UNIVERSITÀ DEGLI STUDI DI MILANO

FACOLTÀ DI SCIENZE E TECNOLOGIE

PH.D THESIS

CYCLE XXX

Engineering Compressed Static Functions and Minimal Perfect Hash Functions

Supervisor Prof. Sebastiano Vigna
Co-Supervisor: Prof. Paolo Boldi

Ph.D. Candidate:
Marco Genuzio
Matr. Nr. R10919

Academic year 2016 - 2017

Contents

Abstract

Static functions are data structures meant to store arbitrary mappings from finite sets to integers; that is, given universe of items U, a set of $n \in \mathbb{N}$ pairs (k_i, v_i) where $k_i \in S \subset U, |S| = n$, and $v_i \in \{0, 1, \ldots, m-1\}, m \in \mathbb{N}$, a static function will retrieve v_i given k_i (usually, in constant time). When every key is mapped into a different value this function is called *perfect hash function* and when $n = m$ the data structure yields an injective numbering $S \to \{0, 1, \ldots n-1\}$; this mapping is called a *minimal perfect hash function.*

Big data brought back one of the most critical challenges that computer scientists have been tackling during the last fifty years, that is, analyzing big amounts of data that do not fit in main memory. While for small keysets these mappings can be easily implemented using hash tables, this solution does not scale well for bigger sets.

Static functions and MPHFs break the information-theoretical lower bound of storing the set S because they are allowed to return *any* value if the queried key is not in the original keyset. The classical constructions technique for static functions can achieve just $O(nb)$ bits space, where $b = \log(m)$, and the one for MPHFs $O(n)$ bits of space (always with constant access time). All these features make static functions and MPHFs powerful techniques when handling, for instance, large sets of strings, and they are essential building blocks of space-efficient data structures such as (compressed) full-text indexes, monotone MPHFs, Bloom filter-like data structures, and prefix-search data structures.

The biggest challenge of this construction technique involves lowering the multiplicative constants hidden inside the asymptotic space bounds while keeping feasible construction times. In this thesis, we take advantage of the recent result in random linear systems theory regarding the ratio between the number of variables and number of the equations, and in perfect hash data structures, to achieve practical static functions with the lowest space bounds so far, and construction time comparable with widely used techniques. The new results, however, require solving linear systems that require more than a simple triangulation process, as it happens in current state-of-the-art solutions. The main challenge in making such structures usable is mitigating the cubic running time of Gaussian elimination at construction time. To this purpose, we introduce novel techniques based on *broadword programming* and a heuristic derived from *structured Gaussian elimination.* We obtained data structures that are significantly smaller than commonly used hypergraph-based constructions while maintaining or improving the lookup times and providing still feasible construction.

We then apply these improvements to another kind of structures: *compressed static hash functions*. The theoretical construction technique for this kind of data structure uses prefix-free codes with variable length to encode the set of values. Adopting this solution, we can reduce the space usage of each element to (essentially) the entropy of the list of output values of the function. Indeed, we need to solve an even bigger linear system of equations, and the time required to build the structure increases.

In this thesis, we present the first engineered implementation of compressed hash functions. For example, we were able store a function with geometrically distributed output, with parameter $p = 0.5$ in just 2.28 bits per key, independently of the key set, with a construction time double with respect to that of a state-of-the-art non-compressed function, which requires $\approx \log \log n$ bits per key, where n is the number of keys, and similar lookup time. We can also store a function with an output distributed following a Zipfian distribution with parameter $s = 2$ and $N = 10^6$ in just 2.75 bits per key, whereas a non-compressed function would require more than 20, with a threefold increase in construction time and significantly faster lookups.

Glossary

perfect hash function (PHF)

A function that defines an injective mapping from the set of keys to the set of values. 1

lazy Gaussian elimination

A heuristic that reduces the number of equation to be solved by standard Gaussian Elimination. 3

hypergraph

A structure consisting of a pair (V, E); the set V is the set of nodes, and E is the set of edges; each edge is a subset of V. 11

nodes

See hypergraph, we use w to identify a node, they are also called *vertex*. 11

edges

A set of nodes, we use e to identify an edege. 11

r***-uniform hypergraph***

A hypergraph such that each edge contains exactly r nodes. 11

degree

The number of edges which contain a node. 11

k**-core**

A maximal induced connected subgraph such that each node has degree at least k, and it is denoted by the letter K_k. 12

hinge

A variable uniquely associated with an equation by the peeling process. 13

cycle

A finite sequence of adjacent nodes wherein a vertex is reachable from itself, it is usually identified with the letter C. 13

path

A sequence of adjancent nodes and it is usually identified with the letter P. 13

triangulated

A square matrix converted into an upper triangular form. 14

random graph model

We use Erdös-Rényi random graph model, this model chooses uniformly a graph $G(m, n)$ from the collection of all possible graph which have m nodes and n edges. 15

weight

Number of non-zero elements in a vector. 15

orientation

The process that assigns uniquely a node to an edge. 16

collision

A collision occurs when there are two distinct keys k_i, k_j such that $f(k_i) = f(k_j)$ that is when the function is not injective. 19

rank data structure

A data structure that allows `rank` operation. 20

bit array

A Sequence of bits, we assume that it is possible to access any position of the bit array in constant time, and it is denoded by the symbol $\mathcal{B}$. 20

s-bit digesting function

A function such that associate with each element of the universe a string of s bits. 20

code

It is a map c from a set of symbols Σ to the set of bit string $\{0, 1\}^*$. 23

codewords

The range of code, the length of a codeword is $|c(\sigma)|$. 23

prefix-free

A code is *prefix-free* if given $\sigma, \tau \in \Sigma$ it never happens that $c(\sigma) \preceq c(\tau)$ or $c(\tau) \preceq c(\sigma)$. 24

List of Symbols

n Number of keys. 1

m The number of variables in a linear system of equations that is the number of nodes in the hypergraph. 11

$a_{i,j}$ Element of the $m \times n$ matrix A in row $0 \leq i \leq n-1$ column $0 \leq j \leq m-1$. 11

b_l Element of size n vector B in position $0 \leq l \leq n-1$. 11

A It is the matrix of coefficients of a linear system. 11

x Vector of variables. 11

B It is the vector of constant terms of a linear system. 11

r Number of nodes in an edge. 11

k Minimum degree of a node in order to remain after the *peeling* procedure. 12

$O(E)$ Asymptotically grows at most $k \cdot (E)$ for some constant $k > 0$. 12

$\perp$ Special symbol for undefined value. 13

c Any ratio between the number of nodes and the number of edges for a random r-hypergraph. 15

$\gamma_{r,k}$ The threshold for the appearance of a k-core in a r-hypergraph. 15

γ_r Shortcut to $\gamma_{r,2}$. 15

β_r Calkin lower bound. 15

$c_{k,r}$ Threshold for the solvability of k-XORSAT problem. 16

U Universe of keys. 19

$\log(x)$ Logarithm base 2 of x. 20

e Euler's numer, $e = 2.718281828\ldots$. 20

u The size of the Universe U. 20

ϵ A small constant near zero. 20

$\{0,1\}^n$ Set of all the possible bit strings of length n, we use $\{0,1\}^*$ to represent the set of all possible bit string. 20

$|\mathcal{B}|$ The size of the bit array $\mathcal{B}$. 20

$\texttt{rank}_{\mathcal{B}}(p)$ Number of ones in the bit array $\mathcal{B}$ up to position $0 \leq p \leq |\mathcal{B}|$. 20

$o(n)$ Asymptotically grows at most $k \cdot (n)$ for all the constants $k > 0$. 20

$\mathcal{B}[i]$ The bit in position $0 \leq i \leq |\mathcal{B}| - 1$ in the bit array $\mathcal{B}$. 20

b The number of buckets for Hash Compress Displace data structure. 21

$d \in \{0, 1, \ldots, r-1\}$ The index of the hinge variable in an equation. 22

$\Theta(n^3)$ Asymptotically bounded within constant factors k_1, k_2 above and below n^3. 23

$\ll$ Much less than. 23

σ An element in Σ, also called *symbol*. 23

$\preceq$ Prefix partial order,for $\mathcal{X}, \mathcal{Y} \in \{0,1\}^*$ we have $\mathcal{X} \preceq \mathcal{Y}$ iff $\exists \mathcal{Z} \in \{0,1\}^*$ such that $\mathcal{Y} = \mathcal{X}\mathcal{Z}$. 24

$\mathcal{X}\mathcal{Y}$ Given two bit string $\mathcal{X}, \mathcal{Y}$, we use $\mathcal{X}\mathcal{Y}$ when all the bits in $\mathcal{Y}$ are appended after the bits in $\mathcal{X}$. 24

T A sequence of symbols $\sigma_0\sigma_1 \ldots \sigma_{n-1}$, such that all $\sigma_i \in \Sigma$ for $0 \leq i \leq n-1$. 24

B_l An element of the set of buckets $\{B_0, B_1, \ldots B_{m-1}\}$ in position $0 \leq l \leq n-1$ for BeV construction. 25

$c(f(k_i))_d$ The bit in position $0 \leq d \leq |c(f(k_i))| - 1$ in the codeword $c(f(k_i))$. 26

$\mathbf{F}_2$ Galois field containing 2 elements, $\{0,1\}$. 26

$\mathbf{F}_3$ Galois field containing 3 elements, $\{0,1,2\}$. 30

n' The size of a chunk. 40

L Maximum final length for Length-Limited Huffman code. 42

ℓ Maximum length of the decoding table for canonical Huffman code. 44

w_l The length of the longest codeword in the l-th chunk. 45

N Number of element of a Zipfian distribution. 67

s Exponent characterizing a Zipfian distribution. 67

Chapter 1

Introduction

Motivation

Since the early ‘50s computer scientists focused their efforts on how to store information and how to retrieve information. One of the most common models of data representation is the key/value mapping. This abstraction is well suited to store functions with a finite domain. Although it looks like an elementary model, several solutions have been offered, each of them with a different purpose and different tradeoffs.

A collection of key/value pairs such that each key does not appear more than once is known as dictionary or map, and can be implemented using standard techniques such as hash tables [16] or search trees [25].

For any key in the domain, a dictionary returns the matching value. For keys outside of the domain, the dictionary usually returns a special value, denoting the absence of the key. Dictionaries are ubiquitous in applications.

A map can have additional properties: it can be, for instance, a *perfect hash function (PHF)* or a *minimal perfect hash function (MPHF)*. A perfect hash function is an injiective mapping, this is, each key is associated with a distinct value. If a function is perfect, the size of the keyset is equal to $n \in \mathbb{N}$, and the set of output values is $\{0, 1, \ldots, n-1\}$, then the function is called *minimal*.

Based on the data to store and on the task to perform, computer scientists separated *static* from *dynamic* data structures. A static data structure does not allows dynamic modifications: when adding an element or changing the value associated to a key, all the data structure must be recomputed from scratch. With this premise is clear that this kind of data structure is excellent for big dataset that changes rarely or that takes a long time to be computed.

An important building blocks of some static data structure are static functions. In this thesis we will mainly focus on a (nearly) succinct representation of static hash functions.

Computer scientists have studied succinct data structure since the late ‘80s. A data structure is *succinct* if its space usage is equal to the information-theoretical lower bound plus a smaller-order factor, and the speed of its operations is the same as that of a standard data structure. Unfortunately some of these theoretical data structures are impractical, the main causes are their complexity and the fact that the required size of the keyset is too big. So the computer scientists accepted a mild increase in space to provide simple and very efficient practical data structures.

In this thesis, we start from the traditional technique proposed by Majewski, Wormald, Havas, and Czech [41] (or MWHC) to build a static data structure for function representation. This technique is very simple; it uses a construction based on random linear system solving, and can be adapted to store minimal perfect hash functions in a very efficient way. If the ratio between variables and equation in the linear system is above a certain threshold we can find the solutions in linear time. But this ratio influence the final size of the data structure. During the years, computer scientists focused their reasearch on the methods to reduce the space occupation of the data structure. The most practical solutions are: use a smaller ratio [2, 20] and compression [35].

