O3, -std=c++17 -march=native -mbmi2 -msse4.2` as used in PTHash. We carried out the experiments on a machine having Xeon(R) CPU E7-8890 v4 with a clock-speed of 2.20GHz. All codes are available at `https://gitlab.inria.fr/ucar/hedge-queries`.

4.1 Data set

We present experiments both on real-life and constructed data. We use the real-life data to compare the different algorithms, and use the constructed data to investigate the behavior of different methods with respect to different problem parameters.

We have downloaded tensors from FROSTT [17], and built the associated d-uniform d-partite hypergraphs. The properties of the hypergraphs are shown in Table 1. The hypergraphs in the table are sorted in decreasing order of the number n of hyperedges. The tensors `delicious-3d` and `delicious-4d` contain the same data, with different dimensions. It turns out that the hyperedges are unique with or without the fourth dimension. Therefore, both hypergraphs have the same number of hyperedges. A similar observation was made concerning `flickr-*d`, while the number of hyperedges in `vast-2015-mc1-*d` differ only by 91. We also experimented with three bipartite graphs which correspond to real-life matrices available in The SuiteSparse Matrix Collection [4]. These are listed in the table with $d = 2$. These tensors and matrices arise in diverse applications; ranging from natural language learning (`nell-*`), to e-mail data (sender-receiver-word-date in `enron`), and protein graphs (`kmer_A2a`) to social networks (`com-Orkut`).

The constructed data are built using a model similar to the well-known Erdös-Renyi random graph model. Given a desired number d of parts, a desired number s of vertices in each part, and a desired number n of hyperedges, the model $\mathcal{R}(d, s, n)$ creates a random hypergraph as follows. First, n hyperedges are created by sampling their vertices in the ith part uniformly at random from the range $[0, s)$. Then, duplicate hyperedges are discarded. Note that the number of hyperedges can be slightly smaller than n, and that the maximum element in a part can be different from $s - 1$.

name	d	size in each dimension	n
kmer_A2a	2	170,728,175 ×170,728,175	360,585,172
queen_4147	2	4,147,110 × 4,147,110	329,499,288
com-Orkut	2	3,072,441 × 3,072,441	234,370,166
nell-1	3	2,902,330 × 2,143,368 × 25,495,389	143,599,552
delicious-3d	3	532,924 × 17,262,471 × 2,480,308	140,126,181
delicious-4d	4	532,924 × 17,262,471 × 2,480,308 × 1,443	140,126,181
flickr-3d	3	319,686 × 28,153,045 × 1,607,191	112,890,310
flickr-4d	4	319,686 × 28,153,045 × 1,607,191 × 731	112,890,310
nell-2	3	12,092 × 9,184 × 28,818	76,879,419
enron	4	6,066 × 5,699 × 244,268 × 1,176	54,202,099
vast-2015-mc1-3d	3	165,427 ×11,374 × 2	26,021,854
vast-2015-mc1-5d	5	165,427 × 11,374 × 2 × 100 × 89	26,021,945
chicago_crime	4	6,186 × 24 × 77 × 32	5,330,673
uber	4	183 ×24 × 1,140 × 1,717	3,309,490
lbnl-network	5	1,605 × 4,198 × 1,631 × 4,209 × 868,131	1,698,825

Table 1 – Real-life test data corresponding to the hypergraphs used in the experiments.

4.2 Implementation and measurement details

As BBH, PTHash, and RecSplit accept character strings as input, each hyperedge is converted into a string, both for constructing the associated MPHFs and answering the hyperedge queries. To limit the impact of the conversion on the run time of the corresponding methods, we do this conversion by casting the d consecutive memory locations holding the d vertices of an hyperedge as `(char *)`. We measured the performance of uBBH, uPTHash, and uRecSplit with and without the time needed for casting the hyperedges and queries.

We have implemented FKSlean using an instance of `vector` from `std` per bucket. This has memory allocation and management overhead during the construction phase, as we need to manipulate n vectors. Since the input is static, one could handle all required memory with a single vector (or array), and have reduced construction time. Our choice was made for simplicity to ease future developments.

The construction time reported for FKSlean includes all steps:

i. Find a prime number $p > n$. This step is carried out with a brute force algorithm for testing primality. For $c = n + 1, n + 2, \ldots$, we compute $\mathrm{mod}(c, \ell)$ for $2 \leq \ell \leq \sqrt{c}$ until $\mathrm{mod}(c, \ell) = 0$ for some ℓ, in which case c is not prime; if $\mathrm{mod}(c, \ell) \neq 0$ for all ℓ in the defined range, then c is returned as p. Note that this approach tests at most $n-1$ candidates c for $n > 1$, as there is always a prime number between n and $2n$ for $n > 1$ by Bertrand's postulate [10, p. 455].

ii. Find a $\mathbf{k}$ for the first level hash satisfying Lemma 1 or Corollary 2.

iii. Allocate all n buckets.

iv. Build the set K of d-tuples as described in Section 3.2.

v. Store the ids of the hyperedges in the corresponding places in each bucket.

We report the average of five runs in all measurements per hypergraph. The queries are generated randomly as they would be in the tensor application. A set of nonzero positions (hyperedges) are created and used as queries. At each of the five repetitions of the experiments, the same set of queries are used. The time to generate queries is not reported.

4.3 Comparisons

Table 2 gives the construction time for the ten methods on the real-life hypergraphs from Table 1. The run time of HashàlaFKS is given in seconds, and those of the other methods are given as ratios to that of HashàlaFKS for each hypergraph. The last row of this table gives the geometric mean of the ratios of construction times to that of HashàlaFKS. As seen in this table, on all instances the construction times of uRecSplit-(8,100) and uBBH-(1) are uniformly longer than that of HashàlaFKS, whereas the other seven methods lead to faster construction than HashàlaFKS. uRecSplit-(5,5) and uPTHash-(2) are the fastest methods for construction followed by uPTHash-(4), uPTHash-(1), FKSlean, uBBH-(5), and uPTHash-(3). The order of the MPHF methods obtained concurs with the reported results in recent work [15]. In another set of experiments, we measured the construction time with all instantiations of uRecSplit, uBBH, and uPTHash without taking the time spent in casting the hyperedges to character strings. While these methods gained around 0.05 in proportion to HashàlaFKS's construction time, the overall ranking remained the same.

