Algorithms and data structures for hyperedge queries

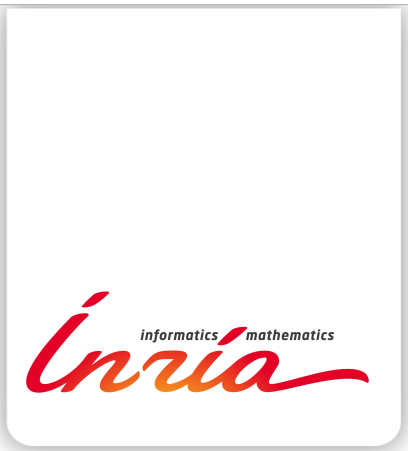
Jules Bertrand, Fanny Dufossé, Bora Uçar

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 — February 2021 — 21 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 shorter query response time than the other considered alternatives, while having the shortest or the second shortest construction time.

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 basé 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, avec le temps de construction le plus court ou le deuxième plus court.

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 requirement 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 sampled uniformly at random. For sampling zeros in a d-dimensional tensor, a d-tuple of indices is generated randomly and tested if the input tensor contains a nonzero at that position. If so 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. Kolda and Hong propose a method based on sorting as a preprocess and then using binary search during query time for sampling zeros of the tensor. 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 mentioned implementation of sampling for zeros can still 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.

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 in $O(n)$ 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 necessary changes for efficiency, both in terms of run time and memory requirements, in Section 3, for adapting it to answer hyperedge queries. 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, $O(d)$ query response time 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. There are a number of recent perfect hashing methods [7, 9, 12, 13]. While these are highly efficient with practically efficient implementations, the hyperedge query problem should be addressed on its own; otherwise efficiency will be lost. We compare our approach experimentally with the current state of the art perfect hashing methods and demonstrate improvements both in the construction and the query response time. We note that there are recent work which use random hypergraph models to build variants of Cuckoo hashing methods [1, 5, 9, 14, 17, 18]. Our work is not in the same domain: we seek hashing methods for querying the existence of hyperedges in a given hypergraph.

2 Preliminaries and background

We give a brief summary of the hashing method by Fredman et al. [8]. 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.

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' 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, $\Pr(X \geq c) \leq \frac{\mathbf{E}(X)}{c}$.

Let $U = \{0, \ldots, m-1\}$ be the universe, $S \subseteq U$ with $|S| = n$ be the set to be represented, and p be a prime number greater than $m-1$. We first recall that a family of hash functions H 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 $h \in H$, which is a celebrated result [2] and can also be found in more recent treatment [15, Ch. 4].

The FKS method [8] first chooses a $k \in U$ uniformly at random. It then assigns each element x of S to a bucket B_i where $i = kx \bmod p \bmod n$. This creates n buckets. Then, for a bucket B_i containing $b_i > 0$ elements, a storage space of size b_i^2 is allocated, and a number $k^{(i)} \in U$ is chosen. Then each element x that is assigned to B_i is stored at the index $k^{(i)}x \bmod p \bmod b_i^2$ of the storage space. A first requirement is that $\sum_i b_i^2$ should be $O(n)$ so that the method uses linear space. A second requirement is that each $k^{(i)}$ should enable a perfect hashing—that is, two different items in the same bucket should have different key values. If hash functions from a 2-universal family is used, then these requirements are met.

The values k and $k^{(i)}$ 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 if this k were used in the first level hash function. If $\sum_i b_i^2 < 3n$, then k is accepted; if not, this process is repeated. For bucket B_i, a random $k^{(i)} \in U$ is chosen, and the mapping $k^{(i)}x \bmod p \bmod b_i^2$ is tested to see if it is an injection for the set of elements assigned to this bucket. If so, that $k^{(i)}$ is accepted, if not, another one is tried. Fredman et al. note that with the bound $\sum_i b_i^2 < 3n$ and the injection requirement with b_i^2 space, one may need to test many k and $k^{(i)}$. In order to obtain $O(n)$ time, in expectation, they suggest testing for $\sum_i b_i^2 < 5n$ and using $2b_i^2$ space for the bucket B_i. In these cases 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 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, $\ell = k^{(i)}x \bmod p \bmod 2b_i^2$ is computed, and the location ℓ is checked if it is equal to q. If so, $q \in S$, otherwise not. This last comparison 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 hashing function for the two levels and show that they work for our case. That is the first level hashing ensures buckets of suitable storage size, and the second level hashing ensures an injection for each bucket. We then reduce the space requirements of the proposed method.

3.1 The hash function and its properties

For d-partite d-uniform hypergraph $H = (V, E)$, the hyperedge set E is a set of d-tuples. 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 is to convert a d-tuple to a unique integer by linearizing it. 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 there are s_i is the size at dimension i, and a longer formula for higher d can be used. Afterwards, the FKS method can be used as it is. 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 using a d-tuple for finding the hash functions. Furthermore, since storing d-tuples in the buckets could render the storage requirement depend on d, we store the ids of the hyperedges.