The first solution has been proposed by Dietzfelbinger and Pagh [20] and experimented by Aumüller et al. [2]. Unfortunately the experimental results were not rewarding because their construction time was too high. Furthermore, a recent result on a logical problem [19] suggests that the ratio between variables and equation in the linear system can be lowered.

Instead, Hreinsson, Krøyer, and Pagh [35] follow the second option and defined a new teoretical data structure that uses compression. We to decided fill the gap between theoretical and real data structures by implementing this solution.

Problem and Challenges

As already anticipated, some useful approaches, both theoretical and pratical, have been proposed to solve static functions; however there is a large space for improvement, especially from the practical point of view.

In general, we want to reduce the final size of the data structure and keep a reasonable construction time.

The first challenge is the same faced by Aumüller et al. [2]. Reducing the ratio between the number of variables and the number of equation inside the linear system denies the possibility of a linear time system solving. In order to find the solutions one of possible algorithms is Gaussian elimination, but its complexity is cubic in the number of equations. Some better solutions have been proposed during the years [52, 55], but none of them was suitable for sparse equations. So, the first challenge we tackled was finding a suitable heuristic for speed up system solving. The second problem was the implementation of a theoretical data structure for compressed static functions described in [35]. For this construction technique the number of equations, thus the number of variables, present in the linear system depends on the distribution of the output values of the function. This leads to a linear system that contains more equation than the MWHC construction technique and we did not know if the solutions that we proposed in order to speed up system solving were sufficient to obtain a reasonable construction time.

Finally, compressing the data structure requires a system that converts information to symbols and viceversa, this system is tipically called a *code*. One of the challenges was finding an efficient code, both in terms of space consumption and decoding speed.

Main Contributions

As we have seen in the previous sections, most of our problems are related to the solution of big linear systems. Since the systems are modular, and we need exact solutions, some form of Gaussian elimination is the primary method available, and we devised few techniques to speed it up. In particular:

- we introduced the use of *broadword programming* [37] for manipulating equations [28]. These methods aim at packing multiple values in a machine word and process them simultaneously. For our problem, the inner loop of the Gaussian elimination is entirely composed of row operations, i.e. addition and subtraction of two rows of the same matrix. By using this technique we can process sixty-four variables at the time when building static hash functions and thirty-two when building minimal perfect hash functions.
- We propose a new version of *structured Gauss elimination* without parameters, which we call *lazy Gaussian elimination* [28]. This heuristic method aims at reducing the number of operations needed in the solution of a linear system by trying to isolate some variables appearing in a significant number of equations and then rewrite the rest of the system using just those variables. Since our linear system is very sparse, this heuristic method turned out to be extremely useful, reducing the size of the system to be solved by standard elimination to around 4% of the original one.
- We implemented and tested extensively over large dataset several kinds of functions. No scalable construction technique was previously known for our structures. In particular, we engineered for the first time [29] the compressed structures described in [35].

All the code is available as part of the `Java` open source libray Sux4J available online.

Organization

The core of this thesis is the engineering and experimental evaluation of some techniques to store static functions and minimal perfect hash functions.

In the first part, we present the theory behind our approach, by introducing some basic definitions about linear system and hypergraphs, discussing the relevant results about "random linear system solving "and "hyperedge orientability", which are the ground for our construction methods.

After this premise, we presents critically the construction techniques for static hash functions already present in literature, analyzing the strengths and weaknesses of each structure, and, most importantly, introduces the improvements we added to solve the linear system more quickly.

In the second part, we mainly focus on experiments. We will describe the terms of comparison between different construction techniques. We will test how parameters, such as the number of variables, influence these measures. For our experimental analysis, we used mainly two keysets containing hostnames and URLs gathered with a web crawler. For each construction we discuss the results and, where possible, we compare the results with different methods. The analysis for the compressed static functions will be deeper. In particular, we will discuss in which cases gives the best performances, in comparison with standard methods. Furthermore, the solution offered in literature exhibits some flaws for realistic datasets.

In the last chapter, we summarize the results of this work.

Part I

Theory

Contents

Introduction

The first part of the thesis focuses on the theoretical background relevant for our study/ the theoretical basis of our work. We start from Chapter 2 by giving some basic definitions to understand the different construction techniques. We will show how we can represent any linear system by using a hypergraph. Then, we will discuss some conditions for the linear systems to be solvable (Section 2.4). In the third chapter, we will present several construction techniques for the representation of static functions. We will describe the most famous construction techniques. We will start from *Minimal Perfect Hash Functions* in Section 3.1. Then we are going to extend the hypergraph-based construction technique to obtain *static functions* in Section 3.1.3 and compressed static hash functions in Section 3.2. We will see that compressed static functions require an *encoding scheme*, and we will define it in Section 3.2.1. Finally, in Chapter 4 focuses on the improvements of the solving methods introduced in this work. In Section 4.1 we will show how to speed up row operations by using a different representation for the equations. We will use *broadword programming* to perform row operations. In Section 4.2 will present a highly efficient heuristic for linear system processing. The last part is dedicated to some minor improvements for particular construction technique.

Chapter 2

Linear Systems

2.1 Introduction

This chapter introduces to the reader the basis to understand the construction techniques that we will present later. These notions are taken from graph theory, but what they all have in common is that they are related to static functions representation.

2.2 Linear Systems

A linear system containing $n \in \mathbb{N}^+$ equations and $m \in \mathbb{N}^+$ variables can be written as:

$$\begin{cases} a_{00}x_0 + \cdots + a_{0(m-1)}x_{m-1} = b_0 \\ \vdots \\ a_{(n-1)0}x_0 + \cdots + a_{(n-1)(m-1)}x_{(m-1)} = b_{(n-1)} \end{cases} \tag{2.1}$$

Each elements $a_{i,j} \in \mathbb{R}$ are called *coefficients* and $b_l \in \mathbb{R}$ are named *constant terms.* In particular we will use descrete values for the coefficients and for the constant terms. Another way to represent the system is by using a matrix form $Ax = B$ where A is a $m \times n$ matrix and it contains all the coefficients, x is a vector containing m variables and B is a vector with n constant terms.

$$A = \begin{bmatrix} a_{00} & \cdots & a_{0(m-1)} \\ \vdots & \ddots & \vdots \\ a_{(n-1)0} & \cdots & a_{(n-1)(m-1)} \end{bmatrix}; \quad x = \begin{bmatrix} x_0 \\ \vdots \\ x_{m-1} \end{bmatrix}; \quad B = \begin{bmatrix} b_0 \\ \vdots \\ b_{n-1} \end{bmatrix} \tag{2.2}$$

Now, we are going to show another way to represent a linear system by using *hypergraphs*, while in Section 2.4 we will focus on random linear systems.

2.3 Hypergraphs

A *hypergraph* is a pair (V, E) where V is a set of *nodes* and E is a set of non-empty subset of V called *edges*. Each edge can contain an arbitrary number of nodes nodes, but in general edges which connects a node to itself are not allowed. If every edge has exactly[1] $r \in \mathbb{N}^+$ nodes, then we a have a *r-uniform hypergraph* (r-hypergraph from now on). For instance, if $r = 2$ then we have a graph. If the hypergraph is not oriented the *degree* of a node is the number of edges that contain the node. We can use a hypergraph to represent a linear system, as shown in the next example.

Example 2.3.1. Given a set of equation:

$$\begin{cases} a_{00}x_0 + a_{03}x_3 = 3 \\ a_{10}x_0 + a_{11}x_1 + a_{13}x_3 = 0 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + a_{24}x_4 = 1 \\ a_{32}x_2 + a_{34}x_4 = 2 \end{cases} \tag{2.3}$$

The linear system in Equation 2.3 contains four equation and five variables. First, we fill the set of nodes V with all the variables appearing in the system. Then, for each equation we create an edge, i.e. a set of variables, with all the variables inside the equation. All the edges form the set E.

$$\begin{aligned} &V = \{x_0, x_1, x_2, x_3, x_4\}; \\ &E = \{\,\{x_0, x_3\}, \{x_0, x_1, x_3\}, \\ &\quad \{x_1, x_2, x_3, x_4\}, \{x_2, x_4\}\,\} \end{aligned} \tag{2.4}$$

The Figure 2.1 shows the hypergraph that represent the above linear system. The green edge represents the first equation, the yellow one the second, the blue one the third and last equation is red. Note that the coefficients are not rapresented in the hypergraph.

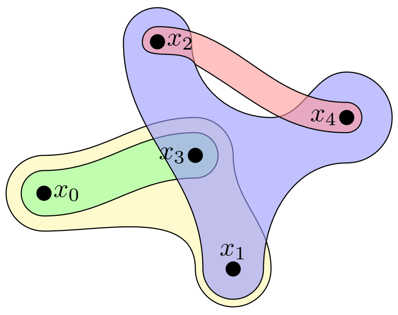


Figure 2.1: The hypergraph representing the linear system in Equation 2.3

[1] In practice we allow each edge to have at most r nodes.

Using the hypergraph representation, we can adapt a *peeling* procedure in order to find the solution of the linear system. This procedure iteratively removes nodes of degree less than $k \in \mathbb{N}^+$ (and their incident edges) until no further node can be removed. The advantage of this procedure can be implemented in linear time with a simple depth-first visit. Usually, this procedure detect a k-core, which is defined as a maximal induced induced connected subgraph such that each node has degree at least k. The procedure is summarized in Algorithm 1, and takes $O(E)$ time. This procudure returns a ordered list of pairs (node, set of edges).

Algorithm 1 Peeling procedure

```
1: function PeelingProcedure(Graph, k)
2:     create an empty list of pairs (nodes, edge) $L$
3:     while Exist a node $w$ with degree less than k do
4:         remove it from the graph
5:         remove the all possible incident edges $e$ from the graph
6:         append to $L$ the pair $(w, e)$ [or $(w, \perp)$ if $w$ has degree 0]
7:         update the degrees of all adjacent nodes
8:     end while
9:     return $L$
10: end function
```

In particular, if $k = 2$ we can solve the linear system in linear time. In fact, the list L contains a distinct node per edge, or, equivalently, one distinct variable per equation, in this case we call this variable the *hinge* of the equation: indeed, each node is uniquely associated to the only edge to which it was incident when it was peeled. If L contains all the nodes then we say that the procedure *succeeded*. We call *acyclic* all the hypergraphs that do not contains any *cycle*, i.e. a *path* of edges wherein a vertex is reachable from itself. Therefore the system is no longer solvable in linear time.

Theorem 1. An hypergraph G is acyclic if and only if for $k = 2$ after the peeling procedure the remaining hypergraph is empty.

Proof. We start by proving that if there exists a cycle inside G and we apply the peeling procedure then the remaining hypergraph is not empty. We define C as the finite sequence of distinct nodes $w_0 w_1 \cdots w_a w_0$. Let us notice that C is a cycle. By definition all the nodes in C have degree at least two and does not exist any vertex $w \in G \setminus C$ such that if we remove w from G then there exists a vertex in C with degree smaller than two. So all the nodes in the cycle cannot be removed by the peeling procedure, and the remaining hypergraph is not empty. Now, we need to prove that if G is acyclic then the remaining hypergraph is empty. By contradiction, suppose that G is acyclic and the remaining hypergraph is a nonempty 2-core. So we need to prove that any nonempty 2-core contains at least a cycle. Let K_2 be the nonempty 2-core and P be the longest sequence $w_0 w_1 \ldots w_{a-1} w_a$ of adiancent nodes in K_2. All the nodes $w_1 \ldots w_{a-1}$ have degree at least two, since the are connected with the previous and the successive node of the sequence. But the node w_a has degree at least two because it belongs to K_2. So either w_a

is connected to another node w not in P, but this case is impossibile since P was choosen to be of maximal length, or it is connected to $w \in w_0, \ldots, w_{a-2}$. Thus if $w = w_i$, $0 \leq i \leq a_2$, then $w_i w_{i+1} w_a w_i$ is a cycle.