Table 3 gives the query response time for the ten methods to answer 10^6 queries on real-life hypergraphs. In this table, the query response time of HashàlaFKS is given in seconds. The

		uRecSplit		uBBH		uPTHash				
name	HashàlaFKS	(8,100)	(5,5)	(1)	(5)	(1)	(2)	(3)	(4)	FKSlean
kmer_A2a	591.28	1.64	0.47	1.74	0.71	0.66	0.42	0.93	0.53	0.63
queen_4147	495.18	1.76	0.46	1.72	0.71	0.64	0.40	0.88	0.52	0.73
com-Orkut	355.32	1.73	0.45	1.53	0.69	0.61	0.39	0.84	0.51	0.61
nell-1	213.19	1.75	0.42	1.28	0.75	0.62	0.41	0.81	0.51	0.73
delicious-3d	196.96	1.83	0.43	1.28	0.74	0.63	0.40	0.82	0.51	0.64
delicious-4d	201.39	1.81	0.44	2.00	1.07	0.70	0.47	0.89	0.59	0.78
flickr-3d	163.38	1.78	0.40	1.22	0.71	0.61	0.39	0.78	0.50	0.61
flickr-4d	163.54	1.78	0.41	1.84	1.01	0.66	0.45	0.84	0.56	0.61
nell-2	111.59	1.74	0.38	1.18	0.62	0.58	0.38	0.73	0.48	0.59
enron	72.88	1.90	0.41	1.80	0.89	0.64	0.44	0.79	0.55	0.68
vast-2015-mc1-3d	34.71	1.87	0.38	1.20	0.49	0.52	0.36	0.65	0.44	0.74
vast-2015-mc1-5d	33.40	1.97	0.41	1.99	0.84	0.60	0.43	0.73	0.52	0.72
chicago-crime	5.77	2.29	0.43	1.99	0.76	0.62	0.46	0.72	0.56	0.68
uber	3.17	2.57	0.48	2.21	0.84	0.66	0.50	0.75	0.60	0.75
lbnl-network	1.29	3.23	0.59	2.99	1.15	0.67	0.54	0.76	0.62	0.87
geo-mean		1.94	0.43	1.67	0.78	0.63	0.43	0.79	0.53	0.69

Table 2 – The construction time for the ten methods on real-life tensors. The run time of HashàlaFKS is given in seconds, and those of other methods are given as ratios to that of HashàlaFKS.

response times of the other methods are given as ratios to that of HashàlaFKS. Geometric mean of the ratios of the response times to that of HashàlaFKS are given in the last row. As seen in this table, uBBH-(1) and uBBH-(5) are considerably slower than HashàlaFKS, while uRecSplit-(8,100) and uRecSplit-(5,5) are competitive with HashàlaFKS. uPTHash variants are 20-to-30% faster than HashàlaFKS; as expected the configurations (1) and (4) are faster than the other two configurations. The proposed FKSlean demonstrates at least 2x improvement with respect to HashàlaFKS in all cases except in the smallest hypergraph lbnl-network. By looking at the last row, we also see that FKSlean is 31% faster than the best method from the literature (0.48 vs 0.70).

In order to further investigate the performance difference among FKSlean and the other methods, we measured the query response time of the instantiations of uRecSplit, uBBH, and uPTHash without taking the casting of queries to character strings into account. While such a casting is necessary for the problem at hand, we show these results in order to highlight the importance of taking the dimensionality of the hyperedges into account in designing the hash function. To do so, we stored all queries into character strings, before querying the data structures, and we measured the time spent only in looking-up the query string and comparing the hyperedge stored at the answer. The results are shown in Table 4. As seen in this table, all methods are improved, and uRecSplit variants become faster than HashàlaFKS. Otherwise the ranking of the methods remains the same, with FKSlean being the fastest method.

In order to investigate how the methods behave with respect to the number of hyperedges and the dimension, we present further experiments with the random hypergraph family $\mathcal{R}(d, s, n)$. In a first set of experiments, we investigate how the run time of different methods change with the number n of hyperedges. To do so, we compare all methods on random hypergraphs $\mathcal{R}(d, s, n)$ with $d = 4$, $s = 10^6$ and n taking different values between 10^6 and 10^9. We expect the construction time to increase linearly with the increasing n, and the query response time to remain more or less constant, independent of n, for all methods.

The construction time of different methods are shown in Table 5. Each cell shows the logarithm (in base 10) of the construction time of a method normalized to that of HashàlaFKS

		uRecSplit		uBBH		uPTHash				
name	HashàlaFKS	(8,100)	(5,5)	(1)	(5)	(1)	(2)	(3)	(4)	FKSlean
kmer_A2a	0.78	1.08	1.17	2.45	1.82	0.54	0.60	0.74	0.56	0.49
queen_4147	0.74	1.13	1.14	2.61	1.99	0.54	0.61	0.73	0.56	0.50
com-Orkut	0.73	1.03	1.07	1.70	1.31	0.51	0.56	0.71	0.53	0.47
nell-1	0.72	1.03	1.08	2.61	2.01	0.66	0.77	0.77	0.68	0.48
delicious-3d	0.71	1.08	1.11	2.63	2.07	0.59	0.69	0.67	0.62	0.45
delicious-4d	0.71	1.12	1.19	2.76	2.21	0.93	1.00	0.97	0.95	0.48
flickr-3d	0.71	0.97	0.99	2.14	1.69	0.62	0.73	0.72	0.64	0.46
flickr-4d	0.70	1.11	1.17	2.63	2.14	0.94	0.99	0.95	0.95	0.47
nell-2	0.66	0.98	0.98	2.22	1.86	0.65	0.76	0.76	0.67	0.46
enron	0.64	1.06	1.07	2.13	1.90	0.92	0.97	0.95	0.93	0.47
vast-2015-mc1-3d	0.61	0.95	0.92	1.54	1.36	0.66	0.75	0.76	0.67	0.48
vast-2015-mc1-5d	0.63	1.03	1.01	1.67	1.54	0.89	0.92	0.92	0.89	0.45
chicago-crime	0.56	0.88	0.86	1.16	0.94	0.75	0.78	0.80	0.75	0.47
uber	0.53	0.91	0.86	1.12	1.00	0.75	0.77	0.80	0.75	0.49
lbnl-network	0.46	0.90	0.85	1.20	1.13	0.76	0.79	0.81	0.77	0.54
geo-mean		1.01	1.02	1.94	1.61	0.70	0.77	0.80	0.71	0.48