Let $\mathbf{k}$ be a randomly chosen element of U. Let $\mathbf{x}$ denote an element of the universe. We use $\mathbf{k}^T\mathbf{x} = \sum_{i=0}^{d-1} k_i x_i$ to denote the inner product of the vectors corresponding to two d-tuples $\mathbf{k}$ and $\mathbf{x}$. Then the first level hash function is computed as $i = \mathbf{k}^T\mathbf{x} \bmod p \bmod n$, and the id of the hyperedge $\mathbf{x}$ is assigned to bucket B_i. We again use b_i to refer to the number of ids assigned to B_i. This hash function makes sure that the linear space requirement is met, as shown in the next lemma. We give the proof in Appendix A for completeness, where we also explain why the bound is $4n$, instead of $3n$.

Lemma 1. *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 $\sum_{i=0}^{n-1} b_i^2 < 4n$.*

Similar to the original FKS method, if we allow $\sum_{i=0}^{n-1} b_i^2 < 7n$, then at least half of the potential $\mathbf{k}$ tuples satisfy the bound, and a sampling approach quickly finds a suitable $\mathbf{k}$. The proof is given in Appendix A, again for completeness.

Corollary 2. *For at least half of potential $\mathbf{k}$ tuples used in the first level it holds that $\sum_i b_i^2 < 7n$.*

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 ${\mathbf{k}^{(i)}}^T\mathbf{x} \bmod p \bmod b_i^2$. Here we assume that $p \geq b_i^2$, which is justified as buckets contain only a few elements. The proof is in Appendix A for completeness.

Lemma 3. *For each bucket B_i with $b_i > 0$ elements, there is a $\mathbf{k}^{(i)}$ such that ${\mathbf{k}^{(i)}}^T\mathbf{x} \bmod p \bmod b_i^2$ is an injection for $p \geq b_i^2$.*

As also done by Fredman et al. if we allow $2b_i^2$ space for bucket B_i, then half of $\mathbf{k}^{(i)}$s define an injection by Markov inequality, and hence $\mathbf{k}^{(i)}$ can be found quickly.

Corollary 4. *For at least half of potential $\mathbf{k}^{(i)}s$, ${\mathbf{k}^{(i)}}^T\mathbf{x} \bmod p \bmod 2b_i^2$ is an injection.*

3.2 Reducing the space requirements

While the previous lemmas show that we can use the FKS method with d-tuples as $\mathbf{k}$ and $\mathbf{k}^{(i)}$s, there is a catch. For a bucket i, we need a space of size d to store $\mathbf{k}^{(i)}$. This can result in a large space utilization, since the number of empty buckets is small, as shown in the next lemma.

Lemma 5. *For a random $\mathbf{k} \neq [0, \ldots, 0]$ and a random set S of size n, there are at least $n\left(1 - e^{-1+1/p}\right)$ nonempty buckets, where e is the base of natural logarithm.*

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

We first compute the expected value over $\mathbf{k}$ and S of the random variable $R_{S,i} = |\{\mathbf{x} \in S | R_{\mathbf{x}} \neq i\}|$. For a fixed tuple $\mathbf{k} \neq [0, \ldots, 0]$, we first evaluate the number of $\mathbf{x}$ such that $R_{\mathbf{x}} = i$. If $R_{\mathbf{x}} = i$, we should have

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

This means that there are $t_i = \left\lfloor \frac{p-1-i}{n} \right\rfloor + 1$ possible values for $\mathbf{k}^T\mathbf{x} \bmod p$. First note that $\frac{n \cdot t_i}{p} \geq 1 - \frac{1}{p}$, as $p > n > i$. For any value j in the right hand size of (1), 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 $x_i^j, i \neq \ell$, there exists one unique value of x_ℓ^j 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 (1) holds, and hence for which $R_{\mathbf{x}} = i$, over all p^d tuples.

There are therefore $\binom{p^{d-1} \cdot (p-t_i)}{n}$ sets S for which $R_{\mathbf{x}} \neq i$, where $\binom{n}{f} = \frac{n!}{f!(n-f)!}$. Then, for a fixed value $\mathbf{k}$, the probability that none of the elements of an arbitrarily chosen S maps to i is

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

We can thus bound $\Pr(R_{S,i} = 0)$ as follows:

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

By defining 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

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

which concludes the proof. □

Storing a d-tuple $\mathbf{k}^{(i)}$ for all nonempty buckets will consume too much memory. Such a storage can easily dominate the memory requirements when d is large. An obvious improvement is to avoid the second level hashes for buckets with only one element; an improvement which is also described in the original FKS method.