So we proved that in K_2 there is at least one cycle. Now, K_2 is a induced subgraph of G, thus G contains a cycle, and this concludes the proof since contradicts the initial proposition.

$\square$

Example 2.3.2. In Figure 2.2 we show a complete example of the peeling procedure for the following system:

$$\begin{cases} x_1 + 3x_3 = 200 \\ x_3 + 2x_4 = 24 \\ x_3 + 41x_5 = 311 \\ x_2 + 10x_5 = 120 \end{cases} \rightarrow \begin{array}{l} V = \{x_1, x_2, x_3, x_4, x_5\}; \\ E = \{\, eq_1 = \{x_1, x_3\}, eq_2 = \{x_3, x_4\}, \\ \quad eq_3 = \{x_3, x_5\}, eq_4 = \{x_2, x_5\} \,\} \end{array} \tag{2.5}$$

The red nodes have degree equal to 1 and are going to be removed within the next step and appended in the list of hinges. At the end of the procedure, all the equations are sorted, and the list $L = [(x_1, eq_1), (x_2, eq_4), (x_5, eq_3), (x_3, eq_2), (x_4, \perp)]$. If we move from the the last element of L to the first one, the hinge variable did not appeared in a previous equation.

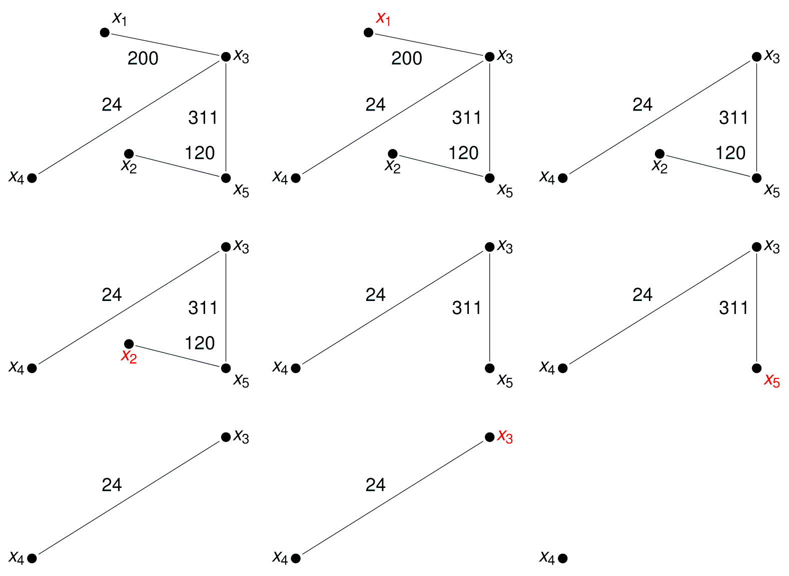


Figure 2.2: Peeling procedure applied to a to the graph defined by Equation 2.5

Now the system is *triangulated*, and is possibile to solve it by backward substitution. We assign the last standing variable, x_4, to 0 and we begin to solve the linear system. We start from the equation $x_3 + 2x_4 = x_3 + 0 = 24$, we assign $x_3 = 24$ and then we go back and substitute x_3 in the whole equations until all the system is solved. After x_3, we assign $x_5 = 7$ then $x_2 = 50$ and we conclude with $x_1 = 128$. This procedure does not work for every hypergraph; we will discuss the appearances of 2-cores in the next section.

2.4 Constants for Solvability / Orientability

As we have seen in the previous section, if the peeling procedure succeded then it is always possible to find a solution for the linear system. We now introduce the definition of a random graph generated by using the Erdös-Rényi *random graph model*.

Definition 2. Given $n, m \in \mathbb{N}^+$, the Erdös-Rényi random graph model choose uniformly at random a graph $G(m, n)$ from the collection of all possible graph which have m nodes and n edges.

This model can be trivially generalized for the hypergraphs. Computer scientists studied the presence of the k-cores in random graphs in several papers over the years. We denote with c the ratio between the number of nodes and the number of edges for a random r-hypergraph. Computer scientist focused their attention to the minimum value of c such that the random r-hypergraph is acyclic with high probability.

In 1994 Havas et al. [33] conjectured and verified experimentally the value of this threshold for graphs and 3-hypergraph. In 1996, Pittel et al. [48] provided a sound analysis of the threshold for the appearance of k-cores in random graph $G(m, n)$. In their analysis they also calculate the size of the k-core. Studying k-XORSAT formulas, Molloy [44] extended the heuristic from [48]. He proposed a simpler technique in order to compute the threshold for the appearance of a k-core in a random r-hypergraph. This particular threshold in the following text is denoted as $\gamma_{r,k}$. Such threshold is a consequence of Theorem 1 in [44]:

Theorem 3. For any $r, k \leq 2$ with r, k not both equal to 2. The threshold for the appearance of a k-core in a r-hypergraph $\gamma_{r,k}$, is defined by this equation:

$$\gamma_{r,k} = r! \cdot \left(\min_{x>0} \frac{x \cdot (r-1)!}{(1 - e^{-x} \sum_{i=0}^{k-2} \frac{x^i}{i!})^{r-1}} \right)^{-1} \tag{2.6}$$

The results for $k = 2$ and $r = 3$ are in accordance with the experimental values appearing in [33] and [17]. From now on we define $\gamma_r = \gamma_{r,2}$. In Figure 2.3 we show values for the threshold γ_r. As we proved earlier in Theorem 1, the absence of a 2-core is equivalent to acyclicity for r-hypergraph.

As you can see in Figure 2.3 $\gamma_{r,2}$ has a global minimum for $r = 3$.

Beyond that, the question arises whether it is possible to solve the system using a smaller ratio between nodes and variables than the threshold for acyclicity. One of most important

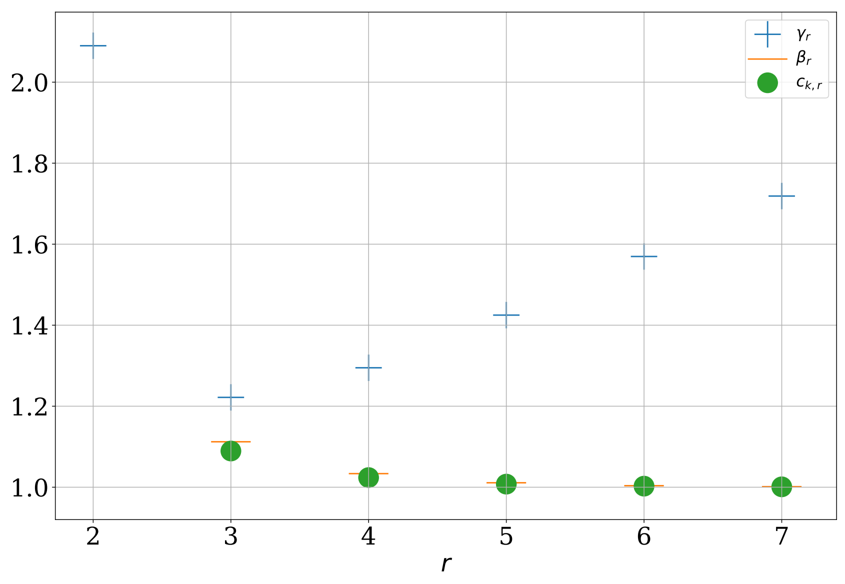


Figure 2.3: Values for γ_r, β_r and $c_{2,r}$ for a r-hypergraph

results in this direction is the bound presented by Calkin [14]. He proved the existence of a constant β_r, such that if $m > \beta_r n$ and the rows of the matrix A are just drawn at random from vectors of *weight* r then the limit of the probability that the rows of A are linearly dependent is zero as the number of rows tends to infinite (Theorem 1 [14]). In contrast with γ_r which has a finite minimum, β_r vanishes quickly as r increases. Using the Equation 2.7 it is possible to compute the value of β_r. An approximation of those values is plotted in orange in Figure 2.3.

$$\beta_r = 1 - \frac{e^{-r}}{\ln 2} - \frac{1}{2\ln 2} \cdot (r^2 - 2r - \frac{2r}{\ln 2} - 1) \cdot e^{-2r} \pm O(r^4) \cdot e^{-3r} \tag{2.7}$$

In [20] the authors suggested a theoretical data structure based on the solution of systems by Gaussian elimination. In [2] the authors perform some experiments using this approach. However, in spite of trying several techniques (e.g., splitting the keyset into small buckets, precomputing all the pseudoinverse of system matrices, trying different algorithm for system solving, etc.), there were not sufficient improovements for construction time (see for instance discussion in [2] and in [49]). In Section 3.1.3 we are going to give more details about this construction.

Actually, since we are interested in solving the system, and this might happen even if the system matrix has not full rank, we can even consider the k-XORSAT solvability threshold $c_{k,r}$ [19], which is smaller, and plotted in Figure 2.3.

We cite the values in the first line of the table presented in Section 2 of [19], as we can see, this values are smaller than the ones computed using the Equation 2.7.

Table 2.1: Values described by Dietzfelbinger et al. [19]

k/r	2	3	4	5	6	7
2	-	0.91793	0.97677	0.99243	0.99737	0.99906

2.4.1 Orientation

Building minimal perfect hash functions requires to associate each edge with one of its nodes, as we discussed previously. We call this association an *orientation*. When one solves a linear system by triangulation, the triangulation gives as a byproduct an orientation. However, when we apply Gaussian elimination this is no longer true, and an orientation has to be computed explicitly. In [19] Dietzfelbinger et al., studying k-XORSAT solvability, showed that the threshold for the orientability of a 3-hypergraph is the same as the one for k-XORSAT solvability. Since Goerdt and Falke have proved a result analogous to the one for k-XORSAT for modulo-3 systems [30], we can obtain an orientation of a random 3-hypergraph using the *generalized selfless algorithm* [19], and then solve the modulo-3 linear system induced by the orientation to get a perfect hash function. The generalized selfless algorithm runs in linear time in the number of the edges and has some controlled probability of failure. In case such a failure occurs, we generate a new hypergraph. The computation of solvability threshold is still an hot topic, as demonstated in a very recent paper [3].

2.5 Conclusions

So far, we presented the basic tools that we are going to use in the next chapters:

- how to represent linear system using a hypergraphs;
- a linear procedure to triangulate the system;
- constant for solvability and orientability of random linear systems.

As we will see later, we will put together all these elements to describe construction techniques for minimal perfect hash functions, static hash functions, and compressed functions.

Chapter 3

Construction Techniques

Introduction

In this chapter, we will present a summary of the construction techniques for minimal perfect hash functions, static hash functions, and compressed hash functions. We start from the minimal perfect hash function problem, which is historically the most relevant. In Section 3.1 we will define the problem and present some of the most successful techniques. The first will be *Hash Compress Displace* (HDC) in Subsection 3.1.1. Then we will present the hypergraph-based construction techniques in Subsection 3.1.2. We will begin with the one proposed by Majewsky, Wormald, Havas, and Czech (MWHC), then the refinement suggested by Botelho, Pagh and Ziviani (BPZ). Finally, in Section 3.2 we describe two different construction techniques that can be used to represent a compressed static function. The first one uses buckets and random tables and is described in Subsection 3.2.3. The second is based on linear systems and is presented in Subsection 3.2.4.

3.1 Functions

Our description starts from minimal perfect hash functions. Let us fix a finite set U, the *universe*, and a set $S \subset U$, the *key set*, whose elements are called *keys*. The problem can be defined as follows.