Table 3 – The query response time for 10^6 queries with the ten methods. That of HashàlaFKS is given in seconds. The response times of the other methods are given as ratio to that of HashàlaFKS. Geometric mean of the ratios of the response times to that of HashàlaFKS are given in the last row.

	uRecSplit		uBBH		uPTHash			
name	(8,100)	(5,5)	(1)	(5)	(1)	(2)	(3)	(4)
kmer_A2a	0.99	1.01	1.54	1.20	0.47	0.50	0.62	0.49
queen_4147	1.07	1.09	2.53	1.89	0.52	0.54	0.70	0.54
com-Orkut	0.96	0.96	1.30	1.03	0.47	0.49	0.61	0.47
nell-1	0.97	0.96	1.78	1.41	0.50	0.52	0.69	0.51
delicious-3d	1.01	1.03	2.56	1.96	0.53	0.55	0.74	0.55
delicious-4d	1.06	1.11	2.65	2.15	0.59	0.59	0.74	0.59
flickr-3d	0.94	0.93	1.57	1.27	0.50	0.52	0.68	0.51
flickr-4d	1.00	1.02	1.57	1.32	0.53	0.54	0.66	0.54
nell-2	0.94	0.91	1.16	1.03	0.49	0.51	0.70	0.51
enron	1.01	1.00	2.01	1.78	0.57	0.58	0.73	0.58
vast-2015-mc1-3d	0.93	0.88	1.47	1.25	0.50	0.50	0.73	0.52
vast-2015-mc1-5d	0.94	0.91	1.54	1.46	0.54	0.50	0.71	0.56
chicago-crime	0.90	0.86	1.11	0.88	0.52	0.51	0.68	0.53
uber	0.88	0.85	1.02	0.93	0.52	0.51	0.66	0.52
lbnl-network	0.83	0.78	1.03	1.02	0.51	0.42	0.59	0.51
geo-mean	0.96	0.95	1.58	1.32	0.52	0.52	0.68	0.53

Table 4 – The query response time for 10^6 queries with all instantiations of uRecSplit, uBBH, and uPTHash. The response times are given as ratios to that of HashàlaFKS presented in Table 3. Geometric mean of the ratios of the response times to that of HashàlaFKS are given in the last row. The time spent in casting queries as character strings are not included.

		uRecSplit		uBBH		uPTHash				
n	HashàlaFKS	(8,100)	(5,5)	(1)	(5)	(1)	(2)	(3)	(4)	FKSlean
1,000,000	0.00	0.62	-0.13	0.54	0.12	-0.03	-0.13	0.00	-0.07	-0.03
10,000,000	1.30	1.63	0.93	1.58	1.19	1.11	0.95	1.17	1.05	1.17
100,000,000	2.38	2.65	2.01	2.65	2.39	2.22	2.04	2.32	2.14	2.22
1,000,000,000	3.50	3.69	3.19	3.93	3.50	3.56	3.21	3.77	3.43	3.30

Table 5 – Construction time in seconds for all the studied methods on random hypergraphs. Each value is normalized to the construction time of HashàlaFKS for n =1,000,000 and the logarithm of the normalized values are shown.

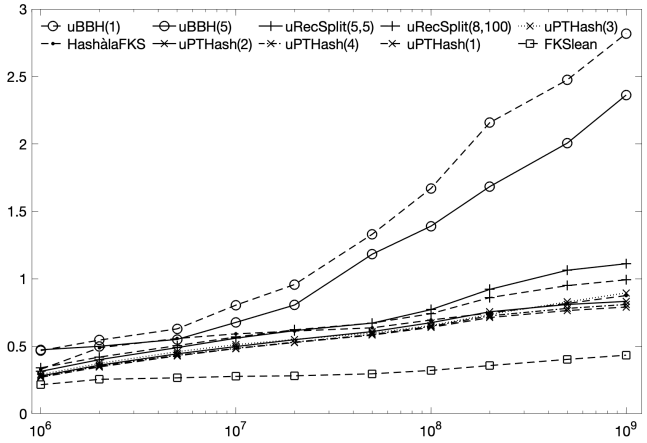
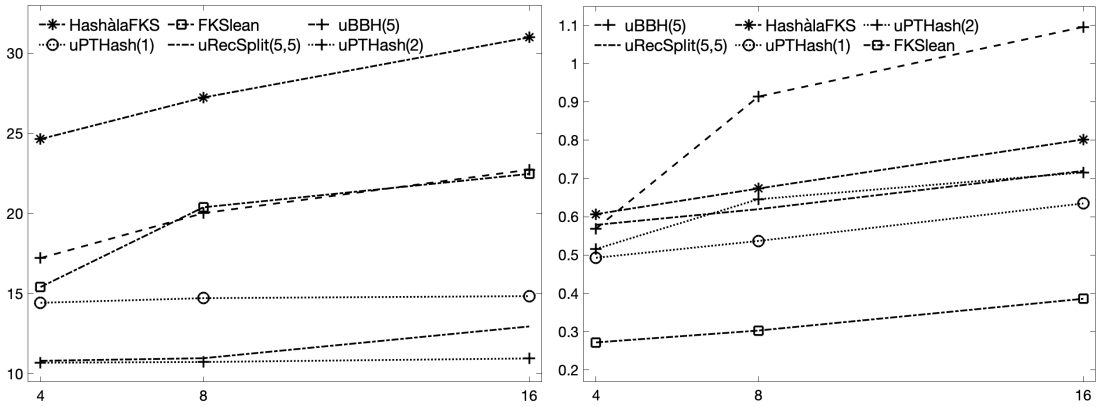


Figure 2 – The response time for 10^6 queries, in seconds, for all methods on the random hypergraphs. At $n = 10^9$, the plots correspond to, from top to bottom, to the methods as listed in the legend (from left to right, top to bottom).

for $n = 1,000,000$. This normalization helps us see how the run time increases between two consecutive values of n. All of the studied data structures are constructed in almost linear time. Indeed, when multiplying the size of the hypergraph by 10^3, the construction time is at most multiplied by $10^{3.8}$ for uPTHash-(3). This might be due to cache effects, which is more important for $n = 10^6$ but not as much so for the larger values of n. The order of the methods are largely compatible with the order seen on the real-life instances: uRecSplit-(5,5) and uPTHash-(2) are the fastest methods; uPTHash-(4), uPTHash-(1), FKSlean, uBBH-(5), and uPTHash-(3) usually follow them, while uRecSplit-(8,100) and uBBH-(1) being the slowest two.