In order to further reduce the memory and obtain a lean variant of FKS for hyperedge queries, we propose to share the second level hash functions among the buckets. We thus propose the following variant of FKS, which is explained in Figure 1. Keep the first level the same as in the original FKS method and obtain a $\mathbf{k}$ that results in a suitable bound on the memory utilization. Let K' be a set of d-tuples, initially empty. Then, for each nonempty bucket B_i with $b_i > 1$ check if any $\mathbf{k}' \in K$ defines an injection for the hyperedges assigned to B_i. If so, store a reference (pointer or index) to that $\mathbf{k}'$ in K. If none of the existing $\mathbf{k}'$ is an injection for B_i, or K is empty, choose a random $\mathbf{k}' \in U$ defining an injection for B_i by sampling and trial, add it to K, and store a reference to the new $\mathbf{k}'$. As seen in the figure, for buckets with one id, we just store the id of the hyperedge instead of a reference to a $\mathbf{k}'$ in K. For an empty bucket B_i, we store only b_i, which is 0.

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

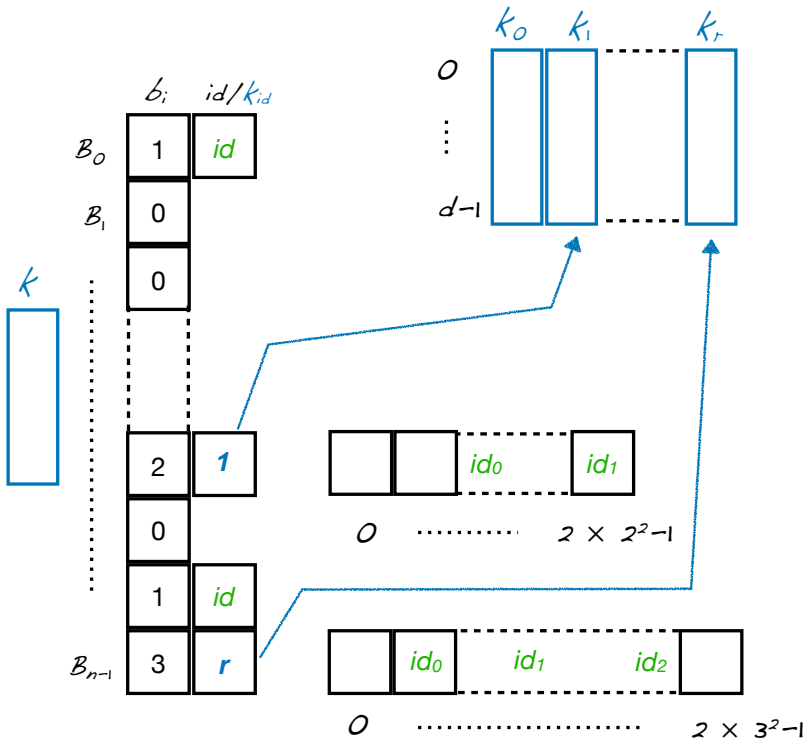


Figure 1 – In the proposed variant of FKS, for each bucket B_i, we store the number b_i of hyperedges assigned to B_i. If b_i is 0, nothing else is stored for B_i. If b_i is 1, the id of the hyperedge mapping to i is stored. If b_i is larger than 1, then the index of a suitable d-tuple from $\mathbf{k}_0$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

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 $n\left(2-e^{-1+1/p}\right)+\sum_{i,b_i>1}2b_i^2$, if we store the number of elements, the ids, and the references to the $\mathbf{k}\in K$ with the same data type, on top of the set of d-tuples stored in K. Using the bounds in Lemma 1 and Lemma 5, we see that the total space requirements is less than $(10-e^{-1+1/p})n$ in theory, independent from d. This is independent from d, if we have only a few hash functions. The following theorem shows that we indeed need a small number of d-tuples to define all hash functions.

Theorem 6. *The largest number of* $\mathbf{k}'\in K$ *is* $1+\log_2 n$ *in expectation.*

Proof. For each bucket, we know from Corollary 4 that at least half of the $\mathbf{k}^{(i)}$ are such that ${\mathbf{k}^{(i)}}^T\mathbf{x} \bmod p \bmod 2b_i^2$ is an injection. For a given bucket i, let X_i be a random variable counting the number of different trials we did to find a suitable $\mathbf{k}^{(i)}$. 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 tried t times 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 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 describes the number of $\mathbf{k}'\in K$. We will bound the expectation of R_K to obtain the bound stated in the theorem. We first note that

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

The bound obtained in (2) 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 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 (2) 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* $\mathbf{k}' \in K$ *exceeds* $t \log_2 n + 1$ *is bounded by* n^{1-t}.

One could create K with $r = 1 + \log_2 n$ tuples at the beginning, and for each bucket B_i with $b_i > 1$ can randomly sample from this set. This would result in theoretical worst-case linear time construction in expectation, that is in $O(nd)$ time. Our construction approach of incrementally building K can be pessimistically bound to be of time complexity $O(\sum db_i \log_2 n) = O(n \log_2 n)$. This is so as each trial for a bucket B_i costs $O(db_i)$. We sacrifice the theoretical guarantee for practical value, as incrementally building K could mostly result in less than $1 + \log_2 n$ elements in K.