Definition 4. Given a subset S of size n of the universe U, store an injection $f : S \to \{0, 1, \ldots, n-1\}$ so that $f(k_i)$ can be computed in constant time for any $k_i \in S$.

This task can be easily implemented using a direct-addess tables [16]. While this method has the advantage that has a constant worst case lookup time, the size of the data structure depends on the size of the universe. If the size of the universe is large then this solution may be unfeasible. To overcome this issue, computer scientists proposed to use hash tables. The idea is to use a function on a key to compute the position where the value is stored. One of the main issue with this approach is that sometimes two distinct keys may have the same hash value so that a *collision* occurs. Over the years, several solution have been proposed in order to avoid collisions such as: separate chaining or cuckoo hashing or open addressing. All this solutions offers an average constant lookup time, but the worst case lookup time is $O(n)$. However, minimal perfect hash functions are allowed to return *any* value if the queried key is not in the original set S; this relaxation allows to break the information-theoretical lower bound of storing the set S and have a constant worst-case lookup time. Finally, we can state that the construction for minimal perfect hash function achieves just $O(n)$ bits. This makes minimal perfect hashing functions a powerful technique when handling, for instance, large sets of strings, and they are essential building blocks of space-efficient data structures such as (compressed) full-text indexes [9], monotone MPHFs [5, 7], Bloom filter-like data structures [10], and prefix-search data structures [6].

Fredman and Komlós [26], in one of the first studies on the topic, proved that the minimum space required by a minimal perfect hash function is $n\log(x)e + \log\log u - O(\log n)$ bits, if $n^{2+\epsilon} \leq u$, where u is the size of the universe U and ϵ a small constant near zero. This bound has been proven to be tight by Mehlhorn [42]; his construction takes $n\log e + \log\log u + O(\log n)$, but the lookup time is high.

The first construction with constant lookup time is the one proposed by Schmidt and Siegel [50]. Their construction uses $O(n + \log\log u)$ bits, but they did not give a quatitative extimation about the time required to find this structure. The best theoretical result, in this direction, came ten years later by Hagerup and Toley [32]. Their construction has expected linear time construction and occupies $n\log e + \log\log u + O(n(\log\ \ logn)^2/\log n + \log\log\log u)$ bits. However, the space usage of this technique is reasonable for an unfeasibly large n.

Once the best theoretical construction method is established, there are different lines of development, more focused on evaluation time and simplicity of construction method than optimal space. The easiest solution that has been proposed so far is perfect hash compression. Every perfect hash functions $f' : U \to \{0, 1, \dots, m-1\}, m \geq n$ can be "compressed" into a minimal perfect hash function using an extra bit array of size m and a *rank data structure* on this array. Given a *bit array* $\mathcal{B} \in \{0,1\}^n$, we define $|\mathcal{B}|$ the size of the bit array $\mathcal{B}$ and the $\texttt{rank}_{\mathcal{B}}(p)$ operation returns the number of ones up to position p. Jacobson [36] proposed a constant time implementation for the rank data structure that uses $o(n)$ additional bits. More recently, several authors proposed alternative constructions [31, 47, 54].

Consider a bit array $\mathcal{B}$ such that $\mathcal{B}[i] = 1$ if and only if i is an image of some key $k_i \in S$ through f'. Now we can define a new MPHF $f : U \to \{0, 1, \dots, n-1\}$ by

$$f(k_i) = \texttt{rank}_{\mathcal{B}}(f'(k_i)).$$

As we have seen before, rank queries can be executed in constant time, so f is as fast as f'. In this construction technique we use $m + o(m)$ extra bits.

Minimal perfect hash functions cannot be used to establish whether an element of the universe U is a member of the keyset S. One way to fix (partially) this behavior is to define an s-bit digesting function $\sigma : U \to \{0, 1, \dots, 2^s - 1\}$. Then, we can store an array of signatures A such that $A[f(k_i)] = \sigma(k_i)$. When we query for an element of the universe $x \in U$, we check whether $A[f(x)] = \sigma(x)$. If this is not true, x does not belong to S. If it is true, either x belongs to S or we hit a false positive. However, if σ is fully random, the probability of a false positive is at most 2^{-s}. This kind of data structure are named *approximate membership* data structure.

3.1.1 Hash, Displace and Compress

We will now present one of the most successful construction techniques. This is a modification of Pagh's construction "hash and displace" [46] that based is contruction on an old idea of Tarjan and Yao [53].

The construction technique we are going to analyze is called "hash, displace and compress" and was described in [8]. It builds a perfect hash function $f : U \to \{0, 1, \dots, p-1\}, p > n$ and then it compresses it using a rank structure. This technique adopts two levels of hashing and relies on the availability of a sequence of fully random hash functions.

Assume that exists an infinite sequence $h_0, h_1, \dots : U \to \{0, 1, \dots, p-1\}$, such that:

- $h_j(k_i)$ can be evaluated in constant time;
- no storage space is needed;
- $h_j(k_i)$, $k \in S$, $j \geq 0$ are fully random values, uniformly distributed in $\{0, 1, \dots, p-1\}$.

The first level of the data structure is defined by a random hash function $g : U \to \{0, 1, \dots, b-1\}$, $b \in \mathbb{N}^+$. This function divides U in b buckets. So, S is also divided into b buckets, but the cardinality of each bucket may vary. Each bucket B_j is defined by the following equation:

$$B_j = \{k_i \in S \mid g(k_i) = j\} \qquad 0 \leq j < b.$$

We define a indexing function $\sigma : \{0, \dots, b-1\} \to N$. For each bucket B_j (we enumerate buckets in decreasing-size order) we now pick the first function $h_{\sigma(j)}$ in the sequence above such that $h_{\sigma(j)}$ is injective on B_j, and moreover it has no collision with the $h_{\sigma(l)}$, $0 \leq l < j$. We can now describe our perfect hash function

$$f'(k_i) = h_{\sigma(g(k_i))}(k_i).$$

Once a perfect hash function has been defined, its output can be ranked to obtain a minimal perfect hash function.

From the analysis in Section 3 [8], w.h.p. $\sigma(j) < C \log n$ for some constant C. Thus, each number $\sigma(j)$ can be represented using $\log \log n + O(1)$ bits.

3.1.2 Hypergraph-based Constructions

In this section we analyze two different construction techniques. Both use an r-hypergraph to represent a random linear system.

The first technique was introduced in [41], and it is the one we referred to as the "standard MWHC technique". The authors intend to find a way to store an *order-preserving minimal perfect hash function*, which is a minimal perfect function mapping each element of the keyset to its rank in some prescribed order. Note that, in spite of the name, this problem is much more similar to the problem of storing a static function than to minimal hashing , and indeed the MWHC technique can be trivially extended to store arbitrary static functions.

As defined in Section 2.4, we call c ratio between nodes and variables. Let us fix a function $f : S \to \{0, 1, \dots, p-1\}$, $p > n$ we want to represent. Consider $r \geq 2$, and fully random hash functions $h_j : U \to \{0, 1, \dots, m-1\}$, $m = cn$. We can define a random r-hypergraph $G = (V, E)$ with $V = \{w_0, w_1, \dots, w_{m-1}\}$, and, for each $k_i \in S$, an edge $\{h_0(k_i), h_1(k_i), \dots, h_{r-1}(k_i)\}$. We think of the edge as representing the equation

$$w_{h_0(k_i)} + w_{h_1(k_i)} + \cdots + w_{h_{r-1}(k_i)} = f(k_i) \tag{3.1}$$

As we discussed in Example 2.3.2, if the peeling procedure succeded we obtain a list of equations, each associated with a distinct variable that does not appear before in the list: the system is now triangulated and can be easily solved by substitution. Remark that if the ratio c is greater than γ_r then the peeling succeded with high probability. The construction requires linear time, and we compute $f(k_i)$ by replacing the variables $w_{h_0(k_i)}, \dots, w_{h_r(k_i)}$ in Equation 3.1 with the corrisponding solution. This can be evaluated in constant time.

The same construction can be adapted to obtain a perfect hash function [15]: since during the triangulation we assign a one of the possible r variables, say $w_{h_d(k_i)}$, $d \in \{0, 1, \dots, r-1\}$, to the equation associated with k_i, we can set

$$w_{h_0(k_i)} + w_{h_1(k_i)} + \cdots + w_{h_{r-1}(k_i)} = d(k_i) \mod r.$$

We store the position $d(k_i)$ in order to obtain a perfect hash function from S to p. When we want to lookup for a value compute $d(k_i)$ by replacing the variables $w_{h_0(k_i)}, \dots, w_{h_r(k_i)}$, then we compute $h_d(k_i)$. As we discussed previously, we can use the ranking data structure to obtain a BPZ minimal perfect hash function [12].

In Section 4 of [12] Bothelo et al. analyze the space consumption of the BPZ representation using the 2-hypergraph. The authors state that a perfect hash function can be stored in $\approx 2.09n$ bits.

Instead, for minimal perfect hash function the value $r = 3$ provides the lowest space usage. Moreover, it makes it possible to avoid to store a rank bit array. Indeed, to create a rank data structure we assign two bits for $d(k_i)$ as follows:

- 00 all the variables that are not an hinge and are equal to zero;
- 01 variables equal to one;
- 10 variables equal to two;
- 11 hinge variables equal to zero.

Now, ranking can be performed by counting pairs of nonzero bits, more precisely, by counting the occurences of 01, 10 and 11. Finally, the resulting structure uses ≈ 2.62 bits per element.

3.1.3 Linear Systems

The MWHC-based construction for static functions leave the space for further improvements: Dietzfelbinger and Pagh [20] introduced a new construction that makes constant in the space bound for static functions *arbitrarily close to one.* In according with Calkin's Theorem [14](see theorem 1), a constant β_r exists such that if $m \geq \beta_r n$, A is a matrix of size $m \times n$ and the rows of the matrix A are drawn at random from vectors of weight r, then A has full rank with high probability. As we can see in Figure 2.3,β tends to one quickly as r increases, in opposition with the behavior of γ_r (which has a finite minimum). Thus, the denser the rows, the closer m can be to n. For example, if $r = 3$, $\beta_3 \approx 1.12 < \gamma_3 \approx 1.23$. Unlike MWHC's linear-time peeling algorithm, general matrix inversion requires super*quadratic* time ($\Theta(n^3)$ with Gaussian elimination).

Some experiments in this direction have been performed by Aumüller et al. [2]. To obtain a feasible algorithm, they shard the set S into small sets using a hash function, and compute the static functions on each subset independently. In their study, they have performed two main experiments. In the first experiment, authors keep in a look-up table some of the possible pseudo-inverse matrices for fixed sizes $n' \ll n$ and $m' \ll m$. The size of the table grows incredibly fast even for small values of n' m'. Furthermore using small values is impossible to achieve the threshold $c_{k,r}$. However, in [2] authors confirmed that using this technique the expected time required to build the function is linear. In the second experiment the authors tried to solve the linear system directly. The time required to build the function with 10^7 keys splitted in sets of size $n' = 500$ and $m' = 1.1 \cdot n'$ is about 40 minutes. Consequently, they concluded that it is possible to beat the hypergraphbased construction, but the time required for system solving makes the construction unusable (for more details, see for instance [49]).

3.2 Compressed Static Functions

3.2.1 Codes

We now introduce the basic concepts which are used widely in the next section, dealing with compressed static functions. We start from the definition of code.

Definition 5. A *code* for a set of symbols Σ is a map $c : \Sigma \to \{0,1\}^*$. The elements of $c(\sigma)$, $\sigma \in \Sigma$ are called *codewords*.

