Algorithms and data structure for hyperedge queries
M1 internship report

Jules Bertrand
ENS de Lyon

Supervised by:
Bora Uçar and Fanny Dufossé
Team ROMA, LIP, Lyon, France

Contents

Introduction **2**

1 Background and notation **3**

2 Algorithms and data structures **4**
2.1 Usual sparse matrix storage and extensions to hypergraphs 5
2.2 Hashing hyperedges 6

3 A data structure with a $\mathcal{O}(D)$ time for query response **8**
3.1 Preliminary results 9
3.2 Algorithm and example 12
3.3 Time and space complexity 13

4 Experiments **14**
4.1 Construction time 14
4.2 Query answer time 16
4.3 Size of data structure 18

5 Conclusion **18**

Introduction

We consider the problem of querying the existence of hyperedges in hypergraphs. More formally, let $H = (V, E)$ be a hypergraph, where V is the set of vertices, and E is the set of hyperedges. The problem is to answer a set of queries of the form "is $h \subseteq V$ a member of E ?" one by one. We are interested in data structures and algorithms enabling constant time per query in the worst-case. We will mostly deal with D-uniform D-partite hypergraphs, where the vertex set is the union of D disjoint sets $V = \bigcup_{i=1}^{D} V_i$, and each hyperedge has exactly one vertex from each part V_i.

In the special case where $D = 2$, the queries ask for existence of edges in a graph or for a non-zero coefficient in a matrix.

The difficulty of this problem is the following : our structure has to have a construction complexity and a size in memory which doesn't grow quickly with the number of elements in the universe. In this way, this structure may be useful to work on sparse matrices or graphs and hypergraphs with a high number of vertices.

The problem arises in the context of sparse tensor decomposition [KH19]. Kolda and Hong investigate a stochastic, iterative method for enabling the decomposition of dense and sparse tensors. For the sparse case, they propose a sampling method in which the non-zeros and zeros of a sparse tensor are sampled separately for accelerating the convergence of the stochastic decomposition method. This method is called stratified sampling and works as follows. The non-zeros of the input tensor sampled uniformly at random. For sampling zeros in a D dimensional tensor, a set of positions $(i_1, \ldots, i_D)$ are created and checked if the given tensor contains a non-zero at that position, if so the index is rejected and a new one is sampled, until a desired number of zeros are sampled. Sampling non-zeros is straightforward, as the non-zeros of a tensor are available in a list. On the other hand testing if a given position is non-zero in a tensor is time consuming. That is why Kolda and Hong propose and investigate other approaches, where the stratified sampling method is shown experimentally to be the most useful one.

During my internship, I began by reviewing the existing approaches, starting by the sort based ones, as used by Kolda and Hong [KH19]. Other solutions, mainly using hashing, allow a $\mathcal{O}(D)$ time for query response but a non-zero false-positive probability. In other words, the data structure may answer that an element is in the set when it isn't, but won't make any mistakes when dealing with elements that are really in the set. I have then tried to adapt a method [FKS84] used to store a set of bounded integers. I compared it to the sort based approaches to show that the gain in time complexity is measurable with hypergraphs and graphs of realistic size.

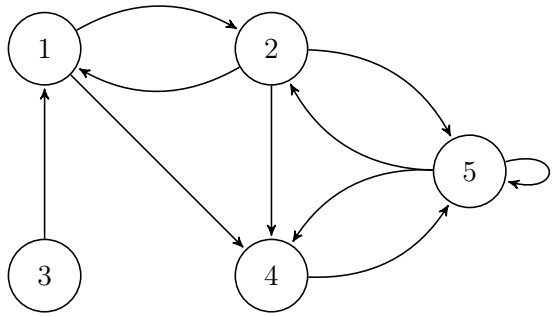
1 Background and notation

Graphs and hypergraphs are data structures frequently used in computer science. They can be used to describe constraints, to represent coalitions in game theory or to modelize a set with an associated relation.

Graph can be considered as a set V of vertices and a set $E \subset V^2$ of edges while a hypergraph considers $E \subset \mathcal{P}(V)$. Therefore, graphs can be considered as a specific case of hypergraphs.

A hypergraph is said finite if V is finite and therefore E is finite. We worked only on finite graphs and finite hypergraphs, as they both can be stored naively as their set of edges.

The easiest way to store a graph is to use its adjacency matrix. It is a matrix M of size $|V| \times |V|$ such that $M_{i,j} = 1$ if $(i, j) \in E$ and $M_{i,j} = 0$ otherwise.



$$\begin{pmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \end{pmatrix}$$

Figure 1: A graph and its adjacency matrix.

Such a representation can hardly be used to store a hypergraph because it require to be able to store a set of size $|V|^{|V|}$ which grows too quickly.

Therefore why we usually use sample matrices. For a given hypergraph, its sample matrix M is defined as follows : each row corresponds to a vertex and each column corresponds to a hyperedge. $M_{i,j} = 1$ if j is a vertex of the i-th hyperedge and $M_{i,j} = 0$ otherwise. One can remark that it is the representation of the bipartite graph built by the vertex and the hyperedges with the belonging relation. This bipartite graph is known as the incidence graph of the hypergraph.

In this report, we will consider only D-partite hypergraphs. These hypergraphs verify this property : V can be partitioned in D sets such that each vertex of E has a unique intersection with each of the elements of the partition. The hypergraph H can be written as $(\bigcup_{d=1}^{D} V_d, E \subseteq V_1 \times V_2 \times \ldots \times V_d)$ with $V_i \bigcap V_j = \varnothing$ if and only if $i \neq j$.

To each hypergraph, it is possible to associate a D-partite hypergraphs, with D being the size of the biggest hyperedge, if we have an order on the vertices of V.

Given $H = (V, E)$, we build D copies V_i^* of $V \bigcup \{\omega\}$. We consider ω bigger than any element of V. $V^* = \bigcup_{d=1}^{D} V_d$ and for each hyperedge e in E, we build a corresponding $e^* \in V^*$ by sorting the elements of e and adding ω at the end to keep e^* sorted and of size D. E^* is the set of such e^*.

The hypergraphs $H^* = (V^*, E^*)$ is a D-partite hypergraphs.

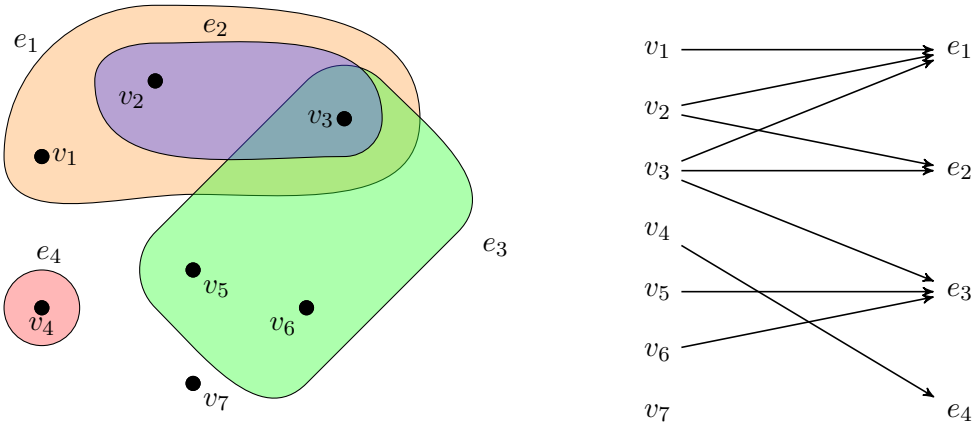


Figure 2: A hypergraph and its incidence graph.