The query response time with different methods on the $\mathcal{R}(d, s, n)$ family hypergraphs with the same parameters as before is shown in Figure 2 where the x-axis is in the log -scale. In this figure, the ranking of the methods is as seen before on the real-life data. FKSlean is faster than the others, and the difference is visible for all values of n. Indeed, on the largest size of hypergraphs we tested, FKSlean is almost twice as fast as the second fastest method, uPTHash-(1), which is built to optimize the look-up time. This figure also shows that the query response time of all methods increases slightly with the increasing number of hyperedges. This is understandable, as with larger n, the data structures become larger, and the random accesses due to hashing in the data structures requires more time.



(a) Construction time (b) Query response time

Figure 3 – The construction time and the response time for 10^6 queries, in seconds, for six methods in hypergraphs from the family $\mathcal{R}(d, s, n)$ for $d = \{4, 8, 16\}$, $s = 10^6$, and $n = 2 \times 10^7$. At $d = 8$, the plots correspond to, from top to bottom, the methods as listed in the legends (from left to right, top to bottom).

We next investigate the behavior of the methods under study with respect to d. We restrict the analysis to uRecSplit-(5,5), uBBH-(5), uPTHash-(1), uPTHash-(2), HashàlaFKS, and FKSlean as others are not as competitive. Figure 3a and Figure 3b show the construction and query response time of the six methods on the random hypergraphs $\mathcal{R}(d, s, n)$ where $d = \{4, 8, 16\}$, $s = 10^6$, and $n = 2 \times 10^7$. As seen in these figures, the ranking of the methods with respect to the construction time and the query response time are the same as before. Figure 3a reveals that uPTHash variants have more stable construction time with respect to increasing d, and hence length of keys. Figure 3b shows that the query response time of all methods have a similar trend with respect to increasing d, apart from uBBH-(5) whose query response time increases more.

We now look at the space utilization of FKSlean, and see how the theoretical properties shown in Lemma 1, Proposition 5, and Theorem 6 compare with the results in practice. We start with the quantity $\sum_{i=0}^{n-1} b_i^2$ that we bound as $4n$ in Lemma 1, and the total space requirement of FKSlean (except the set K), which is $2n - m + \sum_{i,b_i>1} 2b_i^2$, where m is the number of empty buckets. In all our experiments, the number of nonempty buckets and hence the number m of empty buckets were clustered around their expected values. In particular, the number of nonempty buckets was observed to be between $0.6315n$ and $0.6484n$. Figure 4 shows that $\sum_{i=0}^{n-1} b_i^2$ is always close to $2n$ in these experiments. The largest value of $\sum_{i=0}^{n-1} b_i^2$ is $2.0038n$, obtained for the second hypergraph (`Queen_4147`), and the smallest value was $1.9037n$, obtained for the last hypergraph (`lbnl-network`). This observation prompts us to look for a tighter analysis of Lemma 1. Accordingly, the total space requirement $2n - m + \sum_{i,b_i>1} 2b_i^2$ of FKSlean (except the set K) was less than $5n$. The largest and smallest values of $2n - m + \sum_{i,b_i>1} 2b_i^2$, are $4.9042n$ and $4.6945n$ and are obtained for the same hypergraphs H2 and H15. We also looked at the maximum value of a bucket size $\max_i b_i$ obtained in any hypergraph in our data set (real-life and random), and saw that it was always less than 13. Lastly, we look at the number of d-tuples stored in K, that is the total number of different $\mathbf{k}^{(i)}$, which is bounded by $1 + \log_2 n$ in Theorem 6. In all instances the ratio of the number r of tuples in K to $\log_2 n$ is observed to be between 0.36 and 0.42. This confirms the suitability of the proposed approach of constructing K incrementally.

The `unordered_map` from `C++ std` reports the load factor `LF`, which is the number of elements

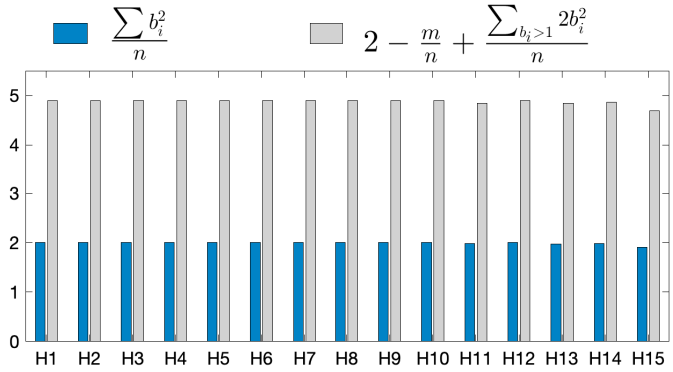


Figure 4 – The bar charts of $\frac{\sum b_i^2}{n}$ and $(2 - \frac{m}{n}) + \sum_{i,b_i>1} \frac{2b_i^2}{n}$, where $m$is the number of empty buckets. The hypergraphs are named as Hi, for $i = 1, \ldots, 15$ and are given in the same order as Table 1.

n divided by the number of buckets used, of a map. Inverse of `LF` is thus proportional to the memory requirements. In the experiments with the real-life instances, `LF` was observed to be in between 0.9268 and 0.9971, and similarly with the random hypergraphs, it was in between 0.9370 and 0.9984. HashàlaFKS is therefore space-efficient, requiring around $1.08n$ locations.

4.4 Summary of comparisons

The methods uRecSplit-(5,5) and uPTHash-(2) usually have the shortest construction time. The methods uPTHash-(1), uPTHash-(4) and the proposed FKSlean follow them, which are then followed by uPTHash-(3) and uBBH-(5). The methods uRecSplit-(8,100) and uBBH-(1) have the largest construction time; even larger than HashàlaFKS. The proposed FKSlean has the shortest look-up time, this is so even without taking the time spent in converting hyperedges to strings. In the tensor application, the hash tables are built once, and queries for zeros of the tensor are issued at each iteration. Therefore, the construction time of different hashing methods can be amortized over the iterations, and across different decompositions of the same tensor. With these in mind FKSlean becomes the method of choice, especially for large d, n, or the number of queries q. HashàlaFKS could also be useful because of its simplicity. One just needs to implement methods to find a prime number $p > n$ and to compute $(\mathbf{k}^T\mathbf{x} \bmod p) \bmod n$. The `unordered_map` from `C++ std` efficiently takes care of the rest.