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 $\mathbf{k}^{(i)}$'s index in K is read, 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 partition on the vertices and uniformity on the hyperedge sizes. Let $H = (V, E)$ be a hypergraph with the maximum size of an edge, called rank, r. Let us assume that 0 is not a member of V. Let us 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 one hyperedge e' corresponding to a unique hyperedge $e \in E$ of the original hypergraph. The vertices in e' are sorted, and for each part $j = |e|, \ldots, r-1$ the vertex 0 is added to e. The hypergraph H' is therefore r-uniform and r-partite. Any query q is also sorted into q', and for each missing dimension $j = |q|, \ldots, r-1$ the vertex 0 is added to q', making q' of size r as well. A query q posed on H can thus be answered equivalently by the query q' on H'.

The above transformation of padding queries and hyperedges with 0 for missing positions in the hyperedges should be done implicitly, otherwise the run time and space requirements can increase prohibitively. In particular, when processing a hyperedge h', only the original vertices should be used while computing inner products with $\mathbf{k}$ vectors. Since each hyperedge and query can be sorted in linear time (by counting sort), the construction takes linear time and requires linear space, and query response time is linear in the size of the query.

4 Experiments

We compare the proposed algorithm with state of the art methods. The first algorithm, which we call RadixSort, sorts the hyperedges in linear time using radix sort [3, Section 8.3], and then allows a binary search during query time. The query time is obviously will be larger than worst case and average case constant time algorithms. We experiment with RadixSort principally to put the construction time of different algorithms in perspective. Another alternative is to use a state-of-the-art average-case constant time hashing method, with a good hash function. As the FKS method's first level hash function has desirable property of bounding the square of the total collisions, we used the hash function $\mathbf{k}^T\mathbf{x} \bmod p \bmod n$. Since this method obtains key values à la FKS, we call it HashàlaFKS. We next turned our attention to the state of the art minimal perfect hash functions. These functions are static data structures mapping a given set S with n elements to $\{0, \ldots, n-1\}$. Therefore, they guarantee worst case constant time query time with the minimal space of size n. There are a number of publicly available implementations of minimal perfect hash functions [7, 9, 12, 13]. We use uRecSplit [7] in our experiments, as it is the most recent one and has a publicly available implementation. The proposed method is called FKSlean. Since the input is a static set, we did not compare with the variants of Cuckoo hashing [5, 9, 17, 18].

As the `unordered_map` from `C++ std` provides a standard implementation of the average-case constant time hashing method, and uRecSplit is available in `C++`, we implemented RadixSort, HashàlaFKS, and the proposed method FKSlean in `C++`. All codes are compiled with `g++ version 9.2` with `-O3` optimization flag. We carried out out experiments on a machine having Xeon(R) CPU E7-8890 v4 with a clock-speed of 2.20GHz.

4.1 Implementation and measurement details

HashàlaFKS and RadixSort are straightforward and do not need explanation. Here we give some details about our implementation of FKSlean and our use of uRecSplit.

uRecSplit accepts items as character strings. In order to use uRecSplit, we converted each hyperedge $\mathbf{x} = [x_0, \ldots, x_{d-1}]$ into a string. We first converted each x_i to a string and prefixed it with enough zeros to have $\lfloor \log_{10}(s_i) \rfloor + 1$ characters, where s_i is the number of vertices in the ith vertex part. The strings for each i are then concatenated. uRecSplit with suggested parameters are used to compute a minimal perfect hash function. A table of size n is allocated and the id of each hyperedge is stored at the position where its string representation is mapped. A query is answered by first transforming it to a string as described above, and then comparing the query with the unique hyperedge whose id is stored in the position to which the query maps.

We have decided to implement FKSlean using a vector from `std` per bucket. This has memory allocation and management overhead during construction, 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 to ease future developments regarding the dynamic case, where hyperedges come and go.

We report average of 5 runs in all measurements per hypergraph, as the queries and the hash functions for FKSlean and HashàlaFKS are randomly constructed. The construction time reported for FKSlean includes everything required: (i) finding a prime number $p > n$; (ii) finding a $\mathbf{k}$ for the first level hash satisfying a tighter version of Lemma 1, in which we check $\sum_i b_i^2 < 3n$, which turned out to hold always in all of our experiments; (iii) allocating all n buckets; (iv) testing each $\mathbf{k}' \in K$ for perfect hashing with a table of size $2b_i^2$ for each i and adding one more $\mathbf{k}'$ to K by sampling and

trial if none of the existing ones was suitable for the bucket B_i; (v) and finally storing id's of the hyperedges in suitable places in each bucket. For the query response time we report the average of five runs for answering the queries. The time to generate queries is not reported.