We can associate a D-partite hypergraph $H = (\bigcup_{d=1}^{D} V_d, E)$ with a given D dimensional tensor T. In this hypergraph, there is a unique hyperedge for each non-zero of T, where the vertices of the hyperedge are the coordinates of the position of the corresponding non-zero. With this correspondence, sampling a non-zero from T corresponds to sampling a hyperedge from H. Sampling a zero corresponds to creating a random hyperedge by choosing a random vertex from each vertex part and testing if that hyperedge exists in E.

Because of the direct link between tensors and hypergraphs and the one between matrices and graphs, we can consider one or the other without any problem.

Our aim in this report is to design an $\mathcal{O}\left(\sum_{e\in E}|h| + \sum_{d=1}^{D}|V_d|\right)$ space structure that allows $\mathcal{O}(D)$ time for query response in the worst case. In other words, the goal is to find a structure with an efficient time for query response with a memory storage linear in the number of hyperedges.

In all our analysis of complexity, we will consider that both the access to an integer in memory and usual integer operations can be performed in $\mathcal{O}(1)$.

2 Algorithms and data structures

When one want to use an algorithm which needs the answer to a similar query many times, precomputing may be a way to improve the execution time. It consists in modifying the way to store the concerned data in order to accelerate the answers to a specific query. It is often used to compute complex functions, such as trigonometric functions or logarithmic ones, in order to avoid spending time to approximate them with the necessary precision during the execution of the program. But it can be used in many context such as finding the Lowest Common Ancestor of two vertices in a tree. Without any precomputation, such a query has a complexity linear in the height of the tree. With just a computation of complexity linear in the number of vertices, it is possible to answer to these queries in

constant time.

We want to study data structures and linked algorithms that allow efficient memory space and low time for query response in the case we study. We will try to use precomputation in order to be able to answer our queries quickly.

2.1 Usual sparse matrix storage and extensions to hypergraphs

Graphs and therefore matrices are a simple sub-case of hypergraphs. One can try to extend methods developed to store sparse matrices to the storage of hypergraphs. In such matrices the number of non-zero coefficients is way smaller than the number of zero ones. That's why a naive implementation, storing the value of each coefficient, although allowing a $\mathcal{O}(1)$ time for query response, is a very bad solution regarding the memory used. This is even quickly unusable in practice as the memory used grows quadratically with the number of vertices.

In the case of a sparse matrix, storing only the non-zero coefficients is a key-principle. Different data structures can be used and require less memory compared to the full-matrix approach. But accessing individual elements becomes slower. There exists a lot of different way to compress a sparse matrix, depending on how it will be used. Some of them enable quick modifications such as *List of lists*, where each sub-list matches one row, or the *Coordinate format* in which each non-zero coefficient is stored as a pair (row, column).

Matlab, a software used for numeric scientific computation, proposes an other format, the *Compressed storage by columns* one [GMS92].

The non-zero elements are stored in a one-dimensional array in column-major order. A second array of integers stores the row indices of these values. The last array stores one element per column in the matrix and encodes the index in the two previous arrays where the given column begins.

There exists a similar format named *Compressed storage by rows* where the rows and the columns have reverse roles. Depending on the shape of the matrix, one may be preferable to the other, as the storage space required depends on the number of columns or rows.

The operations we may want to apply to the matrix might make one preferable, for example, row slicing is way faster on matrix stored on compressed storage by rows.

However, the choice to use such a format to store sparse matrices in Matlab is due to the efficiency of usual matrix operations when such a compression is used.

As matrices can be seen as a set of pairs, hypergraphs can be seen as a set of tuples. Therefore a data structure made for D-uniform hypergraphs could easily be applied to matrices. Because of the similarity between matrices and hypergraphs, one may want to use the formats detailed previously to store hyperedges. But when we want to adapt one of these formats, the number of nested lists grows quickly with the dimension of the concerned hypergraphs.

Therefore, we chose to privilege the *Coordinate format*, which allows us to store hypergraphs as a list of D-tuple. Once sorted, it enables a time complexity to answer a query of

	Worst case time complexity				Size of the format in memory
	Construction	Addition	Multiplication	Access to an element	
Full-matrix storage	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^3)$	$\mathcal{O}(1)$	n^2
List of lists	$\mathcal{O}(nz\log(nz)+n)$	$\mathcal{O}(nz+n)$	$\mathcal{O}(n^2\times nz\log(nz))$	$\mathcal{O}(\log(nz))$	$2nz+n$
Coordinate format	$\mathcal{O}(nz\log(nz))$	$\mathcal{O}(nz)$	$\mathcal{O}(n^2\times nz\log(nz))$	$\mathcal{O}(\log(nz))$	$3nz$
Compressed storage by rows	$\mathcal{O}(nz\log(nz))$	$\mathcal{O}(nz)$	$\mathcal{O}(n\times nz)$	$\mathcal{O}(\log(nz))$	$2nz+n+1$

Table 1: Performance of different storage format of sparse matrix. The matrices are considered of size $n \times n$ with nz non-zeros elements.

$\mathcal{O}(D\log(n))$ with n being the number of hyperedges. There exists other formats but the coordinate one is the most used and answers quickly to our queries [BK08]

Although all these data structures allows us to use as few memory as possible and to compute quickly classic matrix operations, the time complexity to answer our queries isn't better as logarithmic in the number of hyperedges in our hypergraphs because they have been built to answer other queries.

2.2 Hashing hyperedges

As usual sparse matrix and tensor storage gave unsatisfying results, we decided to look for data structures build to answer quickly a more global query : "is x a member of the set S". Some of the methods used to built such structures could be adapted to show good results in our specific case, the belonging of an hyperedge to a hypergraph.

Our research showed us that lots of data structures designed to answer these queries were probabilistic. These data structures allowed a non-zero false-positive probability. In other words, the data structure will consider that some elements are in the set when they are actually not. But the contrary will never append.

Our problem is now to store n elements of a subset S' of S.

We studied at first the Bloom filters [Blo70]. Such a data structure require k hash functions $(h_i)_{1\leqslant i\leqslant k} : S \to [\![0, m-1]\!]$. They are supposed to be perfect hashing functions. In other words, if x follows a uniform law on S, for any i in $[\![1, k]\!]$, $h_i(x)$ follows a uniform law on $[\![0, m-1]\!]$.

Using a bit-array of size m, for each elements x of the subsets S', we modify the bit index by $\{h_1(x), \ldots, h_k(x)\}$ to 1. If we want to know if an element of S is in S', we compute $h_1(x), \ldots, h_k(x)$ and check if all the corresponding bits equals 1. If it is case, x is

considered as an element of S'. We can notice that an element x' which doesn't belong to S' can be considered as in the set while not being hashed exactly as an element in S' to .

For a given n and m, if we want to minimize the false positive rate, we have to choose k as close as possible to $\frac{m}{n}ln(2)$. If k is smaller, the false-positive probability will grow because of the unreliability of the membership test as few bits are checked. If k is bigger, the false-positive rate will grow because of an high number of bits set to 1.