One aspect we did not deal with is the bits-per-key complexity of hash functions, which measures the storage required to represent a minimum perfect hash function. We did not concern ourselves with this complexity measure, as the motivating application stores the hyperedges with a total size of $O(nd)$ in order to answer queries for zeros of the tensor. In these cases, the bits-per-key complexity is not deemed to be an important aspect [12]. To store the hash functions of FKSlean, one needs at least to store $\mathbf{k}$, the number of elements and the index of a d-tuple in K for each bucket with mode than one elements—one does not store the ids of hyperedges in buckets, see Figure 1. Since the number of d-tuples in K is $O(\log_2 n)$, one needs $\Omega(\log_2 \log_2(n))$ bits to store the id of an d-tuple per bucket. That is, the bits-per-key complexity will be $\Omega(\log_2 \log_2(n))$. This amount is larger than 4 even when $n = 10^6$, which is already much larger than the bits-per-key complexity of the current state-of-the-art MPHFs.

5 Conclusion

We investigated the problem of answering queries asking for the existence of a given hyperedge in a given hypergraph, with a special focus on d-partite, d-uniform hypergraphs arising in a tensor decomposition application. We proposed a perfect hashing method called FKSlean based on a well-known approach [8]. FKSlean has provably smaller space requirements than a direct adaptation of the original approach, thanks to the reuse of hash functions. Experimental results demonstrated in practice that the space requirement is in fact less than $5n$ plus an additional $O(d \log_2 n)$ term for storing the shared hash functions. We compared FKSlean with the methods using the current state-of-the-art minimal perfect hash functions (MPHFs), and the `unordered_map` from `C++ std` equipped with the hash function used by FKSlean. Experiments on real-life and constructed data showed that FKSlean obtains the shortest query response time among all alternatives while having a construction time in the middle of the pack.

We have four lines of future work. On the application of interest with tensors, we need an implementation of the stochastic gradient method in an efficient library (instead of in Matlab) to test the effects of the proposed method in that particular application. On the more algorithmic side, we plan to address the dynamic case where hyperedges come and go. A suitable starting point is given by Dietzfelbinger and others [6]. On the theoretical side, we observed that for any randomly chosen d-tuple $\mathbf{k}$ in the first level hashing, $\sum_i b_i^2$ was less than $3n$ in all experiments, while being clustered around $2n$. Can this tighter bound be shown theoretically? We also want to parallelize the construction of the proposed data structure.

Acknowledgment

We thank Ioannis Panagiotas for his comments on a preliminary version of this report.

References

[1] F. C. Botelho, R. Pagh, and N. Ziviani. Practical perfect hashing in nearly optimal space. *Information Systems*, 38(1):108–131, 2013.

[2] J. L. Carter and M. N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154, 1979.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 3rd edition, 2009.

[4] T. A. Davis and Y. Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software*, 38(1):1:1–1:25, 2011.

[5] M. Dietzfelbinger and S. Walzer. Dense peelable random uniform hypergraphs. In M. A. Bender, O. Svensson, and G. Herman, editors, *27th Annual European Symposium on Algorithms (ESA 2019)*, volume 144 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 38:1–38:16, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[6] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing*, 23(4):738–761, 1994.

[7] E. Esposito, T. Mueller Graf, and S. Vigna. RecSplit: Minimal perfect hashing via recursive splitting. In *Proceedings of the Symposium on Algorithm Engineering and Experiments (ALENEX)*, pages 175–185, Philadelphia, PA, 2020. SIAM.

[8] M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *J. ACM*, 31(3):538–544, 1984. ISSN 0004-5411.

[9] M. Genuzio, G. Ottaviano, and S. Vigna. Fast scalable construction of minimal perfect hash functions. In *15th International Symposium on Experimental Algorithms*, pages 339–352, St. Petersburg, Russia, 2016. Springer-Verlag.

[10] G. H. Hardy and E. M. Wright. *An Introduction to the Theory of Numbers*. Oxford University Press, 6th edition, 2008.

[11] T. G. Kolda and D. Hong. Stochastic gradients for large-scale tensor decomposition. *SIAM Journal on Mathematics of Data Science*, 2(4):1066–1095, 2020.

[12] A. Limasset, G. Rizk, R. Chikhi, and P. Peterlongo. Fast and Scalable Minimal Perfect Hashing for Massive Key Sets. In C. S. Iliopoulos, S. P. Pissis, S. J. Puglisi, and R. Raman, editors, *16th International Symposium on Experimental Algorithms (SEA 2017)*, volume 75 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:16, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-036-1. doi: 10.4230/LIPIcs.SEA.2017.25. URL `http://drops.dagstuhl.de/opus/volltexte/2017/7619`.

[13] I. Müller, P. Sanders, R. Schulze, and W. Zhou. Retrieval and perfect hashing using fingerprinting. In J. Gudmundsson and J. Katajainen, editors, *International Symposium on Experimental Algorithms*, pages 138–149, Copenhagen, Denmark, 2014. Springer International Publishing.

[14] R. Pagh and F. F. Rodler. Cuckoo hashing. In F. M. auf der Heide, editor, *Algorithms — ESA 2001*, pages 121–133. Springer Berlin Heidelberg, 2001.

[15] G. E. Pibiri and R. Trani. Pthash: Revisiting FCH minimal perfect hashing. In *44th SIGIR, International Conference on Research and Development in Information Retrieval (to appear)*, 2021. URL `https://arxiv.org/abs/2104.10402`.

[16] P. Sanders, K. Mehlhorn, M. Dietzfelbinger, and R. Dementiev. *Sequential and Parallel Algorithms and Data Structures: The Basic Toolbox*. Springer International Publishing, 2019.