Let $\preceq$ denote the *prefix partial order*, that is, for $\mathcal{X}, \mathcal{Y} \in \{0,1\}^*$ we have $\mathcal{X} \preceq \mathcal{Y}$ iff $\exists \mathcal{Z} \in \{0,1\}^*$ such that $\mathcal{Y} = \mathcal{X}\mathcal{Y}$ The main property we will use is the following one:

Definition 6. A code is *prefix-free* if given $\mathcal{X}, \mathcal{Y} \in \Sigma$ it never happens that $c(\mathcal{X}) \preceq c(\mathcal{Y})$ or $c(\mathcal{Y}) \preceq c(\mathcal{X})$.

Given a sequence of values in Σ and a code, we can concatenate the codewords associated with each element of the sequence, obtaining a uniquely decodable sequence of bits. Our aim is to choose the code that minimizes the length of the sequence of bits.

In our case, the symbols are independent from each others and we must compress each symbol by itself. Under this assumption the space lower bound for a sequence of symbols T is given by the 0-th order empirical entropy [38]. Let $p(\sigma)$ be the number of occurences of the symbol σ in T, the 0-th order empirical entropy of T is defined by the equation:

$$H(T) = \sum_{\sigma \in \Sigma} p(\sigma) \log \frac{1}{p(\sigma)} \tag{3.2}$$

There exist some static codes, such as Elias's γ code [23], that match this bound if the frequencies of the symbols follow a specific distribution. Given a set of frequencies it is always possible to build an optimal code with space close to the lower bound. One of these codes can be computed using the Huffman algorithm. Given a set of frequencies and symbols, it creates a binary tree that represents an optimal code for those frequencies. The space usage of the Huffman code consumes at most one bit more than the empirical entropy

3.2.2 General Considerations

Having introduced these ideas, we can apply the previous concepts to compressed static functions and describe construction techniques. Although static hash functions have been deeply studied, the study of compressed static functions is a recent research direction. Since we are solving the problem of storing static functions, we will use the same notation we used in the previous section.

The questions to answer are two:

- How much can we reduce the space usage?
- How does this affect the lookup time?

We can answer the first question by saying that it depends on the list of output values V. More precisely, we want to represent a function using space close to $nH(V)$, where $H(V)$ is the 0th-order entropy as defined in Equation 3.2.From now on, when we do not specify the list of values, we identify the 0th-order entropy as H_0.

As in the case of static functions, there is a simple construction that involves a minimal perfect hash function. The first step creates a minimal perfect hash function $f : S \to \{0, 1, \ldots, n - 1\}$. Then the list of values V is encoded using Huffman coding. For each key $k_i \in S$, the corresponding codeword $c(f(k_i))$ is appended in a contiguous bit array, and the starting position of each codeword is represented using a succinct representation for monotone lists, or prefix sums, such as Elias–Fano[22, 24]. This data structure stores a monotone increasing list of n natural numbers $0 \leq x_1 \leq \ldots \leq x_{n-1} < 2^s$ for $s \in \mathbb{N}$, using $2 + s - \log n$ bits per element when $n < 2^s$ or $1 + 2^s/n$ bits otherwise. Furtermore using a modern data structure the random access is performed in constant time [54]. When we want to query the value for any key $k_i \in S$ we perform the following steps:

- compute $f(k_i)$;
- query the value for $f(k_i)$ in the prefix sum data structure;
- access the bit array in that position;
- decode a codeword.

All these steps, can be implemented in constant time using a proper data structure. However, to obtain this results this construction technique requires many space overheads. Recently two new, better solutions have been proposed to improve this simple method. The first one is suggested by Hreinsson, Krøyer, and Pagh [35] and the latter by Belazzougui and Venturini [10] .

3.2.3 BeV

Now, we present the solution proposed by Belazzougui and Venturini in [10]. The first step of their construction splits the keyset S into $m = \Theta(\frac{n \log\log n}{\log n})$ buckets using a function $G : S \to \{B_0, B_1, \ldots, B_m\}$. Each of these buckets contains at most $b = c\frac{\log n}{\log\log n}$ keys, where c is an arbitrary constant smaller than 1. Then the authors use the function $Q : S \to \{0, 1, \ldots, b^2\}$ such that all keys in the same bucket are mapped injectively. This way, the function G gives the index of the bucket and the function Q gives the offset inside the bucket. The authors assumed that both of these functions can be computed in constant time. Each bucket B_l can be represented using this equation:

$$B_l = \{(o_j, v_i) | \forall (k_i, v_i), G(k_i) = l \wedge Q(k_i) = o_j\}. \tag{3.3}$$

For each bucket, they combined the offset information and the value of the key. The next step creates a fully random $n^\epsilon \times b^2$ table TR, where ϵ is a constant smaller than 1. They look for the row index r_l, such that $TR[r_l, o_j] = v_i$ for any pair $(o_j, v_i) \in B_l$. All the row indexes are then encoded and appended in a bit array. Since the row indexes are stored by using a variable length binary code, all the starting positions need to be represented by using a prefix sum data structure. To lookup the value v_i of the key k_i they follow these steps:

- compute the bucket $G(k_i)$ and the offset $Q(k_i)$
- retrieve the starting position of the index row for the bucket $B_{G(k_i)}$ by using the prefix sum data structure
- access the table $TR[r_{G(k_i)}, Q(k_i)]$

In their analysis, Belazzougui and Venturini state that $\log^3 n$ attempts are necessary to find the suitable table. The time required to search the match for each bucket configuration is $n^{2\epsilon}b$. Thus, the overall time required is $O(n^{2\epsilon}b \log^3 n)$, which is linear for $\epsilon < 1/2$. The authors analyze the space required to store the bit array containing all the row indexes. Their analysis is based on the probability that a generic row of TR matches the bucket configuration. More precisely, the probability that the row index r_l is such that $TR[r_l, o_j] = v_j$ for every pair of the bucket configuration B_l described in Equation 3.3. At the end of the analysis, they state that the expected space required to store the bit array is bounded by the 0th-order entropy of the values.

3.2.4 HKP

The second construction technique is the one presented in [35]. This approach builds on the construction of static functions based on linear systems, and as such relies on r fully random hash functions $h_0, h_1, \ldots, h_{r-1}$.To preoceed with the (description) of this method, we assume to have built a suitable optimal (or close to optimal, see [35] for details) code $c : \Sigma \to \{0,1\}^*$ for the output values. We denote with $c(f(k_i))_d$ is the d-th bit of the codeword $c(f(k_i))$, $0 \le d < |c(f(k_i))|$.

Then we build the equations to encode each *bit* of $c(f(k_i))$, for $k_i \in S$. More precisely, the j-th bit of $c(f(k_i))$ gives rise to the equation

$$w_{h_0(k_i)+j} \oplus w_{h_1(k_i)+j} \oplus \cdots \oplus w_{h_{r-1}(k_i)+j} = c(f(k_i))_j. \tag{3.4}$$

Here the w's are variable representing the content of a bit array. Thus, we obtain a linear system on $\mathbf{F}_2$ of $\sum_{k_i \in S} |c(f(k_i))|$ equations, each with exactly r variables. Since the system is random, by the known bounds on the satisfiability of k-XORSAT instances [19, 21], which are equivalent to such linear system on $\mathbf{F}_2$, we know that we need the bit array to be $c_{2,r} \sum_{k_i \in S} |c(f(k_i))|$ bits long ($c_{2,3} \approx 1.089$, $c_{2,4} \approx 1.024$).

Note that reading the values from the bit array defined by the equations above (if any) is a very simple process: given $k_i \in S$, one computes the positions $h_j(k_i)$ in the bit array and XORs the stream of bits starting at those position. Then, one decodes the first codeword in c from the stream and returns the associated value.

We now notice that if the distribution associated with the prefix-free code c matches exactly the distribution of the values, $\sum_{k_i \in S} |c(f(k_i))|/n$ is approximately H_0, which suggests that in general the space used per key will be very close to $c_{2,k} H_0$, that is, very close to the entropy. In practice, as we will see, there will be some auxiliary data which must be stored and we will not be able to work with a truly optimal code, but both limitations have a relatively small impact on the whole data structure.

The authors of [35] show that for distributions in which the most frequent element $\bar{\sigma}$ has a frequency very close to one, it is useful to store the set of keys mapped to $\bar{\sigma}$ using a separate approximate data structure. The trick can be applied to our engineered construction orthogonally, so we will not discuss this issue further.

Applying this technique we shifted from n equations (for the non-compressed case) to $\sum_{k_i \in S} |c(f(k_i))|$ equations—an H_0-fold increase in the size of the linear system, to which an even larger increase in construction time follows, as Gaussian elimination is cubic.

Chapter 4

Improvements

Introduction

This section focuses on the original core of this thesis work. We propose an innovative engineering method for constructions on linear systems that we have originally developed. Two are the main improvementes we implemented: the first is the introduction of a broadword programming. We represent the equations of the linear system using a bit array; this makes faster row operations; more details are given in Section 4.1. We combine broadword programming with a new heuristic that tries to minimize the number of row operation required by the standard Gaussian elimination, *lazy Gaussian elimination*, which is presented in Section 4.2. These two changes have a significant impact on construction time as we will see in Table 4.2. In Section 4.3 we describe other improvements that can be applied to some specific construction techniques. For instance, in Subsection 4.3.1 we will show how we can remove the ranking required by the BPZ construction. In Subsection 4.3.3 we describe our technique for generating lengthlimited canonical Huffman codes. In Subsection 4.3.4 we will analyze the drawbacks of the “structured" hash functions proposed in [35].

4.1 Broadword Programming

Our first step towards a practical solution by Gaussian elimination is *broadword programming* [37] (a.k.a. SWAR“SIMD Within A Register”), this terminology indicates a set of techniques to process multiple values simultaneously by packing them into machine words of w bits and performing the computations on the whole words. In theoretical succinct data structures, it is common to assume that $w = \Theta(\log n)$ and reduce to subproblems of size $O(w)$, whose solution can be precomputed into sublinearsized tables and looked up in constant time(this technique is calle fourrussian technique [1]). For practical values of n, however, the space used by these tables is far from negligible; in this case, broadword algorithms are usually sufficient to compute the same functions in constant or nearconstant time without having to store a lookup table.

In our case, the inner loop of Gaussian elimination is entirely composed of row operations: given vectors x and y, and a scalar α, compute $x + \alpha y$. Moreover, when performing back substitution, we will need to compute rowmatrix multiplications, where a row is given by the coefficients of an equation, and the matrix contains the solutions computed so far.

4.1.1 First Case: Static Functions ($\mathbf{F}_2$)

In the case of static functions, the underlying field is $\mathbf{F}_2$, and it is trivial to perform row operations and process w elements at a time. More precisely the bit i of word j represent the coefficient of the variable of index $wj + i$. Since the coefficients can be 0 or 1, we can simply pack one element per bit, and since the scalar can only be 1, the sum is just a bitwise XOR (`x ^ y`, using the C notation).

To perform rowmatrix multiplications, one can iterate on the ones of the row and add up the corresponding $b$bit rows in the matrix. Iterating on the ones is easy: the position of the least significant one can be found by an constant time computation, and the bit can be deleted with the standard broadword trick `x = x & x`.[1]

4.1.2 Second Case: Minimal Perfect Hash Function ($\mathbf{F}_3$)

For MPHFs construction, the linear system is over the field $\mathbf{F}_3$, which requires more sophisticated algorithms. First, we can encode each element $\{0, 1, 2\}$ into 2 bits, thus fitting $w/2$ elements into a word. The scalar α can only be 1 or 1, so we can treat the cases $x + y$ and xy separately.