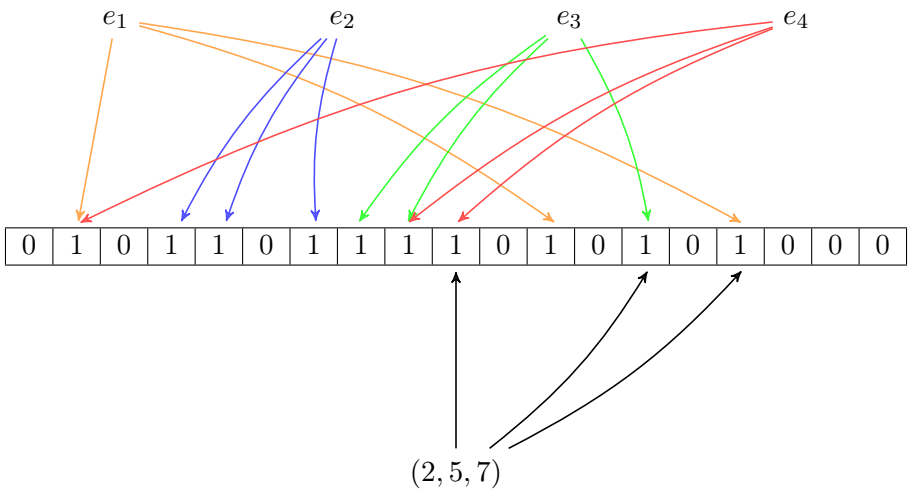


Figure 3: Example of a Bloom filter applied to the hypergraph of figure 2.

In this example, the hash functions used are the dot product of the sorted hyperedges by $(2, 3, 4), (5, 5, 5)$ and $(7, 9, 3)$ quotient by 19. $(2, 5, 7)$ is the example of an hyperedge considered in the hypergraph by the Bloom filter while it is not.

The other alternative was the cuckoo filters which are based on hashing of the same name [FAKM14]. The purpose is to avoid collisions by computing different indices where we can store an hash of the element. In that way, we will be able to move this element if we want to store an other element in that place.

Usually, cuckoo filters require two perfect hash functions, one returning an index $hash$ and the other one returning a fingerprint f of the element hashed. The two indices i, j where one can store the fingerprint are defined by hash(x) and $hash(x) \oplus f(x)$. When one want to add an element to the filter, one will compute the fingerprint of the element and its indices. If one of them is free, the fingerprint may be stored in the available place. If it is not the case, one of the already stored fingerprint can be moved to its other index in order to free space for the new element. It is possible because one is able to find the other index, if one know one index and the fingerprint stored. An insertion may fail because the array is too full and the replacement of a fingerprint create a loop.

One may want to use another involutory function than XOR to determine the second index. It is possible to replace the computation of the 2 indices by k indices if we want to use a function such that $f^k = id$.

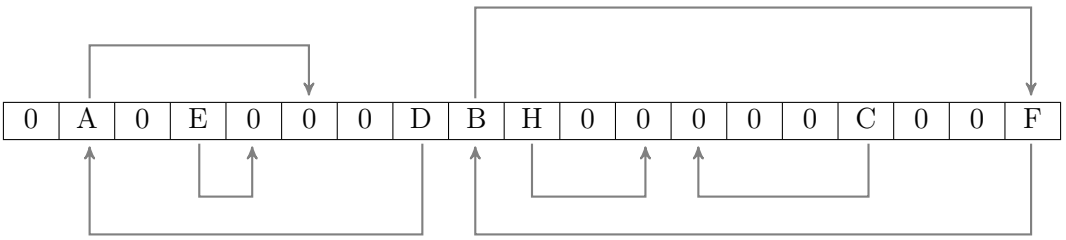


Figure 4: Example of a cuckoo hashing.

The arrows of figure 4 symbolize the alternative index where each element can be stored. If we want to add an element which has 7 and 8 as possible indices, we can move D and A in order to free the 7-th index. But if we want to insert the element in the 8-th cell, it will fail as the possible indices for B and F are creating a loop.

In order to have an idea of the performance of such data structures, one may want to use them on big instances. For a false positive probability of 0.19%, a Bloom filter uses 13 bits per element in the set, while a cuckoo filter uses 12.6 bits per element but is slower to answer the queries [FAKM14].

As we consider that usual operations on integers have a complexity of $\mathcal{O}(1)$, computing the fingerprint and the indices of an hyperedge can be performed in $\mathcal{O}(D)$. As an insertion may fail, it is sometimes required to rehash the cuckoo filter but the amortized insertion complexity remains $\mathcal{O}(D)$. Adding an element in a Bloom filter can be performed in $\mathcal{O}(D)$ for the same reasons. Building a set of n elements using one of these structures can be performed in $\mathcal{O}(Dn)$

As answering a query in both structures only requires computation of hash functions and access to indices, it can be performed in $\mathcal{O}(D)$.

Despite the problem to find perfect hash functions on a set of hyperedges, these structures have a major issue [RK12]. It is due to the size of the universe in comparison with the considered set. Even for a small false positive probability, a main part of the elements considered as in the set are actually not.

We can consider an example to understand better the problem. For a 3-partite hypergraph with 1000 vertices in each subset of V and 1000000 hyperedges with a false positive probability of only 0.1%, in average there are 999000 false positive. The probability for a positive query to be an actual hyperedge of the hypergraph is around $\frac{1}{2}$.

So theses data structure will not be able to represent correctly a hypergraph even if we gain a $\mathcal{O}(D)$ complexity time to answer a query. It would require a big amount of memory to have a reliable filter on hypergraphs.

3 A data structure with a $\mathcal{O}(D)$ time for query response

After analysing classics matrices storage and probabilistic methods to represent a set, we studied a method built on integer sets [FKS84]. Considering n integers of $[\![1, m]\!]$, this

data structure allows us to compute in $\mathcal{O}(1)$ if an element is in the set with a table of size linear in n.

This method use two layers of hashing. The first layer is made to construct small subsets of the set. For each of this subset, the second layer is an injection built in order to avoid collisions. The complexity of the construction of the table is $\mathcal{O}(n)$.

A naive generalization of this method can be used on graphs and hypergraphs but it require to use large numbers. It is based on a bijection between the set of the hyperedges and the integers. We work on a generalization which avoid such a problem in order to improve the construction time, the query answer time and the memory used.

3.1 Preliminary results

Our algorithm uses two layers of hashing in order to have an perfect hashing on the hyperedges of the set. In other words, we are not avoiding collisions between a hyperedge of the hypergraph and one which is not.

Each layer is built using a probabilistic method. We prove in this section two properties which assure us that the two layers are constructible.

We work in the universe $U = [\![0, n-1]\!]^d$, the set of d-tuples with each element of the tuple being an integer between 0 and $n-1$. We want to store W, a subset of the universe, where $|W| = N$. Let p be a prime number larger than N and n. We recall that as p is prime, every element of $\mathbb{Z}/p\mathbb{Z}$ has a multiplicative inverse. Let U' be an enlarged universe, $U' = [\![0, p-1]\!]^d \backslash (0, \ldots, 0)$.

Theorem 1.
Let $B(s, W, \mathbf{k}, j) = |\{\mathbf{x} \in W | (\mathbf{k}^T \mathbf{x}) \mod p \mod s = j\}|$.
We have the following inequality :

$$\sum_{\mathbf{k} \in U'} \sum_{j=0}^{s-1} \binom{B(s, W, \mathbf{k}, j)}{2} < \frac{|U'| N^2}{s} .$$

If we consider a hash function which associate $(\sum_{i=1}^{d} k_i \times x_i \mod p) \mod s$ to a hyperedge $(x_1, \ldots, x_d)$, $B(s, W, \mathbf{k}, j)$ is the number of hyperedges of the hypergraph such that our hash function characterized by $\mathbf{k}$ send them on j. Since there is no collision for single elements, $\max\{0, B(s, W, \mathbf{k}, j) - 1\}$ gives the number of collisions for j. Note that the bigger the value of $B(s, W, \mathbf{k}, j)$, the more collisions there are for j.
The total number of collisions is thus equal to $\sum_{j=0}^{s-1} \max\{0, B(s, W, \mathbf{k}, j) - 1\}$.