[17] S. Smith, J. W. Choi, J. Li, R. Vuduc, J. Park, X. Liu, and G. Karypis. FROSTT: The formidable repository of open sparse tensors and tools. Available at `http://frostt.io/`, 2017.

[18] S. Walzer. *Random hypergraphs for hashing-based data structures*. PhD thesis, Technische Universität Ilmenau, Germany, 2020.

[19] S. Walzer. Peeling close to the orientability threshold–spatial coupling in hashing-based data structures. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2194–2211. SIAM, 2021.

A Omitted proofs

We give the omitted proofs from Section 3. For convenience, we repeat the body of the lemmas and corollaries.

We start with Lemma 1 repeated below. Its proof follows closely the original proof by Fredman et al. and is given for completeness. After the proof, we explain why the upper bound is higher than that of the original theorem of Fredman et al.

▶ **Restatement of Lemma 1**: For a given set $E \subseteq U$ of n hyperedges, there is a $\mathbf{k} \in U$ such that when $(\mathbf{k}^T\mathbf{x} \bmod p) \bmod n$ is used as the first level hash function, we have $\sum_{i=0}^{n-1} b_i^2 < 4n$.

Proof. For a given $\mathbf{k}$, let $b_i^{(\mathbf{k})}$ denote the number of hyperedges in E having the same hash value $i = (\mathbf{k}^T\mathbf{x} \bmod p) \bmod n$. The number of two-element subsets $\{\mathbf{x}, \mathbf{y}\}$ of E with the same hash value i is therefore $\binom{b_i^{(\mathbf{k})}}{2}$, that is $\frac{b_i^{(\mathbf{k})}(b_i^{(\mathbf{k})}-1)}{2}$. We will compute $\sum_{\mathbf{k}\in U}\sum_{i=0}^{n-1}\frac{b_i^{(\mathbf{k})}(b_i^{(\mathbf{k})}-1)}{2}$ so that we can obtain the average number of two-element subsets of E with the same hash value over all potential $\mathbf{k}$ tuples. As there is at least one $\mathbf{k}'$ for which the corresponding sum $\sum_{i=0}^{n-1}\frac{b_i^{(\mathbf{k}')}(b_i^{(\mathbf{k}')}-1)}{2}$ is no larger than the average, we will use that $\mathbf{k}'$ to attain the bound stated in the lemma.

We first observe that

$$\sum_{\mathbf{k}\in U}\sum_{i=0}^{n-1}\frac{b_i^{(\mathbf{k})}(b_i^{(\mathbf{k})}-1)}{2} = \sum_{\substack{\mathbf{x},\mathbf{y}\in E\\ \mathbf{x}\neq\mathbf{y}}} \left|\{\mathbf{k}\in U : (\mathbf{k}^T\mathbf{x} \bmod p) \bmod n = (\mathbf{k}^T\mathbf{y} \bmod p) \bmod n\}\right| , \tag{2}$$

as the right hand side counts the total number of times any two different hyperedges $\mathbf{x}, \mathbf{y}$ of E have the same value for hash value over all $\mathbf{k} \in U$.

We will bound the right hand side of (2) from above. For this, we need a bound on the number of different $\mathbf{k}$ for which any $\mathbf{x}, \mathbf{y} \in E$ give the same value

$$(\mathbf{k}^T\mathbf{x} \bmod p) \bmod n = (\mathbf{k}^T\mathbf{y} \bmod p) \bmod n \ . \tag{3}$$

In other words, we want to count the number of different $\mathbf{k}$ for which we have

$$(\mathbf{k}^T\mathbf{x} - \mathbf{k}^T\mathbf{y}) \bmod p \in \left\{0, \pm n, \pm 2n, \ldots, \pm\left\lfloor\frac{p-1}{n}\right\rfloor n\right\} \ . \tag{4}$$

This general formula applies to any $p > n$. In the proposed algorithm, as in the original FKS method, p is a prime number larger than n, which we find by testing the numbers $n+1, n+2, \ldots$ for primality and accept the first prime as p. By Bertrand's postulate [10, p. 455] there is at least one prime number r with $t < r < 2t$ for every integer $t > 1$. Therefore the set on the right hand side of (4) contains three numbers $\{0, \pm n\} \equiv \{0, p-n, n\}$.

Since $\mathbf{x} \neq \mathbf{y}$, there is at least one dimension $0 \leq \ell < d$ such that $x_\ell \neq y_\ell$. Let us arbitrarily pick one such ℓ and write

$$\left(k_\ell(x_\ell - y_\ell) + \sum_{j\neq\ell} k_j(x_j - y_j)\right) \bmod p \in \{0, p-n, n\} \ . \tag{5}$$

Since we treat each pair $\mathbf{x}, \mathbf{y}$ once, we can assume $x_\ell > y_\ell$ without loss of generality. Let us take a value v from the right hand side of (5) and fix $\left(k_\ell(x_\ell - y_\ell) + \sum_{j\neq\ell} k_j(x_j - y_j)\right) \bmod p = v$. We will count the number of $\mathbf{k}$ tuples for which this equality holds. Then, multiplying with

the number of elements in the right hand size, 3, will give the total number of times any pair $\mathbf{x}, \mathbf{y} \in E$ satisfies (3).

At (5), we can freely set any k_j for $j \neq \ell$ and k_ℓ must then be chosen accordingly with these selections to make the equation hold. Because p is prime, for each distinct configuration of the k_j values for $j \neq \ell$, there is a unique value of k_ℓ that makes (5) hold. In other words, for any value in the right hand side of (4), there are p^{d-1} different $\mathbf{k}$ tuples, which are formed by considering all different p values for each k_j for $j \neq \ell$.

As there are 3 different right hand side values in (4), and for each one we have at most p^{d-1} different $\mathbf{k}$ tuples satisfying (3), we obtain an upper bound on the right hand side of (2) as

$$\sum_{\substack{\mathbf{x},\mathbf{y}\in S \\ \mathbf{x}\neq\mathbf{y}}} |\{\mathbf{k} \in U : (\mathbf{k}^T\mathbf{x} \bmod p) \bmod n = (\mathbf{k}^T\mathbf{y} \bmod p) \bmod n\}| \leq p^{d-1} 3 \frac{n(n-1)}{2} ,$$

since there are $\frac{n(n-1)}{2}$ pairs $\mathbf{x}, \mathbf{y}$.