4.2 Data set

We present experiments both on real-life and constructed data. We use the real-life data to compare 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 the FROSTT collection [16], 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. `Delicious-3d` and `Delicious-4d` contain the same data, with different dimensions. It turns out that the hyperedges are unique without the dimension discarded from `Delicious-4d` to obtain the other one. 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 (formerly the University of Florida Sparse 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 graphs. Given a desired number of parts d, a desired number s of vertices in each part, and the 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 $s - 1$.

4.3 Comparisons

Table 2 and Table 3 display, respectively, the construction time and the query response time for all methods on the real-life instances, with 10^7 queries. In these two tables, the run time of RadixSort is given in seconds, whereas those of the other three methods are given as the ratios to the run time of RadixSort.

As seen in Table 2, RadixSort has the smallest construction overhead. Sorting in linear time of $O(nd)$ is always less time consuming than other methods; which are linear but have higher overheads. One inconvenience of RadixSort is that it modifies its input. If the caller needs to keep the hyperedge list intact, RadixSort needs to save the whole list of hyperedges. This would be time and memory consuming. The construction time of HashàlaFKS and FKSlean are always better than uRecSplit, while FKSlean's construction time is always better than that of HashàlaFKS. By looking at the geometric mean of ratios, we see that uRecSplit, HashàlaFKS, and FKSlean are slower than RadixSort by factors of 8.17, 3.38, and 2.68 respectively, on this set of hypergraphs. The pairs `Delicious-3/4d`, `Flickr-3/4d`, and `vast-2015-mc1-3/5d`, show how RadixSort's construction time increases with d for the same number of hyperedges. The construction time of the three other methods do not change as much with the increasing number d of parts.

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 × 1443	140,126,181
Flickr-3d	3	319686 × 28153045 × 1607191	112,890,310
Flickr-4d	3	319686 × 28153045 × 1607191 × 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.

name	RadixSort	uRecSplit	HashàlaFKS	FKSlean
kmer_A2a	350.44	3.33	1.66	1.15
Queen_4147	308.01	3.33	1.31	1.31
com-Orkut	126.70	5.72	2.83	2.42
Nell-1	123.28	3.74	1.68	1.42
Delicious-3d	68.03	6.50	2.88	2.30
Delicious-4d	89.76	5.00	2.32	1.84
Flickr-3d	47.18	7.54	3.45	2.86
Flickr-4d	52.21	6.92	3.17	2.62
Nell-2	25.03	9.24	4.15	3.17
Enron	13.15	12.71	5.46	4.33
vast-2015-mc1-3d	5.44	13.98	5.88	3.86
vast-2015-mc1-5d	15.24	5.29	2.37	1.49
Chicago_Crime	1.77	8.76	3.16	2.22
Uber	0.84	11.38	3.85	3.19
LBNL-network	0.22	23.24	4.94	4.72
geo-mean		8.17	3.38	2.68

Table 2 – The construction time of the four methods. That of RadixSort is given in seconds. The construction times of other methods are given as ratio to that of RadixSort. Geometric mean of the ratios of the construction times to that of RadixSort are given in the last row.

name	RadixSort	uRecSplit	HashàlaFKS	FKSlean
kmer_A2a	36.51	0.31	0.23	0.10
Queen_4147	42.63	0.25	0.18	0.09
com-Orkut	40.40	0.27	0.19	0.09
Nell-1	32.51	0.36	0.23	0.10
Delicious-3d	36.35	0.28	0.20	0.09
Delicious-4d	36.33	0.30	0.22	0.10
Flickr-3d	36.65	0.29	0.21	0.09
Flickr-4d	37.24	0.32	0.21	0.09
Nell-2	37.68	0.25	0.18	0.08
Enron	23.55	0.41	0.28	0.13
vast-2015-mc1-3d	32.82	0.25	0.19	0.09
vast-2015-mc1-5d	34.05	0.30	0.21	0.09
Chicago_Crime	27.31	0.27	0.21	0.10
Uber	25.18	0.31	0.22	0.11
LBNL-network	7.53	0.94	0.60	0.32
geo-mean		0.32	0.22	0.10

Table 3 – The response time for 10^7 queries of the four methods. That of RadixSort is given in seconds. The response times of other methods are given as ratio to that of RadixSort. Geometric mean of the ratios of the response times to that of RadixSort are given in the last row.