Proof. For a given $\mathbf{k}$ and a given j, $\binom{B(s,W,\mathbf{k},j)}{2}$ is the number of two element subsets $\{\mathbf{x}, \mathbf{y}\}$ of W such that

$$j = (\mathbf{k}^T \mathbf{x} \mod p) \mod s = (\mathbf{k}^T \mathbf{y} \mod p) \mod s .$$

Therefore $\sum_{j=0}^{s-1}\binom{B(s,W,\mathbf{k},j)}{2}$ is the number of two element sets $\{\mathbf{x},\mathbf{y}\}$ such that

$$(\mathbf{k}^T\mathbf{x} \mod p) \mod s = (\mathbf{k}^T\mathbf{y} \mod p) \mod s\ .$$

Therefore, $\sum_{\mathbf{k}\in U'}\sum_{j=0}^{s-1}\binom{B(s,W,\mathbf{k},j)}{2}$ is the number of pairs $(\mathbf{k},\{\mathbf{x},\mathbf{y}\})$ such that :

$$(\mathbf{k}^T\mathbf{x} \mod p) \mod s = (\mathbf{k}^T\mathbf{y} \mod p) \mod s\ .$$

We can rewrite

$$\sum_{\mathbf{k}\in U'}\sum_{j=0}^{s-1}\binom{B(s,W,\mathbf{k},j)}{2} = \sum_{\substack{\{\mathbf{x},\mathbf{y}\}\\ \mathbf{x}\neq\mathbf{y}}} |\{\mathbf{k} : (\mathbf{k}^T\mathbf{x} \mod p) \mod s = (\mathbf{k}^T\mathbf{y} \mod p) \mod s\}|\ ,$$

since the right hand side formula counts the total number of times any two elements $\mathbf{x}$ and $\mathbf{y}$ of W give the same value $((\mathbf{k}^T\mathbf{x} \mod p) \mod s)$ for different $\mathbf{k}$.

To compute this number, we want to find, for a given $\{\mathbf{x},\mathbf{y}\}, \mathbf{x}\neq\mathbf{y}$, the number of non-zero $\mathbf{k}$ such that

$$(\mathbf{k}^T\mathbf{x} \mod p) \mod s = (\mathbf{k}^T\mathbf{y} \mod p) \mod s\ .$$

If $(\mathbf{k}^T\mathbf{x} \mod p) \mod s = (\mathbf{k}^T\mathbf{y} \mod p) \mod s$, we have:

$$\mathbf{k}^T(\mathbf{x}-\mathbf{y}) \mod p \in \{s, 2s, 3s, \ldots, \left\lfloor\frac{p-1}{s}\right\rfloor s, p-s, p-2s, p-3s, \ldots, p-\left\lfloor\frac{p-1}{s}\right\rfloor s\}\ .$$

Let t be an element of $\{s, 2s, 3s, \ldots, \left\lfloor\frac{p-1}{s}\right\rfloor s, p-s, p-2s, p-3s, \ldots, p-\left\lfloor\frac{p-1}{s}\right\rfloor s\}$.
We want to compute the number of $\mathbf{k}$ such that $\mathbf{k}^T(\mathbf{x}-\mathbf{y}) \mod p = t$.
As $(\mathbf{x}-\mathbf{y})$ is non-zero, one of its coordinates i is non-zero.

$$\mathbf{k}^T(\mathbf{x}-\mathbf{y}) = k_i \times (x_i - y_i) + \sum_{\substack{j=0\\ j\neq i}}^{d-1} k_j \times (x_j - y_j).$$

If $k_i = (t - \sum_{\substack{j=0\\ j\neq i}}^{d-1} k_j \times (x_j - y_j))(x_i - y_i)^{-1}$, we have $\mathbf{k}^T(\mathbf{x}-\mathbf{y}) \mod p = t$.
No other value of k_i give the same result as $f : k_i \mapsto k_i \times (x_i - y_i) + z$ in $\mathbb{Z}/p\mathbb{Z}$ is a bijection. It is due to the existence of an inverse for each element in $\mathbb{Z}/p\mathbb{Z}$.
For all the p^{d-1} possible values for the $k_j, j\neq i$, there exists a unique k_i such that $\mathbf{k}^T(\mathbf{x}-\mathbf{y}) \mod p = t$.

Therefore, for all $\{\mathbf{x},\mathbf{y}\}$, there are p^{d-1} values of $\mathbf{k}$ such that $\mathbf{k}^T(\mathbf{x}-\mathbf{y}) \mod p = t$.

$$|\{s, 2s, 3s, \ldots, \left\lfloor\frac{p-1}{s}\right\rfloor s, p-s, p-2s, p-3s, \ldots, p-\left\lfloor\frac{p-1}{s}\right\rfloor s\}| \leqslant \frac{2(p-1)}{s}\ .$$

There is at most $\frac{2(p-1)}{s}$ possible values for t .
Therefore, for a given $\{\mathbf{x},\mathbf{y}\}, \mathbf{x} \neq \mathbf{y}$, there is at most $\frac{2p^{d-1}(p-1)}{s}$ possible $\mathbf{k}$ such that

$$(\mathbf{k}^T\mathbf{x} \mod p) \mod s = (\mathbf{k}^T\mathbf{y} \mod p) \mod s \ .$$

If we sum on the $\binom{N}{2}$ sets $\{\mathbf{x},\mathbf{y}\}, \mathbf{x} \neq \mathbf{y}$, we have

$$\sum_{\mathbf{k}\in U'} \sum_{j=0}^{s-1} \binom{B(s,W,\mathbf{k},j)}{2} \leqslant \frac{p^{d-1}(p-1)N^2}{s} < \frac{(p^d-1)N^2}{s} \ .$$

As, $|U'| = p^d - 1$, we deduce immediately what we wanted to proof :

$$\sum_{\mathbf{k}\in U'} \sum_{j=0}^{s-1} \binom{B(s,W,\mathbf{k},j)}{2} < \frac{|U'|N^2}{s} \ .$$

□

Corollary 2.
We recall that $N = |W|$.
At least $\frac{1}{2}$ *of the* $\mathbf{k} \in U'$ *are such that*

$$\sum_{j=0}^{N-1} B(N,W,\mathbf{k},j)^2 < 5N \ .$$

Proof. From theorem 1, we have

$$\sum_{\mathbf{k}\in U'} \sum_{j=0}^{N-1} \binom{B(N,W,\mathbf{k},j)}{2} < \frac{|U'|N^2}{N} \ .$$

As at most one half of the terms in the sums can be larger than twice the average value, at least one half of the $\mathbf{k}$ are such that

$$\sum_{j=0}^{N-1} \binom{B(N,W,\mathbf{k},j)}{2} < \frac{2N^2}{N} \ .$$

We deduce:

$$\sum_{j=0}^{N-1} B(N,W,\mathbf{k},j)^2 < 4N + \sum_{j=0}^{N-1} B(N,W,\mathbf{k},j) < 5N \ .$$

□

Corollary 3.
At least $\frac{1}{2}$ *of the* $\mathbf{k} \in U'$ *are such that* $\mathbf{x} \mapsto (\mathbf{kx} \mod p) \mod 2N^2$ *is an injection.*

Proof. By the same reasoning, we show that for at least one half of the value $\mathbf{k}$:

$$\sum_{j=0}^{2N^2-1} \binom{B(2N^2,W,\mathbf{k},j)}{2} < 1 \ .$$

We immediately deduce that for such a $\mathbf{k}$, $\binom{B(2N^2,W,\mathbf{k},j)}{2} = 0$ for all value of j.
Therefore, we have $B(2N^2,W,\mathbf{k},j) < 1$ for all values of j from which we have the corollary.
□

3.2 Algorithm and example

Algorithm 1: Construction of the data structure

Input : D the number of dimensions of the hypergraph, n his number of vertices, H the array of size N containing the hyperedges and p a prime integer bigger than n and N.