In the case we are performing an addition, we can start by simply adding x and y. When elements on both sides are smaller than 2, the result remains smaller than 3 and there's nothing to do. However, when at least one of the two elements is 2 and the other one larger than 0, it is necessary to subtract 3 from the addition result, to bring its value back to the canonical representation in$[0\,..\,3)$. Thus we need to compute a mask that is 3 wherever the result is at least 3, and then subtract it from $x + y$. Note that when the two sides are both 2 the result overflows its 2 bits ($10_2 + 10_2 = 100_2$), but since addition and subtraction modulo 2^w are associative we can imagine that the operation is performed independently on each 2bit element, as long as the final result fits into 2 bits.

```
uint64_t add_mod3_step2(uint64_t x, uint64_t y) {
  uint64_t xy = x | y;
  // Set MSB if (x == 2 or y == 2) and (x == 1 or y == 1).
  uint64_t mask = (xy << 1) & xy;
  // Set MSB also if (x == 2) and (y == 2).
  mask |= x & y;
```

[1] Most recent CPUs, in fact, have a specific instruction to clear the lowest bit.

```
    // The MSB of each 2bit element is now set
    // iff the result is >= 3. Clear the LSBs.
    mask &= 0x5555555555555555 << 1;
    // Now turn the elements with MSB set into 3.
    mask |= mask >> 1;
    return x + y  mask;
  }
```

Subtraction is very similar. We begin by subtracting elementwise y from 3, which does not cause any carry since all the elements are strictly smaller than 3. The resulting elements are thus at least 1. We can now proceed to compute $x + y$ with the same case analysis as before, except now the righthand elements are in $[1\,..\,3]$, so the conditions for the mask are slightly different.

```
  uint64_t sub_mod3_step2(uint64_t x, uint64_t y) {
    // y = 3 - y.
    y = 0xFFFFFFFFFFFFFFFF - y;
    // Now y > 0
    // Set MSB if x >= 2.
    uint64_t mask = x;
    // Set MSB if (x >= 1 and y == 2) or (y == 3).
    mask |= ((x | y) << 1) & y;
    mask &= 0x5555555555555555 << 1;
    mask |= mask >> 1;
    return x + y  mask;
  }
```

Example 4.1.1. We now show a small example of the computation for the addition between two equations. To the pourpose of the example, we limit the size of the machine word to eight. Suppose that we want to sum up these equations: $2x_0+x_1+2x_2+2x_3 = 0$ and $x_0+x_1+2x_2 = 2$. The righthand sides are just summed up. So, we will focus just on coefficients. Each coefficient is represented by 2 bit and we will use a comma to separates each pair of bits. Hence, these equations are converted into two bit sequences: $x = 10, 01, 10, 10$ and $y = 01, 01, 10, 00$. The complicated part of this computation is taking care of the carry. The possible combinations of values that produce a carry are $\{1+2, 2+1, 2+2\}$. The results of line 1 and 2 of `add_mod_step2(uint64_t x, uint64_t y)` is a mask. The most significant bit (MSB from now on) of each pair is set iff the pair (x_i, y_i) is such that $(x_i = 2 \wedge y_i = 1) \vee (x_i = 1 \wedge y_i = 2)$. Let us describe these two steps more thoroughly.

1.

$$\begin{array}{rl} & 10, 01, 10, 10 \quad | \\ & 01, 01, 10, 00 \\ \hline \text{xy} = & 11, 01, 10, 10 \end{array}$$

2.

$$\begin{array}{r l} 1{,}10,11,01,00 & \& \\ 0{,}11,01,10,10 & \\ \hline \texttt{mask} = 10,01,00,00 & \end{array}$$

In the second step we added two bits due to the right shift, but the operator is an "and" thus the blue bits can be omitted. For each pair we highlighted the most significant bit in red. After these two steps, the MSB is set to 1 just for the first pair. In Line 3 we set the MSB to 1 for each pair (x_i, y_i) such that both $(x_i = 2) \wedge (y_i = 2)$.

3.

$$\begin{array}{r l} 10,01,10,10 & \& \\ 10,01,10,00 & \\ \hline 00,01,10,00 & | \\ 10,01,00,00 & \\ \hline \texttt{mask} = 10,01,10,00 & \end{array}$$

Now we set the MSB for all the pairs with both sides equal to 2. In the fourth step we remove the spurious less significant bits using a bit mask. Then, in the fifth step, we set all the bits for the pair with a carry to 3. At last, we sum x and y and we remove 3 for all the pairs that have a carry using `mask`.

4.

$$\begin{array}{r l} 10{,}10,10,10,10 & \& \\ 00{,}10,01,10,00 & \\ \hline \texttt{mask} = 10,00,10,00 & \end{array}$$

5.

$$\begin{array}{r l} 01,00,01,00 & | \\ 10,00,10,00 & \\ \hline \texttt{mask} = 11,00,11,00 & \end{array}$$

6.

$$\begin{array}{rl} x = & 10,01,10,10 \quad + \\ y = & 01,01,10,00 \\ \hline & 11,11,00,10 \\ \texttt{mask} = & 11,00,11,00 \\ \hline \text{result :} & 00,10,01,10 \end{array}$$

The final result is the bit array 00, 10, 01, 10, which represents the equation $2x_1 + x_2 + 2x_3$. In a similar way, it is possible to subtract two equations.

Both addition and subtraction take just ten arithmetic operations and modern 64bit CPUs can execute each operation on a vector of 32 elements with one single instruction. We are left with rowmatrix multiplications.

To compute such multiplications, we use the following broadword algorithm that computes the scalar product of two vectors represented as 64bit words. The function `popcount(uint64_t x)` returns the number of bits equal to 1 in x. This instruction is common in broadword programming and has been introduced in *SSE4.2* Intel instruction set in 2007.

```
uint64_t prod_mod3_step2(uint64_t x, uint64_t y) {
  uint64_t high = x & 0xAAAAAAAAAAAAAAAA;
  uint64_t low = x & 0x5555555555555555;
  uint64_t high_shift = high >> 1;
  uint64_t t = (y ^ (high | high_shift))
               & (x | high_shift | low << 1);
  return popcount(t & 0xAAAAAAAAAAAAAAAA) * 2
       + popcount(t & 0x5555555555555555);
}
```

The expression computing t takes care of placing in a given position a value equivalent to the product of the associated positions in x and y (this can be checked easily with a casebycase analysis). We remark that in some cases we actually use 3 as equivalent to zero. At that point, the last lines compute the contribution of each product. Note that the results still have to be reduced modulo 3.

Example 4.1.2. We now show how we use the function `uint64_t prod_mod3_step2(uint64_t x, uint64_t y)` in order to back substitute solution in an equation. The equation is represented by `x`, meanwhile the solutions are represented by `y`. As in Example 4.1.1 the system is over $\mathbf{F}_3$. We thus need 2 bit to represent each coefficient and each solution. We will use a machine word size equal to six. We want to compute the solution for x_0 from the equation

$$2x_0 + x_1 + x_2 = 1 \tag{4.1}$$

and we know that $x_1 = 2$, $x_2 = 1$. We first convert the equation and the solutions into bit arrays: $10, 01, 01$ for the equation and $00, 10, 01$ for the solutions. The first two steps separate the high bits of the coefficient from the lower bits:

1.
$$\begin{array}{r l} 10,01,01 & \& \\ 10,10,10 & \\ \hline \texttt{high} = 10,00,00 & \end{array}$$

2.
$$\begin{array}{r l} 11,01,00 & \& \\ 01,01,01 & \\ \hline \texttt{low} = 00,01,01 & \end{array}$$

Then we compute the mask `high_shift`

3.
$$\texttt{high\_shift} = 01,00,00$$

The next step is a little bit more complicated and we split it in two parts: one computes `y ^ ( high | high_shift )` and the other one computes `( x |  high_shift | low < < 1)`. Then we AND the two values together. After this step we will obtain `t`.

4.
$$\begin{array}{r l} 10,00,00 & | \\ 01,00,00 & \\ \hline 11,00,00 & \hat{} \\ 00,10,01 & \\ \hline \texttt{right} = 11,10,01 & \end{array}$$

5.
$$\begin{array}{r l} 10,01,01 & | \\ 01,00,00 & \\ \hline 11,01,01 & | \\ 00,10,10 & \\ \hline \texttt{left} = 11,11,11 & \end{array}$$

6.

$$\begin{array}{r l} 11,10,01 & \& \\ 11,11,11 & \\ \hline \texttt{t} = 11,10,01 & \end{array}$$

Each pair of the value `t` represent the multiplication between the solution of a variable and its coefficient. We highlighted the value for x_0 in red, this value is 3, but in $\mathbf{F}_3$ is equal to 0 and does not influence the final result. The next step is unpacking the bitvector and summing all the pairs. To do this we separate the low bits from the high bits in each pair of bits in `t`. Then the high bits will be multiplied by 2.

7.

$$\begin{array}{r l} 11,10,01 & \& \\ 10,10,10 & \\ \hline \texttt{high\_t} = 10,10,00 & \end{array}$$

8.

$$\begin{array}{r l} 11,10,01 & \& \\ 01,01,01 & \\ \hline \texttt{low\_t} = 01,00,01 & \end{array}$$

9.

$$result = \texttt{popcount(high\_t)} * 2 + \texttt{popcount(low\_t)} = 2 * 2 + 2 = 6$$

The result can be bigger than three, so we compute *result* mod 3 and we obtain zero. Now we can substitute the result in the original equation in order to find the values of x_0. More precisely, Equation 4.1 becomes $2x_0 + 0 = 1 \rightarrow x_0 = 2$.

As suggested by Djamal Belazzougui it is possible to further reduce the number of operations required to compute the scalar product by using `prod_mod3_step2Djamal (uint64_t x, uint64_t y)`.

```
uint64_t prod_mod3_step2Djamal(uint64_t x, uint64_t y) {
  uint64_t z =x | y;
  uint64_t w = (z | z >> 1) & 0x5555555555555555;
  uint64_t t = x ^ y ^ w;
  return popcount ( t & 0 xAAAAAAAAAAAAAAAA ) * 2
              + popcount ( t & 0 x5555555555555555 ) ;
}
```

The outputs of `prod_mod3_step2Djamal` and `prod_mod3_step2` may be different, but the results are always congruent $\mod 3$. By using the code proposed by Belazzougui we can compute the scalar product $\approx 20\%$ faster.

4.2 Lazy Gaussian Elimination

Even if armed with broadword algorithms, solving by Gaussian elimination systems of thousands of equations and variables would be prohibitively slow, making the construction of our data structures orders of magnitude slower than the standard MWHC technique.(we will see that we can reduce to this case using HEM; see Section 4.3.1)

Structured Gaussian elimination aims at reducing the number of operations in the solution of a linear system by trying to isolate some variables appearing more frequently, and then rewrite the rest of the system using just those variables. It is a heuristic developed in the context of computations of discrete logarithms, which require the solution of large sparse systems [39, 45]. The standard formulation requires the selection of a fraction of variables (chosen arbitrarily) that appear in a large number of equations, and then some loosely defined refinement steps.

Here we describe a new parameterless version of structured Gaussian elimination, which we call lazy Gaussian elimination. These heuristics turned out to be extremely efficient on our systems; in fact we can reduce the number of equation in the system to be solved by standard elimination to around 4% of the original one.

Consider a system of equations on some field. At any time a variable can be *active*, or *solved* and an equation can be *sparse* or *dense*. Variables that are not active and not solved are called *idle*. Initially, all equations are sparse, and all variables are idle. We will modify the system maintaining the following invariants:

- dense equations do not contain idle variables;
- an equation can contain at most one solved variable;
- a solved variable appears in exactly one dense equation.

Our purpose is to modify the system so that all equations are dense, trying to minimize the number of active variables (or, equivalently, maximize the number of solved variables). At that point, values for the active variables can be computed by standard Gaussian elimination on the dense equations that do not contain solved variables, and solved variables can be calculated easily from the values assigned to active variables.