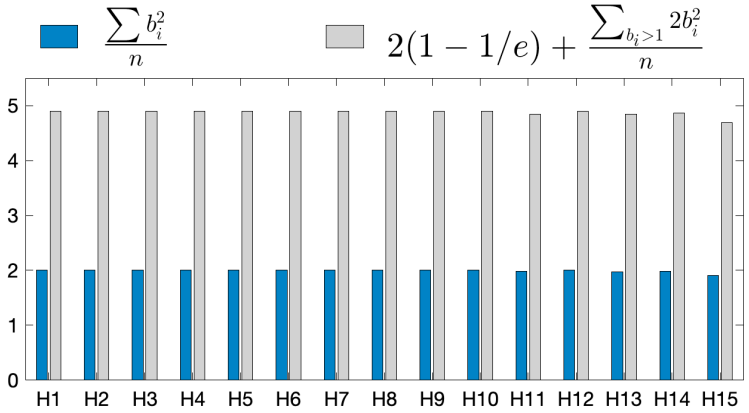
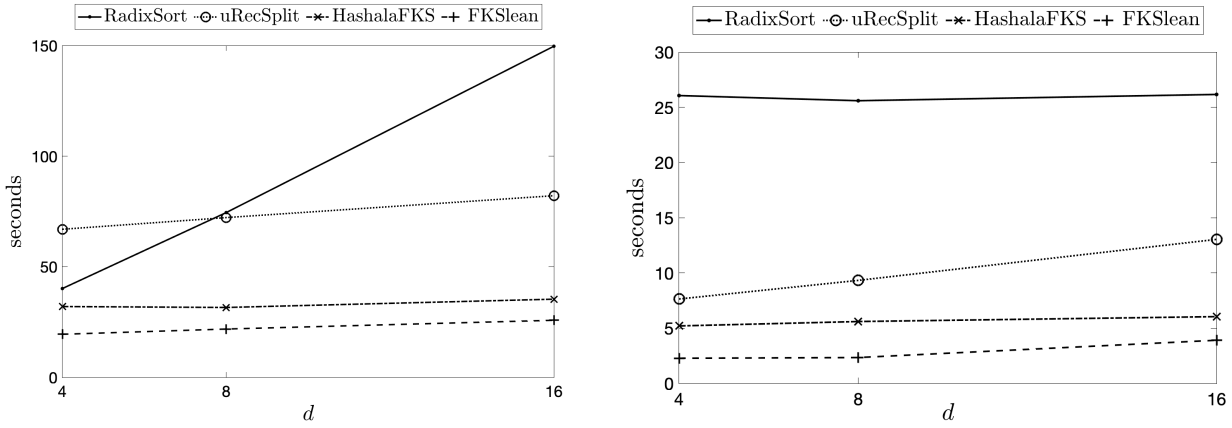


Figure 2 – The bar charts of $\frac{\sum b_i^2}{n}$ and $(2-1/e)+\sum_{i,b_i>1}\frac{2b_i^2}{n}$. The hypergraphs are named as Hi, for $i=1,\ldots,15$ and are given in the same order as Table 1.

As seen in Table 3, the query response time of RadixSort is always larger than those of others. While one expects increasing query response time for RadixSort with the increasing number of parts, the differences between the pairs `Delicious-3/4d`, `Flickr-3/4d`, and `vast-2015-mc1-5d` are not significant. We further investigated this, and saw that even with cases where all queries are positive (hence all d entries should compared), the difference were not significant. Therefore, the reason is likely to be the fact that the number of parts is small. uRecSplit is about three times faster than RadixSort in average, as we see in the last row, while HashàlaFKS and FKSlean are nearly five and 10 times faster, respectively. It is also interesting to note that FKSlean is always faster than HashàlaFKS (on average twice). This shows that the average-case constant time algorithm has higher overhead.

We now look at the space utilization of FKSlean, and see how the theoretical properties shown in Lemma 1, Lemma 5, and Theorem 6 compare with practical results. We start with the quantity $\sum b_i^2$ that we bound as $4n$ in Lemma 1, and the total number of space used by FKSlean, except $\mathbf{k}$ and K, which we stated as $n(2-1/e)+\sum_{i,b_i>1}2b_i^2$. In the bar chart shown in Figure 2, we see that $\sum b_i^2$ is always close to $2n$ in these experiments. This also translates to a bound less than $5n$ for $n\left(2-e^{-1+1/p}\right)+\sum_{i,b_i>1}2b_i^2$. The largest value of $\sum b_i^2$ is $2.0038n$, obtained for the second hypergraph (`Queen_4147`), and the smallest value was $1.9037n$, obtained for the last hypergraph (`LBNL-network`). The largest and smallest values of $n(2-1/e)+\sum_{i,b_i>1}2b_i^2$ are $4.9042n$ and $4.6945n$ and are obtained for the same hypergraphs H2 and H15. This observation prompts us to look for a tighter analysis of Lemma 1. We next look at the number of nonempty buckets, which we bound as larger than $n\left(1-e^{-1+1/p}\right)$ in Lemma 5. In all instances, the number of nonempty buckets is observed to be between $0.6315n$ and $0.6484n$; this is larger than $n(1-1/e)$, which is almost equal to the stated bound when p is large. As seen from these results, the number of nonempty bounds is concentrated around the expected value. Lastly, we look at the number r of $\mathbf{k}'\in K$, 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.