Output: T an array of integers.

Let k be a random integer array of size D with values bounded by $p-1$.
while *k is not verifying corollary 2* **do**
  Generate a new random value for each cell of k.
Store k in the first D cells of an array T in which cells are initialized to 0.
H is now divided in N subsets W_j by the function $\mathbf{x} \mapsto (\mathbf{kx} \mod p) \mod N$.
$sum := d + N$
for $j := 0$ *to* $N-1$ *by* 1 **do**
  if $W_j \neq \varnothing$ **then**
    $T[d+j] := sum$
    $sum := sum + |W_j|^2$
for $j := 0$ *to* $N-1$ *by* 1 **do**
  if $W_j \neq \varnothing$ **then**
    $debut := T[d+j]$
    while *k is not verifying corollary 3 with $W := W_j$* **do**
      Generate a new random value for each cell of k.
    Store k in the first D cells of $T[debut]$.
    $T[debut + d] := |W_j|$.
    for *each hyperedge* $\mathbf{x}$ *in* W_j **do**
      $T[debut + d + 1 + ((\mathbf{kx} \mod p) \mod 2|W_j|^2)] :=$ index of x in H.
return T

The first research of a k has the following goal : dividing H into N subsets of small size. In each subset, we search a new k such that there is no collision. For a hyperedge in H, the two layers will allow us to find the index of the hypergraph and to compare it to itself. If a hyperedge is not is H, for any index returned by the data structure, the comparison will fail.

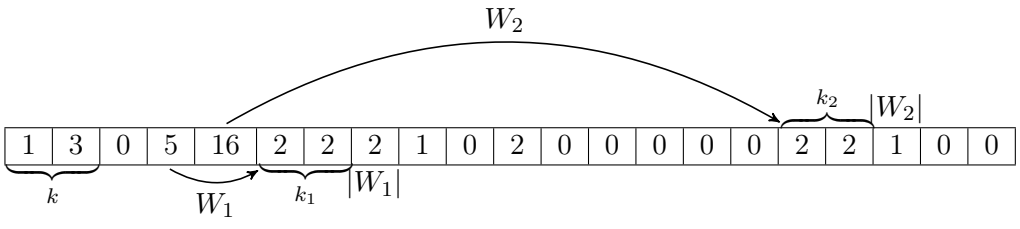


Figure 5: An example on the hypergraph $\{(1,2),(2,3),(2,4)\}$.

In this example, $p = 5$ and $k := (1,3)$. Therefore, $(1,2)$ and $(2,3)$ have the same image 1 by $(\mathbf{k}\cdot \mod 5) \mod 3$ while it maps $(2,4)$ to 2. As W_1 is the first W_j non empty, it is

stored from cell 5 to cell 14 and W_2 is stored from cell 15 to the end.

$(\mathbf{k} \cdot \mod 5) \mod 8$ maps $(1, 2)$ and $(2, 3)$ on the same image for the initial value of k. A new value is computed and we can store the indices of the hyperedges of W_1 are stored using the new function. $\mathbf{k}$ is unchanged for W_2 as there is only one element.

Algorithm 2: Search for a hyperedge

Input : D the number of dimensions of the hypergraph, n his number of vertices, H the array of size N containing the hyperedges, p a prime integer bigger than n and N, T an array build by the previous function and $\mathbf{x}$ a hyperedge.

Output: True if and only if $\mathbf{x} \in H$.

$k :=$ the first D cells of T
$j := (\mathbf{kx} \mod p) \mod N$
if $T[j + d] = 0$ **then**
  return False
$debut := T[j + d]$
$k :=$ the first D cells of $T[debut]$
$S := T[debut + d]$
$index := T[debut + d + 1 + ((\mathbf{kx} \mod p) \mod 2S^2)]$
if $H[index] \neq \mathbf{x}$ **then**
  return False
return True

3.3 Time and space complexity

We first want to analyse the complexity of the construction of the array T.

Thanks to corollary 2, we know that the algorithm will find a $\mathbf{k}$ which verifies the property with a probability at least equal to $\frac{1}{2}$. Testing such a property can be done in $\mathcal{O}(ND)$ by computing each of the $B(2N^2, W, \mathbf{k}, j)$. Therefore, the algorithm will find a suitable value for $\mathbf{k}$ in $\mathcal{O}(ND)$.

The first loop can be computed in $\mathcal{O}(N)$ as one loop can be performed in $\mathcal{O}(1)$.

For a given j, testing if $\mathbf{k}$ verify corollary 3 can be performed in $\mathcal{O}(D+|W_j|^2)$ by checking if each element of W_j has a different image using an array of size $|W_j|^2$. Because there is a probability of $\frac{1}{2}$ of finding a suitable $\mathbf{k}$, the algorithm will perform this step in average in $\mathcal{O}(|W_j|^2)$. Storing $\mathbf{k}$ and $|W_j|$ is made with a complexity of $\mathcal{O}(D)$. Storing the index of all the elements of W_j is done in $\mathcal{O}(|W_j|)$. One execution of a loop is done in $\mathcal{O}(D + |W_j|^2)$. The whole loop is computed in $\mathcal{O}(ND)$ because corollary 2 gave us an upper bound on the sum of the $|W_j|^2$.

Building the array T has a complexity in average of $\mathcal{O}(ND)$.

As knowing if a hyperedge $\mathbf{x}$ is in H requires only comparisons and dot products on vectors of size D, answering a query is performed in $\mathcal{O}(D)$.

We finally want to bound the size of T. The first $\mathbf{k}$ is stored using D cells while the first layer of hash is stored using N cells. For any j, W_j is stored using $2|W_j|^2 + D + 1$ cells. The second layer uses at most $10N + ND + N$ according to corollary 2. The whole array is at most of size $N(12 + D) + D$.

We achieved our goal to build a data structure which answers in $\mathcal{O}(D)$ to our queries. Its construction is still linear in average in the number of hyperedges as is the space in memory. But the upper bound in memory is much worse than the usual storage on hypergraphs.

4 Experiments

We studied both the time and space efficiency but we now want to know if our algorithm is efficient in practice. As a binary search is done in $\mathcal{O}(\log(N)D)$ on a sorted array of N $D-$hyperedges, we want to verify if our algorithm can be faster for realistic values of N. Otherwise, although its complexity is lower, it will not be useful.