Combining with the left hand side of (2), we see that

$$\sum_{\mathbf{k}\in U} \sum_{i=0}^{n-1} \frac{b_i^{(\mathbf{k})}(b_i^{(\mathbf{k})} - 1)}{2} \leq p^{d-1} 3 \frac{n(n-1)}{2} .$$

Since there are p^d different $\mathbf{k}$, for at least one of them the inner sum $\sum_{i=0}^{n-1} \frac{b_i^{(\mathbf{k})}(b_i^{(\mathbf{k})}-1)}{2}$ should be no larger than the average. That is,

$$\sum_{i=0}^{n-1} \frac{b_i^{(\mathbf{k})}(b_i^{(\mathbf{k})} - 1)}{2} \leq 3 \frac{n}{p} \frac{n-1}{2} , \tag{6}$$

for a $\mathbf{k}$. Let $\mathbf{k}'$ denote the tuple attaining that bound. Then,

$$\sum_{i=0}^{n-1} b_i^{(\mathbf{k}')} b_i^{(\mathbf{k}')} - \sum_{i=0}^{n-1} b_i^{(\mathbf{k}')} \leq 3(n-1) ,$$

as $n < p$. Since $\sum_{i=0}^{n-1} b_i^{(\mathbf{k}')} = n$, we obtain the bound stated in the lemma. □

In the original FKS method, we have scalar values. Therefore the equivalent of (4) is

$$k(x-y) \bmod p \in \{0, \pm n, \pm 2n, \ldots, \pm \left\lfloor \frac{p-1}{n} \right\rfloor n\} .$$

Since p is prime and both k and $x-y$ are less than p, the equality $k(x-y) \bmod p = 0$ cannot hold, and the set on the right hand side is $\{\pm n, \pm 2n, \ldots, \pm \left\lfloor \frac{p-1}{n} \right\rfloor n\}$, or $\{n, p-n\}$, when $n < p < 2n$.

► **Restatement of Corollary 2**: Let $E \subseteq U$ be a given set of n hyperedges. Then, for at least half of the potential $\mathbf{k} \in U$, when $\mathbf{k}$ is used in the first level hash function, we have $\sum_{i=0}^{n-1} b_i^2 < 7n$.

Proof. Let X be the random variable representing $\sum_i \frac{b_i^{(\mathbf{k})}(b_i^{(\mathbf{k})}-1)}{2}$ when we randomly choose $\mathbf{k}$. Then, by (6) and the fact that $p > n$, we have $\mathbf{E}(X) \leq 3\frac{n}{p}\frac{n-1}{2} \leq 3\frac{(n-1)}{2}$. By Markov's inequality, $\Pr(X \geq 3(n-1)) \leq \frac{\mathbf{E}(X)}{3(n-1)}$, and hence $\Pr(X \geq 3n) \leq \frac{1}{2}$. Therefore, we have $\Pr(X < 3n) \geq \frac{1}{2}$, and hence for at least half of the randomly chosen $\mathbf{k}$ we have $\sum_i \frac{b_i^{(\mathbf{k})}(b_i^{(\mathbf{k})}-1)}{2} < 3n$. The event that $\sum_i b_i^2 < 7n$ is identical to the event $2X + n < 7n$, and hence holds with probability no smaller than $1/2$. Therefore, at least half of $\mathbf{k} \in U$ satisfies the bound of the corollary. □

▶ **Restatement of Lemma 3**: For each bucket B_i with $b_i > 0$ elements, there is a $\mathbf{k}' \in U$ such that the function $(\mathbf{k'}^T\mathbf{x} \bmod p) \bmod b_i^2$ is an injection for $p \gg b_i^2$.

Proof. The proof follows the same approach used in proving Lemma 1. There are two differences: here we have at most $2\left\lfloor \frac{p-1}{b_i^2} \right\rfloor + 1$ potential values for a pair to have an equal hash value (the equivalent of (4)), and the total number of pairs is $\frac{b_i(b_i-1)}{2}$ instead of $\frac{n(n-1)}{2}$. This leads to an average (over all possible $\mathbf{k}$) no larger than one for each position in the storage space of size b_i^2 for $p \gg b_i^2$. And hence, $\mathbf{k}'$ can be chosen. □

▶ **Restatement of Corollary 4**: Let B_i be a bucket with $b_i > 0$ elements. For at least half of the d-tuples $\mathbf{k}' \in U$, it holds that the function $(\mathbf{k'}^T\mathbf{x} \bmod p) \bmod 2b_i^2$ defines an injection for the elements of B_i for $p \gg b_i^2$.

Proof. We follow the proof of Lemma 1 (and Lemma 3), while replacing E by B_i, n by b_i, and by defining $b_j^{(\mathbf{k}')}$ to be the number of elements of B_i that map to $j \in \{0, \ldots, 2b_i^2 - 1\}$ with the function $(\mathbf{k'}^T\mathbf{x} \bmod p) \bmod 2b_i^2$. We obtain the inequality

$$\sum_{\mathbf{k}' \in U} \sum_{j=0}^{2b_i^2-1} \frac{b_j^{(\mathbf{k}')}(b_j^{(\mathbf{k}')} - 1)}{2} \leq \frac{|U|}{p} \left(2 \left\lfloor \frac{p-1}{2b_i^2} \right\rfloor + 1 \right) \frac{b_i(b_i-1)}{2} .$$

By arithmetic simplification and using $p \gg b_i^2$, we obtain

$$\sum_{\mathbf{k}' \in U} \sum_{j=0}^{2b_i^2-1} \frac{b_j^{(\mathbf{k}')}(b_j^{(\mathbf{k}')} - 1)}{2} \leq |U| \frac{1}{2} . \tag{7}$$

Let X be the random variable represeting $\sum_{j=0}^{2b_i^2-1} \frac{b_j^{(\mathbf{k}')}(b_j^{(\mathbf{k}')}-1)}{2}$ when we randomly choose $\mathbf{k}'$. By (7),

$$\mathbf{E}(X) \leq \frac{1}{2} ,$$

over all potential $\mathbf{k}' \in U$. For a randomly chosen $\mathbf{k}'$ not to be an injection, $b_j^{(\mathbf{k}')} \geq 2$ must hold for some $j \in \{0, \ldots, 2b_i^2 - 1\}$. In that case, we will have the event $X \geq 1$. By Markov's inequality,

$$\Pr(X \geq 1) \leq \mathbf{E}(X) \leq \frac{1}{2} .$$

Therefore, $\Pr(X < 1) \geq \frac{1}{2}$, and hence at least half of the potential $\mathbf{k}' \in U$ defines an injection for B_i for $p \gg b_i^2$. □

▶ **Restatement of Proposition 5**: For a random $\mathbf{k} \neq [0, \ldots, 0]$ and a random set F with n hyperedges, the first level hash function using $\mathbf{k}$ creates at least $n\left(1 - e^{-1+2/p}\right)$ nonempty buckets in expectation, where e is the base of natural logarithm.