We next compare all methods on random hypergraphs $\mathcal{R}(d,s,n)$ with the parameters $d=\{4,8,16\}$, $s=10^6$, and $n=2\times 10^7$. The construction times of the four methods are plotted in



(a) Construction time

(b) Query response time

Figure 3 – The construction time and the response time for 8×10^6 queries, in seconds, for the four 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 $d = 16$, the plots correspond to, from top to bottom, to the methods as listed in the legend.

Figure 3a. As seen in this figure, at this scale, FKSlean has the fastest construction time. RadixSort starts close to HashàlaFKS, but after $d = 8$ becomes the method with the slowest construction time. This is so, since it reorganizes the hyperedges into a sorted order with d passes over the data (as the standard radix-sort method). The response time for 8×10^6 queries is plotted in Figure 3b, and are in line with the results on the real-life instances. We see in Figure 3b that uRecSplit's and FKSlean's query response times are more sensitive to d than others, while the increases are not proportional to the increase in d.

The `unordered_map` from `C++ std` reports the load factor `LF`, which is the number of element 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 real-life instances `LF` was in between 0.9268 and 0.9971, and similarly with the random hypergraphs, it was in between 0.9370 and 0.9984. HashàlaFKS's is therefore space efficient, storing around $1.08n$ ids.

4.4 Summary of comparisons

FKSlean's construction time is larger than that of RadixSort by a factor of 2.68 for the instances in the real-life data set. Nonetheless it allows 10 times faster query response time in the same data set. HashàlaFKS and uRecSplit are slower than FKSlean in both aspects of the run time.

We have seen that when the number of parts and the number of queries are small, RadixSort is preferable, as the gains in the construction time would not be annihilated by the losses in the query response time. It comes with an additional advantage of working only with the hyperedge set (although RadixSort modifies the storage order of the hyperedges). HashàlaFKS could also be useful because of its simplicity. One just needs to implment methods to find a prime number $p > n$, and to compute $\mathbf{k}^T\mathbf{x} \bmod p \bmod m$. The `unordered_map` from `C++ std` efficiently takes care all of the rest. For large d, n, or q, the method of choice is FKSlean, as it is the method with the fastest construction and query times, and has provably small memory requirement.

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 perfect hashing approach [8]. We analyzed the space requirement of the proposed method and showed a theoretical upper bound of around $(10 - 1/e)n$, independent from d. Experimental results demonstrated in practice that the space requirement is in fact less than $5n$. We also compared the proposed method with three other alternatives. The first one, RadixSort, sorts the hyperedges once as a preprocess and then uses binary search to answer queries. The second one, uRecSplit, uses a minimal perfect hashing method to have a minimal storage requirement with worst-case constant time query response time. The third one, HashàlaFKS, is the `unordered_map` from `C++ std` with the hash function used by FKSlean. Experiments on real-life data showed that for small d, RadixSort has the best construction time, and FKSlean is the second best, while FKSlean has the best query response time. Further experiments on a random family of hypergraphs showed that for large d and n, the construction time of RadixSort is higher than that of others, while FKSlean becomes the fastest method for both aspects of the run time.

We have three 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 $\mathbf{k}$ for 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?

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. SIAM, 2020.

[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 *16th International Symposium on Experimental Algorithms*, pages 1–11, London, United Kingdom, 2017.

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

[16] 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.

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

[18] 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 repeat the body of the lemmas and their corollaries from Section 3.1 and give their proofs.

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**: 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 $\sum_i b_i^2 < 4n$.

Proof. For a given $\mathbf{k}$, let $b_i^{(\mathbf{k})}$ denote the number of elements of S 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 S with the same hash value i is therefore $\frac{b_i^{(\mathbf{k})}(b_i^{(\mathbf{k})}-1)}{2}$. 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 S with the same hash value over the potential $\mathbf{k}$ tuples. As there is at least one $\mathbf{k}'$ for which $\sum_{i=0}^{n-1}\frac{b_i^{(\mathbf{k})}(b_i^{(\mathbf{k})}-1)}{2}$ is no larger than the average, we will use that $\mathbf{k}'$ to show the bound stated in the lemma.

We first observe that

$$\sum_{\mathbf{k}\in U}\sum_{i=0}^{n-1}\frac{b_i^{(\mathbf{k})}(b_i^{(\mathbf{k})}-1)}{2} = \sum_{\substack{\mathbf{x},\mathbf{y}\in S\\ \mathbf{x}\neq\mathbf{y}}}|\{\mathbf{k}\in U : (\mathbf{k}^T\mathbf{x} \bmod p) \bmod n = (\mathbf{k}^T\mathbf{y} \bmod p) \bmod n\}| \,, \tag{3}$$

as the right hand side counts the total number of times any pair $\mathbf{x}, \mathbf{y} \in S$ with $\mathbf{x} \neq \mathbf{y}$ give the same value for any $\mathbf{k}$.