An other problem we want to answer is about the construction time. We would like to have an acceptable construction time in comparison with the time to sort the list of hyperedges. Finally we want to evaluate the memory used in order to know if the theoretical bound of $N(12 + D) + D$ is effectively achieved. It will be paired with a test to know if the probability to find a k verifying corollary 2 or corollary 3 is near $\frac{1}{2}$.

All the following data has been acquired using the `crunch2` experimentation machine of the LIP. The technical characteristics are detailed in table 3. The algorithm were executed under reservation of the machine in order to avoid any other processes to be run in parallel to it.

All matrices used as test data where from SuiteSparse Matrix Collection, a set of sparse matrices collected from a wide range of applications [DH11]. The tensors are extracted from FROSTT [SCL+17]. The matrices and tensors used are described in table 4 and table 7

In each experiment, the algorithms of creation and research are executed 10 times in order to average the execution time and the memory used. The results are given in the appendix in the table 6 and table 8. A graphic representation will be used when required in order to study only the interesting data. A link to the codes of the algorithms I used are available in the appendix.

4.1 Construction time

As explained in the section 2.1, we decided to use sorted *Coordinate format* as control case to our algorithm. The search function which results from this choice is a binary search but we have to decide which sort we use. So, we will be able to compare the hyperedges sort and the construction of T.

As we work on hyperedges, which are analogous to tuples of constant size, the radix sort was a relevant choice. We compared it to the sort implemented in the library `algorithm`

of `C++` in order to decide which one will be used as the control case. Detailed results are stored in table 5.

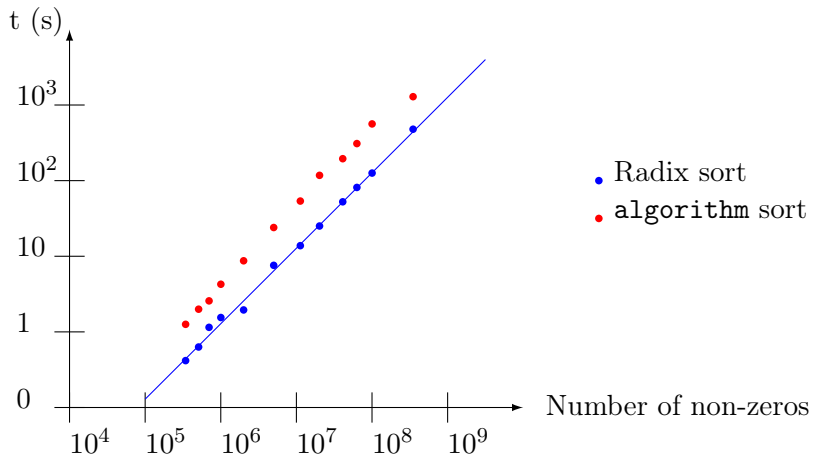


Figure 6: Evolution of sorting time as a function of the number of edges

Even if we see that the points of the two sets are aligned in figure 6, it is not sufficient to say that both algorithm are linear in the number of edges. We have to verify if the slope of the line is equal to 1 as both axis are in a log scale. For example, the line obtained by linear regression with the least-squares method has $f(x) = 0.998x - 0.888$ as equation with $\mathcal{X}^2$ equals to 0.9996 . The line is drawn in blue in figure 6.

We can remark that the sort implemented in `algorithm` is much slower despite being of the same complexity of the radix sort we implemented. The radix sort will then be used as the control case to evaluate the construction time performance of our algorithm.

We first wanted to test if the number of non-zeros was the only parameters which modify the time execution when D remains constant. To test such an hypothesis, we used four matrices with almost the same numbers of non-zeros ($\pm 1‰$) and compare the construction time of the data structures.

Name	Radix sort construction time (s)	Construction time of the array T (s)	Number of rows	Number of columns
SiO2	13.8522	19.529	155,331	155,331
Rail4284	15.864	19.074	4,284	1,096,864
Bmw3_2	13.342	19.347	227,362	227,362
Fcondp2	13.635	18.885	201,822	201,822

Table 2: Construction time of the different matrices

We remark in table 2 that the construction time of the array T remains the same when the dimensions of the matrices vary. It is not totally the case with the radix sort because the number of buckets required are linked to the dimensions of the matrices.

Therefore, we will use only square matrices to compare the two construction time, as a rectangular matrix could greatly modify the results of the radix sort for a given number of non-zero.

We then used matrices with a various number of non-zeros to compare the construction time of the two structures.

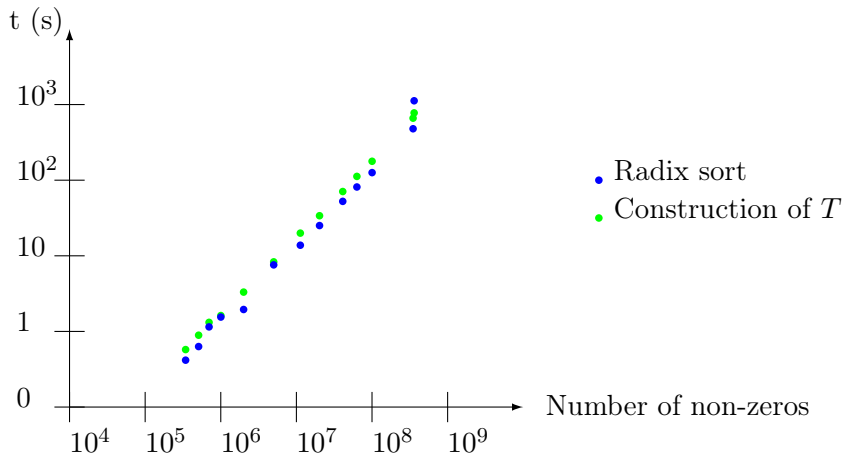


Figure 7: Evolution of construction time as a function of the number of edges

The construction of T is linear in the number of edges as the line obtained by the least-squares method has 1.01 as slope with a $\mathcal{X}^2$ of 0.9996. The construction of the sorted list is faster for almost any matrices of the test set and the difference between the two algorithm grows with the number of edges.

We can remark that `Kmer_A2a` does not follow this phenomenon. It is due to a significantly low number of non-zeros regarding the dimensions of the matrix, as the radix sort efficiency decreases when the size of the matrices grows while the number of non-zeros remains constant. `Wiki-Talk` shows the same result .

On matrices with a relatively high numbers of non-zeros per row or column, the radix sort is clearly faster than the construction of T despite being of the same complexity.

4.2 Query answer time

After having evaluated the construction time of T, we now want to test if our $\mathcal{O}(D)$ query response time is relevant for realistic size of matrices. To do so and to minimize the variance, we computed 10^6 random D-hyperedges search requests in the two structures.

We want to evaluate at first if the query response time remains constant when D is fixed to 2.