The *weight* of a variable is the number of sparse equations in which it appears. The *priority* of a sparse equation is the number of idle variables in the equation. The priority determines if an equation contains variables that can be converted from idle to solved. Lazy Gaussian elimination keeps equations in a minpriority queue and performs the following actions:

1. If there is a sparse equation of priority zero that contains some variables, it is made dense. If there are no variables, the equation is either an identity, in which case it is discarded, or it is impossible, in which case the system is unsolvable, and the procedure stops.
2. If there is a sparse equation of priority one, the only idle variable in the equation becomes solved, and the equation becomes dense. The equation is later used to eliminate the solved variable from all other equations.
3. Otherwise, the idle variable appearing in the largest number of sparse equations becomes active.

Note that if the system is solvable the procedure always completesin the worst case, by making all idle variables active (and thus all equations dense).

We can observe that:

- The weight of an idle variable never changes, as in step 2 we eliminate the solved variable and modify the coefficients of active variables only. It means that initially we can simply sort (e.g., by countsort) the variables by the number of equations in which they appear, and pick idle variables in that order at step 3.
- We do not need a priority queue for equations: simply, when an equation becomes of priority zero or one, it is moved to the left or right side, respectively, of a deque that we check in the first step.

Thus, the only operations requiring superlinear time are the eliminations performed in step 2, and the final Gaussian elimination on the dense equations, which we compute, however, using broadword programming.

Example 4.2.1. We now give an example of how lazy Gaussian elimination works. We start from this system of equations on $\mathbf{F}_2$ and $\oplus$ is a bitwise XOR operation:

$$\begin{aligned} v_0 \oplus v_1 \oplus v_3 &= 1 \\ v_0 \oplus v_1 \oplus v_2 &= 0 \\ v_1 \oplus v_2 \oplus v_4 &= 1 \\ v_2 \oplus v_3 \oplus v_4 &= 1 \end{aligned} \tag{4.2}$$

From the system, we compute the weights of the variables. $w_0 = 2$, $w_1 = 3$, $w_2 = 3$, $w_3 = 1$, $w_4 = 1$. All the variables are set as *idle*. The priority of an equation is the number of *idle* variables: $p_0 = 3$, $p_1 = 3$, $p_2 = 3$, $p_3 = 3$. Initially all the equations are sparse and we don't have any *solved* or *active* variable. Since there are no equations with priority zero or one so we move to step 3. We mark the variable v_1 as active, and change the priorities of the equations in which v_1 appears, $p_0 = 2$, $p_1 = 2$, $p_2 = 2$, $p_3 = 3$. As previously done, step 1 and 2 are skipped. We mark as active the variable v_2, and we decrease priorities p_1, p_2 and p_3. Now $p_1 = p_2 = 1$. We choose to mark the variable v_4 from the third equation as solved. We also decrease the priority of the third equation and we mark it as dense. Now we substitute the variable v_4 in the fourth equation. The system is now the following:

$$\begin{aligned} v_0 \oplus v_1 \oplus v_3 &= 1 \\ v_0 \oplus v_1 \oplus v_2 &= 0 \\ v_1 \oplus v_2 \oplus v_4 &= 1 \\ v_2 \oplus v_3 \oplus v_1 \oplus v_2 &= 1 \end{aligned}$$

From now on we highlight the *dense* equations in red. The last equation can be further reduced, since $v_2 \oplus v_2 = 0$, and then $p_3 = 1$. We proceed with step 2. We mark the variable v_3 as solved. The set of *solved* variables is $\{v_3, v_4\}$, the *idle* variable $\{v_0\}$ and *active* are $\{v_1, v_2\}$ and the set of *dense* equations consists of the third and fourth equations..

$$\begin{aligned} v_0 \oplus v_1 \oplus v_3 &= 1 \\ v_0 \oplus v_1 \oplus v_2 &= 0 \\ v_1 \oplus v_2 \oplus v_4 &= 1 \\ v_3 \oplus v_1 &= 1 \oplus 1 = 0 \end{aligned}$$

Then we substitute the solved variable v_3 in the first equation and we obtain $v_0 \oplus v_1 \oplus v_1 = 1$. From this equation we can mark the variable v_0 as *solved*, and we substitute $v_0 = 1$ in the second equation. So the system looks like this:

$$\begin{aligned} v_0 &= 1 \\ v_1 \oplus v_2 &= 1 \\ v_1 \oplus v_2 \oplus v_4 &= 1 \\ v_3 \oplus v_1 &= 0 \end{aligned}$$

Now that the procedure is over, let us recap all the configurations:

- *solved* variables $=\{v_0, v_3, v_4\}$
- *active* variables $=\{v_1, v_2\}$
- *idle* variables $\emptyset$
- all equations are dense; only the second equation contains just active variables.

Now it is time to use Standard Gaussian elimination on the set of dense equation containing active variables only. In this case we have one equation and two variables, thus we can set one variable to zero and then solve the other one. So we set $v_1 = 0$, thus $v_2 = 1$. Then, we easily compute the solved variables, since they depend on active variables only . We obtain $v_3 = 0$. and $v_4 = 0$.

4.2.1 An Experimental Evaluation of Lazy Gaussian Elimination

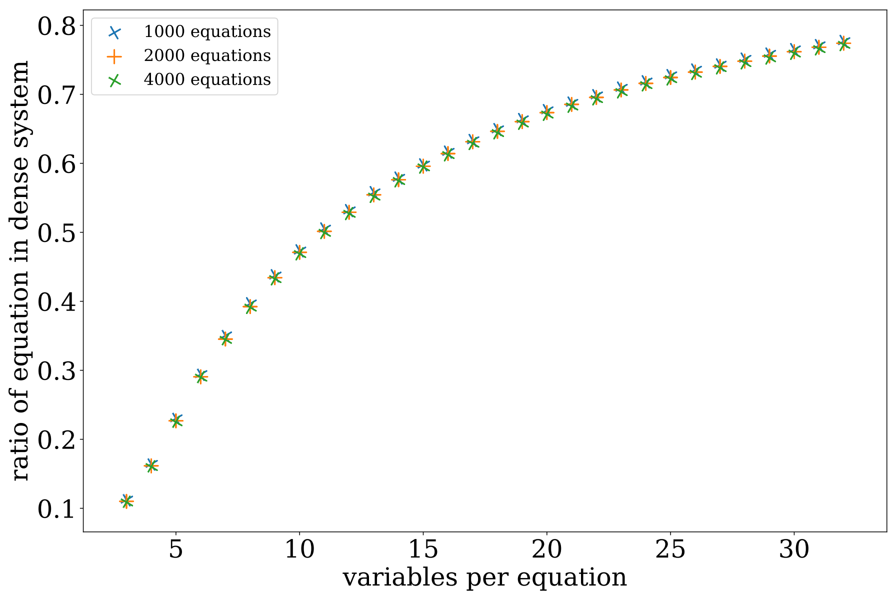


Figure 4.1: Ratio between the reduced and original systems as a function of the number of variables per equation.

In this section we briefly analyze experimentally the effectiveness of lazy Gaussian elimination. More precisely, we want to know the size of the reduced system (i.e., the number of dense equations containing active variables only) for different system sizes and density. In particular, we test systems with $n \in \{1000, 2000, 4000\}$ equations and variables, and generate random systems with $r \in \{3 \,..\, 33\}$ variables per equation.

In Figure 4.1 we plot the ratio between the number of equation in the reduced and original system. Since we have to solve the reduced system by Gaussian elimination, we focus just on small values of r for which the ratio is below 20%.

4.3 Minor Improvements

n this section, we present some special improvements that can only be applied to a specific type of functions, differently from the previous ones, which have general validity.

4.3.1 HEM

One of the main practical issues we find when dealing with big datasets is the amount of main memory that we use during the construction. A simple solution is the classical *divide and conquer* technique as proposed in [13]. This solution, which we call henceforth *HEM* (Heuristic External Memory algorithm), can be implemented orthogonally to almost all construction techniques. The dataset is divided into fixedsized chunks by hashing, and for each chunk, we compute the solution for the linear system. The solutions coming from each chunk are concatenated in order.The number of values in the previous chunk and the seeds of the random generator are stored within the solutions. When we want to find the value for a key, we compute its chunk, then retrieve the offset of the values of the corresponding function, and finally compute the value of that key. The amount of time that we spend to compute the chunk for each key is limited, and it can be hidden under the key loading time.

Our HEM version uses *ondisk bucket sorting* to speed up construction: keys are first divided into 256 ondisk *physical* chunks, depending on the highest bits of their hash value (we use Jenkins's SpookyHash). The ondisk chunks are then loaded into memory and sorted, and virtual chunks of the desired size are computed by either splitting or merging physical chunks. Since we store a 192bit hash plus a 64bit value for each chunk, the upper limit for the keynumber dependent amount of memory used is one bit per key. This value does not include the memory required for the structure to be computed.

Note that in the case of compressed functions we will use a single code for all values. Thus, if we focus on a single chunk, the code we are using is no longer optimal, but this does not change the probability of solving systems. In fact the system size depends only on the sum of the codewords representing values actually in the chunk, and does not depend on the optimality of the code. By concatenating the bit arrays of all chunks, we get the same space we would have used without HEM. Note that the approach of having a local prefixfree optimal code per chunk would never work, as the space used to store the locacessary In the case of minimal perfect hashing, we can further speed up the structure and reduce space by getting rid of the ranking structure that is necessary to make the perfect hashing computed by the system of equations minimal.

In the standard HEM construction, the number of vertices associated with a chunk of size n' is given by $\lceil cn' \rceil$, where $c \in \mathbb{R}$ is a the ratio between nodes and edges, and the offset information contains the partial sums of such numbers.

We will use a different approach: the number of vertices associated with the chunk will be $\lceil c(N' + n') \rceil \lceil cN' \rceil$, where N' is the number of elements stored in previous chunks. The difference with $\lceil cn' \rceil$ is at most one, but using our approach we can compute, given N' and n', the number of vertices associated with the chunk.

Thus, instead of storing the offset information for each chunk, we will store the number N' of elements stored in previous chunks. The value can be used as a base for ranking inside the chunk: this way, the ranking structure is no longer necessary, reducing both the space and the number of memory accesses. When $r = 3$, as it is customary, we can use two bits for each value, minding to use the value 3, instead of 0, for the hinges. As a result, ranking requires just counting the number of nonzero pairs in the values associated with a chunk, which can be performed again by broadword programming:

```
int count_nonzero_pairs(uint64_t x) {
    return popcount((x | x >> 1) & 0x5555555555555555);
}
```

Example 4.3.1. Consider for instance the bit array of solutions $1111, 1111011001111111$. For simplicity, we take $c = 1$. The array is divided in $chunk_0, chunk_1$ and separated by ",". We store these values $[0, 2, 10]$. From these values we know that in $chunk_0$ there are 2 values, and in $chunk_1$ the remaining 8. We want retrieve the value of the key "a" such that:

- it belongs to $chunk_1$
- $h_0(\text{“a”}) = 0$, $h_1(\text{“a”}) = 4$, $h_2(\text{“a”}) = 6$

Since the key has been mapped into $chunk_1$ we will use the perfect hash function g_{chunk_1}. We compute the perfect hash function inside $chunk_1$ like this: $f(\text{“a”}) = h_i(\text{“a”})$, where $i = g_{chunk_1}(0) + g_{chunk_1}(4) + g_{chunk_1}(6) \mod 3 = 1 \implies f(\text{“a”}) = 4$. We now count the nonzero pairs in the red part of this bit array $1111, 1111011001111111$.

$$\begin{array}{ll} 1111011001 & | \\ 0111101100 & \\ \hline 1111111101 & \& \\ 0101010101 & \\ \hline 0101010101 & \end{array}$$

Now we use the function population count on the bit sequence 0101010101 and we obtain 5. So, we sum the result with the offset 2 and we obtain 7.