Proof. For each $\mathbf{x} \in U$ let us define a random variable $R_{\mathbf{x}}^{\mathbf{k}}$ taking on the value $(\mathbf{k}^T\mathbf{x} \bmod p) \bmod n$ for a tuple $\mathbf{k}$. We will compute for any tuple $\mathbf{k}$, the number of sets $F \subseteq U$ of cardinality n such that for all $\mathbf{x}$ in F we have $R_{\mathbf{x}}^{\mathbf{k}} \neq i$. We will obtain for each value i the probability over $\mathbf{k}$ and F that bucket B_i is non-empty. By summing over i, we will obtain the expected number of non-empty buckets for a randomly chosen $\mathbf{k}$ and a random set F of size n.

We first compute the expected value over $\mathbf{k}$ and F of the random variable $R^{\mathbf{k}}_{F,i} = |\{\mathbf{x} \in F | R^{\mathbf{k}}_{\mathbf{x}} \neq i\}|$. For a fixed tuple $\mathbf{k} \neq [0, \ldots, 0]$, let us find the number of $\mathbf{x} \in U$ such that $R^{\mathbf{k}}_{\mathbf{x}} = i$. If $R^{\mathbf{k}}_{\mathbf{x}} = i$, we should have

$$\mathbf{k}^T\mathbf{x} \bmod p \in \left\{ i, i \pm n, i \pm 2n, i \pm 3n, \ldots, i \pm \left\lfloor \frac{p-1-i}{n} \right\rfloor n \right\} . \tag{8}$$

This means that there are

$$t_i = 2 \left\lfloor \frac{p-1-i}{n} \right\rfloor + 1$$

possible values for $\mathbf{k}^T\mathbf{x} \bmod p$. For any value j in the right hand size of (8), let us consider the tuples $\mathbf{x}^j$ such that $\mathbf{k}^T\mathbf{x}^j \bmod p = j$. Since $\mathbf{k}$ is not uniformly zero, there is an index ℓ such that $k_\ell \neq 0$. Then, for any of the p^{d-1} possible values of x^j_r, where $r \neq \ell$, there exists one unique value of x^j_ℓ such that $\mathbf{k}^T\mathbf{x}^j \bmod p = j$. Thus, there exist p^{d-1} such $\mathbf{x}^j$ tuples for j. This yields a total of $p^{d-1}t_i$ alternatives for $\mathbf{x}$ such that (8) holds, and hence for which $R^{\mathbf{k}}_{\mathbf{x}} = i$, among all p^d elements of U.

There are therefore $\binom{p^{d-1}\cdot(p-t_i)}{n}$ sets F for which $R^{\mathbf{k}}_{\mathbf{x}} \neq i$, where the symbol $\binom{a}{b} = \frac{a!}{b!(a-b)!}$ is the binomial coefficient. Then, for a fixed tuple $\mathbf{k}$, the probability that none of the elements of a random F maps to i is

$$\Pr(\cap_{\mathbf{x}\in F} R^{\mathbf{k}}_{\mathbf{x}} \neq i) = \frac{\binom{p^{d-1}\cdot(p-t_i)}{n}}{\binom{p^d}{n}} .$$

We can thus bound $\Pr(R^{\mathbf{k}}_{F,i} = 0)$ as follows:

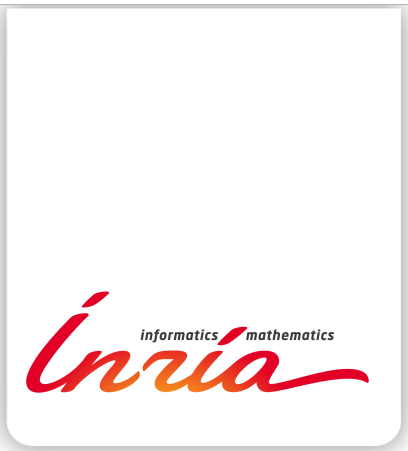
$$\begin{aligned}
\Pr(R^{\mathbf{k}}_{F,i} = 0) &= \Pr(\cap_{\mathbf{x}\in F} R^{\mathbf{k}}_{\mathbf{x}} \neq i) \\
&= \frac{\binom{p^{d-1}\cdot(p-t_i)}{n}}{\binom{p^d}{n}} \\
&= \frac{\big(p^{d-1}\cdot(p-t_i)\big)!}{(p^{d-1}\cdot(p-t_i)-n)!} \frac{(p^d-n)!}{p^d!} \\
&= \prod_{j=0}^{n-1} \frac{\big(p^{d-1}\cdot(p-t_i)\big)-j}{p^d-j} \\
&= \prod_{j=0}^{n-1} \left(1 - \frac{p^{d-1}\cdot t_i}{p^d-j}\right) \\
&\leq \left(1 - \frac{p^{d-1}\cdot t_i}{p^d}\right)^n \\
&\leq \left(1 - \frac{t_i}{p}\right)^n \\
&\leq e^{-\frac{n\cdot t_i}{p}} \leq e^{-1+2/p},
\end{aligned}$$

as $\frac{n\cdot t_i}{p} \geq 2 - \frac{2}{p} - \frac{n}{p} \geq 1 - \frac{2}{p}$ by the definition of t_i and the fact that $p > n > i$.

By defining a binary variable $Y_i = 1$ if bucket B_i is empty and another one $Z = \sum_i Y_i$, we see that the expected number of empty buckets is

$$\mathbf{E}(Z) = \sum_{i=0}^{n-1} \mathbf{E}(Y_i) \leq n e^{-1+\frac{2}{p}} ,$$

for a randomly chosen $\mathbf{k}$ and a random set of n hyperedges, which concludes the proof. □



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399