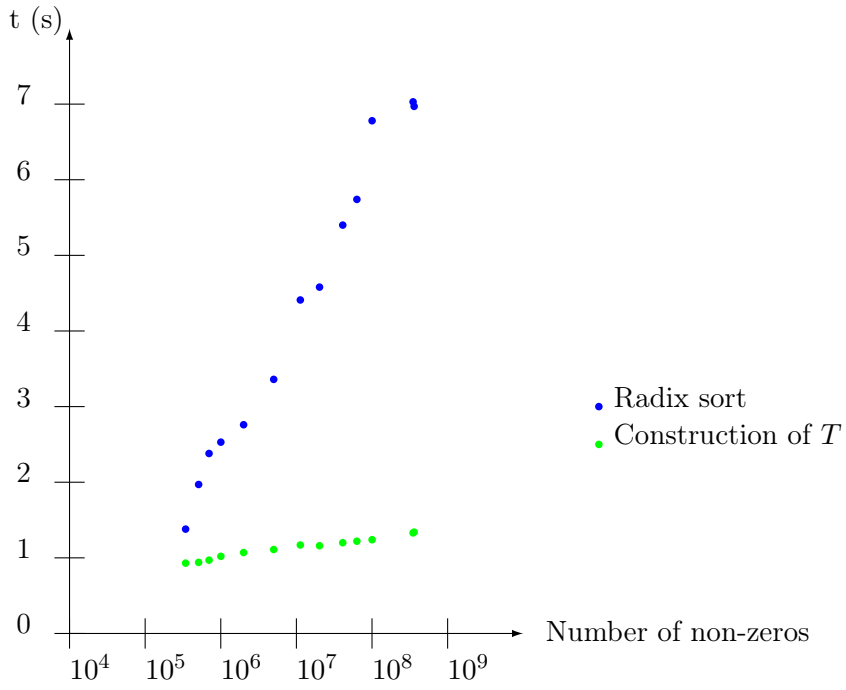


Figure 8: Evolution of query response time as a function of the number of edges

We remark on figure 8 that the query response time is not totally constant when the number of non-zeros grows. However, in addition to being faster than binary search algorithm for any number of non-zero, the query response time is multiplied by less than 1.4, when we multiply the number of non-zeros by 1000. It seems to grow logaritmically but with a low coefficient regarding the one used by the binary search. Such a result is very satisfying as binary search is a fast method when one want to look for an element.

As we evaluated the complexity of a query to $\mathcal{O}(D)$, we tested if it grows fast with the order of the hypergraph. To do so, we considered the query response time to be constant when D is and we then compared the average of the time to answer a query.

We remarked that, in practice, the query response time is not linear in D. It is probably due to the architecture of the machine. Indeed, the cache allows faster access to data close to those already called. The main cost of the search is the first call in memory in each of the steps of the algorithm. The call to values stored in the cache and the computation are fast compared to these calls. Therefore, the algorithm has a constant execution time while D remains small.

4.3 Size of data structure

Finally, we wanted to test if the theoretical bound of memory usage is achieved in practice and to study how this usage vary with D. We can at first remark that the numbers of cells by hyperedges required by T depends only on D. Only `Fcondp2` is an exception to this rule.

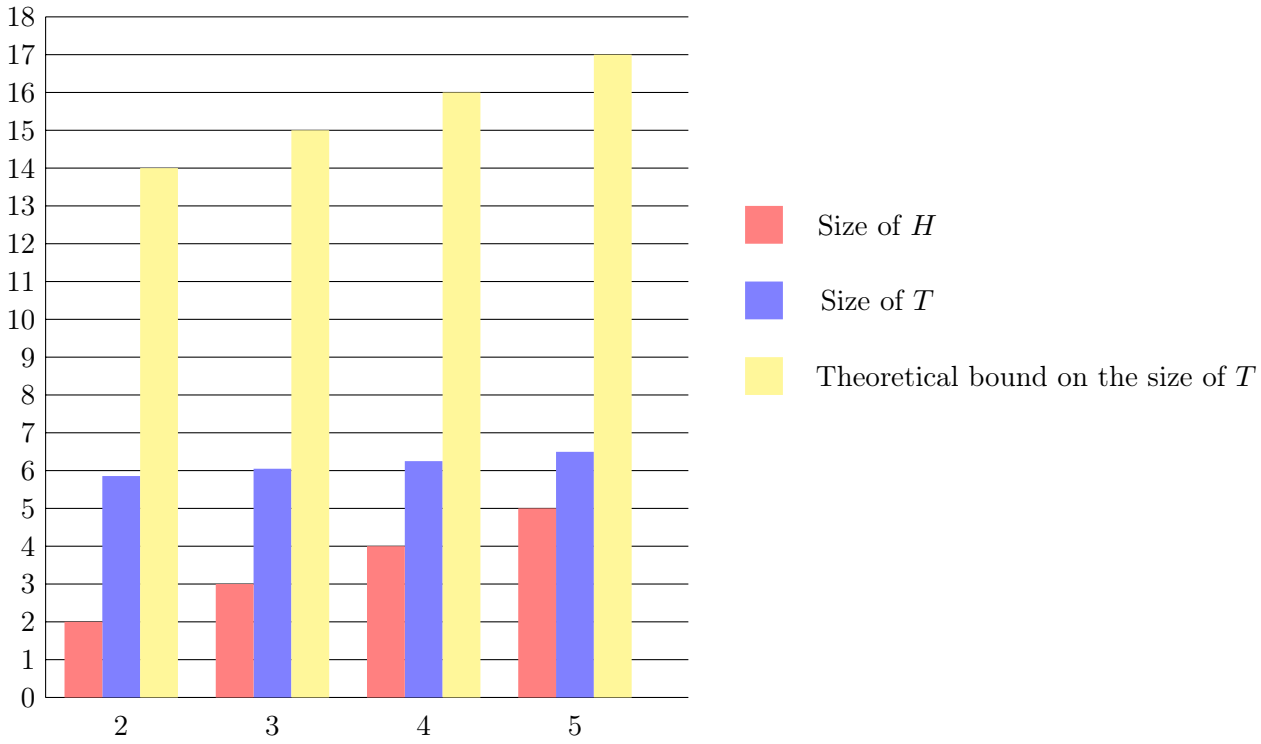


Figure 9: Evolution of the number of cells per hyperedge as a function of the order of the hypergraph

The theoretical bound of $\mathcal{O}(N(12 + D) + D)$ on the size of T is actually far from being achieved even if the number of cells per hyperedges seems to be linear in D, as we can see on figure 9. H grows faster in size than T. It means that if D is big enough, storing T in parallel of H will be negligible in memory usage while it allows to answer our queries faster.

Finally, the bound of $\frac{1}{2}$ obtained in corollary 2 and 3 seems very high in practice. During all the computation, no failure were registered for the first research of **k**.

5 Conclusion

During this internship, we wanted to build a data structure in order to answer our queries in $\mathcal{O}(D)$ using a memory space linear in the size and number of hyperedges. This

goal has been achieved and the difference of complexity has been highlighted on matrices and tensors of realistic size. The main problem of our data structure remains the construction time, much slower than a sort on the hyperedges. The loss in memory usage decreases with the order of the hypergraph.

We can weight our theoretical results by removing the assumption that all access and computation of usual operations on integers can be done in $\mathcal{O}(1)$. A factor $\mathcal{O}(\log(N))$ is hidden in the experiments because all the computations were done on integer of bounded size. The array T requires computation of numbers of size N which may slow down both the construction and the query answer. It is not the case when using a sorted approach. It will improve the memory usage by requiring bigger cells. However, the query response time being way faster using T, using integers of unbounded size will not make the binary search faster.

My internship was at first supposed to be focused on Bloom filters and more generally, on probabilistic data structures, in order to store hypergraphs. However, I took a totally different direction because I worked on a data structure with no error probability. It was both very exciting and frightening for me to be able to change my goal depending on what was the most intriguing. I think that this freedom of research allowed me to enjoy this internship a lot. Moreover, personal research was more in the heart of my internship than last year and I really liked it.

I would like to thanks Bora Uçar, my internship supervisor and Fanny Dufossé for agreeing to supervise me in these complicated conditions. Our regular meetings have allowed me to stay motivated and produce this work which I am satisfied of.