We will bound the right hand side of (3) from above. For this, we need a bound on the number of different $\mathbf{k}$ for which any $\mathbf{x}, \mathbf{y} \in S$ give the same value

$$(\mathbf{k}^T\mathbf{x} \bmod p) \bmod n = (\mathbf{k}^T\mathbf{y} \bmod p) \bmod n \,. \tag{4}$$

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

$$(\mathbf{k}^T\mathbf{x} - \mathbf{k}^T\mathbf{y}) \bmod p \in \left\{0, \pm n, \pm 2n, \ldots, \pm\left\lfloor\frac{p-1}{n}\right\rfloor n\right\} \,. \tag{5}$$

This general formula applies to any $p > m$. In the proposed algorithm, as in the original FKS methods, p is a prime larger than m, which we find by testing the numbers $m+1, m+2, \ldots$ for primality and accept the first prime as p. By Bertrand's postulate [10, p. 455] there is a prime number p with $m < p < 2m$ for any $m > 3$. Therefore the set on the right hand side contains the three numbers $\{0, \pm n\} \equiv \{0, p-n, n\}$.

Since $\mathbf{x} \neq \mathbf{y}$, there is at least one index $0 \leq \ell < d-1$ such that $x_\ell \neq y_\ell$. Let us arbitrarily pick one such ℓ and write

$$\left(k_\ell(x_\ell - y_\ell) + \sum_{j\neq\ell} k_j(x_j - y_j)\right) \bmod p \in \{0, p-n, n\} \,. \tag{6}$$

Since we treat each pair $\mathbf{x}, \mathbf{y}$ once, let us assume $x_\ell > y_\ell$. 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 $\mathbf{k}$ making this equality hold. Then, multiplying with the number of elements in the right hand size, 3, will give the total number of times any pair $\mathbf{x}, \mathbf{y} \in S$ satisfy (4). From (6), we see that we are free to choose k_j for $j \neq \ell$ and need to set the value of k_ℓ to make the equation hold; this k_ℓ is unique as p is prime. In other words, for any value in the right hand side of (5), 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 (5), and for each one we have at most p^{d-1} different $\mathbf{k}$ tuples satisfying (4), we obtain an upper bound on the right hand side of (3) 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 (3), 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 $\mathbf{k}$, one of them should result in a sum no larger than the average

$$\sum_{i=0}^{n-1} \frac{b_i^{(\mathbf{k})}(b_i^{(\mathbf{k})} - 1)}{2} \leq 3\frac{n}{p}\frac{n-1}{2} . \tag{7}$$

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 method, we have scalar values. Therefore the equivalent of (5) is

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

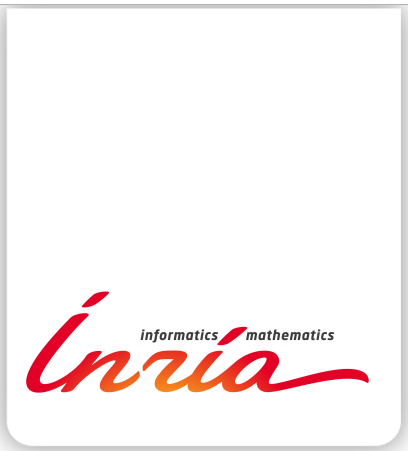
Since p is prime and both k and $x-y$ are less than p, the equality $k(x-y) \mod p = 0$ cannot hold and the set on the right hand side is $\{\pm n, \pm 2n, \ldots, \pm \left\lfloor \frac{p-1}{n} \right\rfloor n\}$, or $\{n, p-n\}$, when $m < p < 2m$.

► **Restatement of Corollary 2**: For at least half of the potential $\mathbf{k}$ used in the first level it holds that $\sum_i b_i^2 < 7n$.

Proof. Let X be the random variable denoting $\sum_i \frac{b_i(b_i-1)}{2}$. Then, by (7), we have $\mathbf{E}(X) \leq 3\frac{n}{p}\frac{n-1}{2} \leq 3\frac{(n-1)}{2}$. By Markov identity, $\Pr(X \geq 3(n-1)) \leq \frac{\mathbf{E}(X)}{3(n-1)}$, and hence $\Pr(X \geq 3n) \leq \frac{1}{2}$. On average, $X < 3n$ for at least half of $\mathbf{k}$. Multiplying X by 2 and adding another n, as at the end of the proof of Lemma 1, finishes the proof. □

▶ **Restatement of Lemma 3**: For each bucket B_i with $b_i > 0$ elements, there is a $\mathbf{k}^{(i)}$ such that ${\mathbf{k}^{(i)}}^T\mathbf{x} \bmod p \bmod b_i^2$ is an injection for $p \geq b_i^2$.

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 t, 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. And hence, $\mathbf{k}^{(i)}$ can be chosen. We needed $p \geq b_i^2$ because of the potential $t = 0$ in (5) adding the term 1 in $2\left\lfloor\frac{p-1}{b_i^2}\right\rfloor + 1$. □



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