References

[BK08] Brett W. Bader and Tamara G. Kolda. Efficient matlab computations with sparse and factored tensors. *SIAM Journal on Scientific Computing*, 30(1):205–231, 2008.

[Blo70] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[DH11] Timothy A. Davis and Yifan Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software*, 38(1):1:1–1:25, 2011.

[FAKM14] Bin Fan, Dave G. Andersen, Michael Kaminsky, and Michael D. Mitzenmacher. Cuckoo filter: Practically better than Bloom. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14, pages 75–88, New York, NY, USA, 2014. ACM.

[FKS84] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with $O(1)$ worst case access time. *J. ACM*, 31(3):538–544, June 1984.

[GMS92] John R. Gilbert, Cleve Moler, and Robert Schreiber. Sparse matrices in matlab: Design and implementation. *SIAM Journal on Matrix Analysis and Applications*, 13(1):333–356, 1992.

[KH19] Tamara G. Kolda and David Hong. Stochastic gradients for large-scale tensor decomposition. arXiv, June 2019. http://arxiv.org/abs/1906.01687.

[RK12] Ori Rottenstreich and Isaac Keslassy. The bloom paradox: When not to use a bloom filter? *IEEE/ACM Transactions on Networking*, 23:1638–1646, 05 2012.

[SCL+17] Shaden Smith, Jee W. Choi, Jiajia Li, Richard Vuduc, Jongsoo Park, Xing Liu, and George Karypis. FROSTT: The formidable repository of open sparse tensors and tools. `http://frostt.io/`, 2017.

Appendix

My gitlab deposit

`https://gitlab.aliens-lyon.fr/jbertr02/internship-m1/-/tree/master`

CPU	Total Cores	Memory (GB)	System type
4 × Intel(R) Xeon(R) CPU E5-4620 0 @ 2.20GHz	32	378	PowerEdge R820 (Dell Inc.)

Table 3: Technical characteristic of the `crunch2` experimentation machine

Name	Matrix Id	Number of rows	Number of columns	Number of non-zeros
Vibrobox	379	12,328	12,328	342,828
Soc-Epinions1	2284	75,888	75,888	508,837
Md2010	2601	145,247	145,247	700,378
G_n_pin_pout	2574	100,000	100,000	1,002,396
Analytics	2851	303,813	303,813	2,006,126
Wiki-Talk	2291	2,394,385	2,394,385	5,021,410
SiO2	1367	155,331	155,331	11,283,503
Rail4284	1658	4,284	1,096,894	11,284,032
Bmw3_2	1219	227,362	227,362	11,288,630
Fcondp2	1260	201,822	201,822	11,294,316
Bundle_adj	2664	513,351	513,351	20,208,051
Emilia_923	2542	923,136	923,136	41,005,206
Geo_1438	2545	1,437,960	1,437,960	63,156,690
Mycielskian17	2773	98,303	98,303	100,245,742
Stokes	2845	11,449,533	11,449,533	349,321,980
Kmer_A2a	2805	170,728,175	170,728,175	360,585,172

Table 4: Characteristics of the matrices used during the experiments

Name	Radix sort construction time (s)	`algorithm` sort construction time (s)
Vibrobox	0.41641	1.26151
Soc-Epinions1	0.62839	2.02824
Md2010	1.1426	2.594435
G_n_pin_pout	1.5429	4.281195
Analytics	1.945	8.7881
Wiki-Talk	7.6071	23.99595
SiO2	13.8522	53.7031
Bundle_adj	25.36	116.861
Emilia_923	52.271	195.553
Geo_1438	81.48	308.404
Mycielskian17	127.09	568.74
Stokes	485,26	1288,2

Table 5: Detailed results of the two sorting algorithms on matrices.

Name	Construction time (s)		Queries answer time (s)$\times 10^6$		Cells by hyperedges of our algorithm
Vibrobox	0.41641	0.572657	1.3878	0.93693	5.7337
Soc-Epinions1	0.62839	0.81272	1.9746	0.94278	5.773
Md2010	1.1426	1.3297	2.3883	0.96637	5.7879
G_n_pin_pout	1.5429	1.6094	2.525	1.0236	5.7791
Analytics	1.945	3.2871	2.7511	1.0744	5.6917
Wiki-Talk	7.6071	8.4037	3.3649	1.1139	5.7901
SiO2	13.8522	19.529	4.4104	1.1743	5.7226
Rail4284	15.864	19.074	4.2967	1.1835	5.8249
Bmw3_2	13.342	19.347	4.2853	1.0759	5.5819
Fcondp2	13.635	18.885	5.0753	1.1859	7.6235
Bundle_adj	25.36	33.557	4.5757	1.1569	5.7176
Emilia_923	52.271	70.486	5.407	1.196	5.6869
Geo_1438	81.48	110.03	5.7412	1.2176	5.7059
Mycielskian17	127.09	178.2	6.7846	1.238	5.7635
Stokes	485.26	665.68	7.0332	1.3328	5.7045
Kmer_A2a	1129.5	770.31	6.9702	1.3375	5.8101

Table 6: Detailed results of the differents algorithms on matrices. Each column contain the results obtained by the radix sort and our algorithm

Name	Order	Dimensions	Number of non-zeros
Chicago-crime-comm	4	6,186 × 24 × 77 × 32	5,330,673
Chicago-crime-geo	5	6,186 × 24 × 380 × 395 × 32	5,330,673
Vast-2015-mc1-3d	3	165,427 × 11,374 × 2	26,021,945
Vast-2015-mc1-5d	5	165,427 × 11,374 × 2 × 100 × 89	26,021,945
Enron	4	6,066 × 5,699 × 244,268 × 1,176	54,202,099
NELL-2	3	12,092 × 9,184 × 28,818	76,879,419
Flickr-4d	4	319,686 × 28,153,045 × 1,607,191 × 731	112,890,310
Flickr-3d	3	319,686 × 28,153,045 × 1,607,191	112,890,310
Delicious-4d	4	32,924 × 17,262,471 × 2,480,308 × 1,443	140,126,181
Delicious-3d	3	32,924 × 17,262,471 × 2,480,308	140,126,181
NELL-1	3	2,902,330 × 2,143,368 × 25,495,389	143,599,552

Table 7: Characteristics of the tensors used during the experiments

Name	Construction time (s)		Queries answer time (s)$\times 10^6$		Cells by hyperedges	
Chicago-crime-comm	5.0469	10.997	5.8201	1.1963	4	6.196
Chicago-crime-geo	6.9872	11.405	4.7894	1.2478	5	6.405
Vast-2015-mc1-3d	22.471	47.5357	5.7498	1.2443	3	6.0196
Vast-2015-mc1-5d	37.017	53.834	6.4008	1.4113	5	6.587
Enron	48.19	123.77	5.5464	1.2343	4	6.318
NELL-2	67.147	170.64	7.3351	1.1918	3	6.0555
Flickr-4d	149.91	237.52	7.2881	1.4224	4	6.321
Flickr-3d	134.82	225.69	7.2831	1.3613	3	6.0565
Delicious-4d	196.5	289.32	6.3093	1.387	4	6.321
Delicious-3d	168.44	276.06	6.2923	1.3292	3	6.0568
NELL-1	214.8	283.4	5.9065	1.3287	3	6.057

Table 8: Detailed results of the differents algorithms on tensors. Each column contain the results obtained by the radix sort and our algorithm.