4.3.2 Compacting Offset and Seeds

After removing the ranking structure, what is left to do is storing the partial sums of the number of keys per chunk, and the seed used for the chunk hash function. This is the totality of the overhead imposed by HEM.

Instead of storing these two numbers separately, we combine them into a single 64bit integer. The main reason that allows us to do so is that due to the extremely high probability of finding a good seed for each chunk, few random bits are necessary to store it: we can just use the same sequence of seeds for each chunk, and store the number of failed attempts before the successful one. Assuming that the probability that the random linear system is solvable is p, we remark that this probability is high since we are above the threshold $c_{k,r}$. The probability of at least one success in k trials is $1(1p)^k$, and in our experiments we always obtain a success in less then 24 trials. For convenience we save 8 bits for the seed, thus we use the remaining 54 bits for storing the partial the partial sums of the number of keys per chunk, which is more than sufficient for any realistic value n.

4.3.3 Codes

In Section 3.2 we stated that we can compress the list of values using a Huffman code. Using the Huffman encoding, however, poses two problems:

- with the classical tree decoder it is impossible to decode a codeword in constant time;
- for skewed distributions the longest generated codeword can be quite long.

So the worst case decoding time depends on the length of the longest codeword. The solution proposed in [35] works as the follows: a subset of most frequent elements of Σ is actually stored using an optimal length-limited code, together with an additional symbol $\perp$ accounting for the rest of the elements. Then, the non-frequent elements are stored explicitly after $\perp$ using a suitable table. Essentially, one trades the optimality for a small cost in space and faster decoding time.

A fast algorithm for constructing an optimal code with a bound on the length of the longest codeword was introduced in [40]. The proposed solution uses the packagemerge algorithm, which is a greedy algorithm that finds the solution in time $O(nL)$, where n is the number of symbols and L the maximum final length. In [43] the authors suggest a tight bound for the inefficiency of such code. They also propose an $O(n)$ time and space algorithm to find an approximate solution. All the codewords longer than L are arranged in a complete binary tree of depth $\bar{L}$; subsenquently, a new node is added at depth $L\bar{L}1$ and the left child will be the old tree, and the right child will be the new tree.

The classical decoding approach for a Huffman code for a bit sequence adopts a walk from the root of the tree to a leaf. For each bit of the sequence, this process chooses the left subtree if the bit is a 0 else it chooses the right subtree. When a leaf is encountered the process ends, and the label of the leaf is the symbol corresponding to that encoding sequence. After a brief test we concluded that this approach was inappropriate, we will give more details in 7.2. So the decoding process based on binary tree had some effect on the lookup time, we decided to use a different practical approach based on canonical Huffman codes [51]. Canonical Huffman codes are based on the observation that among many equivalent Huffman decoding trees, there is one that is canonical, which is the one in which the depth of leaves from left to right is nondecreasing. A table made of two lists of integers of the same length can describe the structure of such a tree; the first list contains lengths, and the second list, in parallel, the number of codewords of that length. Codewords of the same length are consecutive integers, and, more in general, if we leftalign by zero padding all codewords the resulting values are increasing as the symbol frequency decreases (a property that we will use for fast decoding). Indeed, it is possible to decode a canonical code in time proportional to the length of the lists, rather then to the length of the keywords [34]. We also need to store the symbols ordered from left to right as they appear in the tree, as we need to map each codeword to the original symbol.

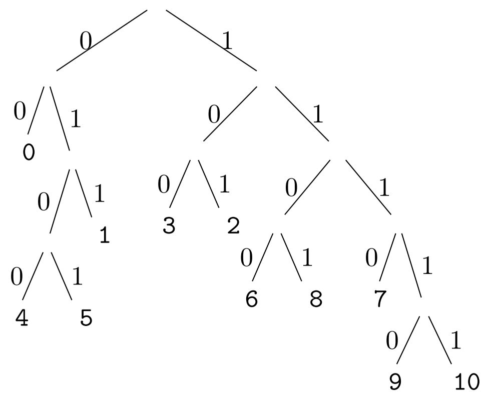
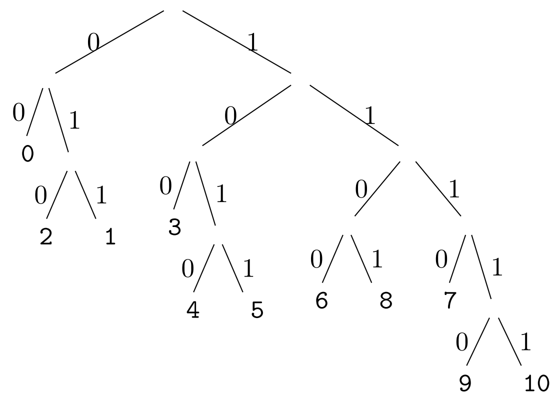
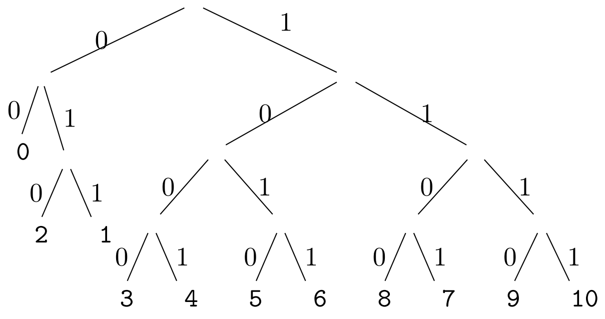


Figure 4.2: Huffman Tree for C (see Table 4.1).



number of codewords	length
1	2
3	3
5	4
2	5

Figure 4.3: Huffman tree for C', and the associated table representation (see Table 4.1)



number of codewords	length
1	2
2	3
8	4

Figure 4.4: Huffman tree for C_L ($\ell = 2$), and the associated table representation (see Table 4.1).

Example 4.3.2. Assume we have the set of symbols and frequencies present in Table 4.1.

In Figure 4.2 we show the tree for an optimal code for the list of symbols and the frequencies in the first two columns. In Figure 4.3 we see the canonical Huffman tree version of the previous Huffman tree. Both codes are optimal since the length of the codeword for the same symbols is equal.

Symbol	Frequency	C	C' (canonical)	C_L (tablelimited)
0	$\frac{17}{52}$	00	00	00
1	$\frac{6}{52}$	011	011	001
2	$\frac{6}{52}$	101	010	010
3	$\frac{5}{52}$	100	100	1000
4	$\frac{4}{52}$	0100	1010	1001
5	$\frac{3}{52}$	0101	1011	1010
6	$\frac{3}{52}$	1100	1100	1011
7	$\frac{2}{52}$	1110	1110	1110
8	$\frac{2}{42}$	1101	1101	1101
9	$\frac{2}{52}$	11110	11110	1100
10	$\frac{1}{52}$	11111	11111	1111

Table 4.1: Data for Example 4.3.2. For each symbol, we show the frequency, the associated codeword in an optimal code C, the associated codeword in the canonical optimal code C', and the associated codeword in the tablelimited canonical code C_L ($\ell = 2$).

We adopted a different construction to limit the length of the code, that we call *tablelimited canonical Huffman code*. Our solution is based directly on the table representation of a canonical code, rather than on the decoding tree. We first set a maximum table length ℓ and a fraction $0 \leq \alpha \leq 1$ (typically, $\alpha = 0.99$). We assume to know, for each symbol, its cardinality in the output values of the function we want to compress.

With a first pass on the table representation, we can compute the sum of the lengths of the codeword representation of the function values. Then, we walk down the table estimating the sum of the lengths of symbols belonging to an initial segment of the table. At each entry of the table we check whether the accumulated length exceeds α as a fraction of the total length. We stop in any case at the entry ℓ.

At the end, we know that we have to limit the table at the entry $\bar{\ell} \leq \ell$ (the inequality happens for very skewed distributions, such as the geometrical distribution, in which very few entries of the table are sufficient to code almost all values). Thus, we sum the cardinality of all the symbols represented after the entry $\bar{\ell}$, and we redistribute the cardinality uniformly among all such symbols (if the division is not exact, we privilege symbols appearing before in the table).

Finally, we recomputed the canonical code using the new cardinalities. Clearly, the new table cannot be longer than $\bar{\ell}+2 \leq \ell+2$. Thus, for a fixed ℓ, decoding happens in constant time. The effect of the construction is that of flattening all codes represented after entry $\bar{\ell}$ in the original table.

Example 4.3.3. In Figure 4.4 we show the tablelimited code for Example 4.3.2. The parameter ℓ is set to 2. As you can see, the code is no longer optimal (for example, the length of the codeword for `3` is now longer).

Finally, we remark that in our setting we can always read enough bits to decode the longest codeword in one operation. Usually, bits are coming from a source stream, and decoding a canonical code require a number of logical operations and some reading from the input stream, depending on the entry of the representation table in which the code is located [34]. In our case, we can just precompute the first codeword for each table entry and left-align it with the longest codeword; decoding is then just down by scanning a table of increasing values until the value fetched from memory is dominated by the current entry.[2] A few simple arithmetic operations complete the decoding.

4.3.4 Random Uniform Hash Functions

After the first implementation of the construction proposed in [35], we experimented a particular behavior of the linear system using HEM. From now on, we fix a *chunk log size* n' in HEM. In practice, n' is the number of higher bits used to divide keys into chunks. The average size of a chunk will be $2^{n'}$.

The construction proposed in [35] uses a special form for the hash functions $h_j, 0 \leq j \leq k1$:

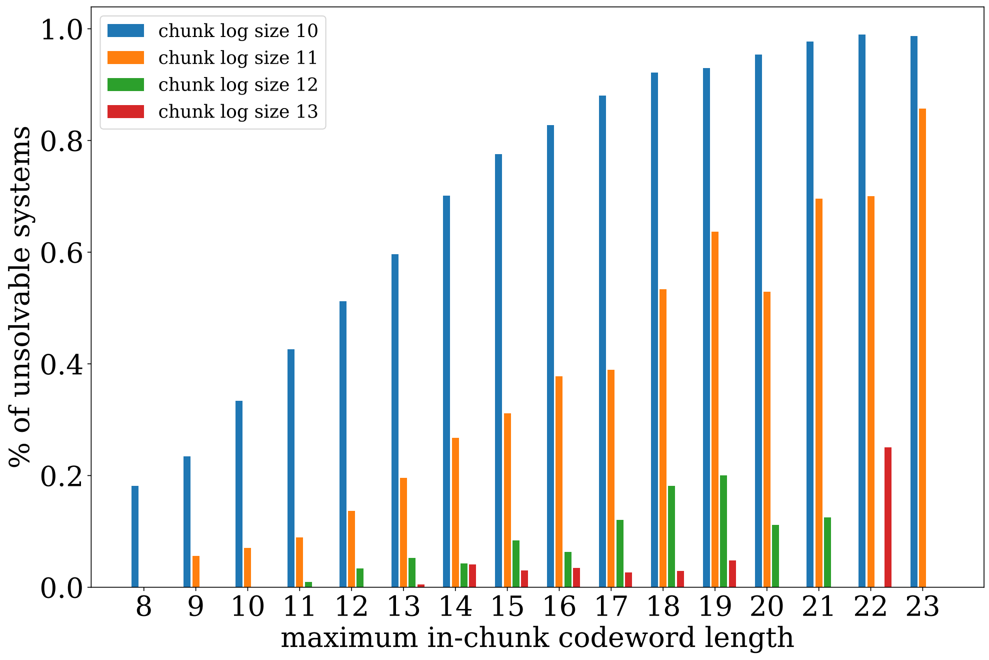
$$h_j(k_i) = h'_j(k_i)w_l + q(k_i),$$

where $k_i \in S$, $h'_{(\cdot)} : S \to \lceil m/w_l \rceil$ and $q : S \to w_l$ are fully random hash functions, and w_l is the length of the longest codeword inside the lth chunk. This choice looks promising, for each chunk it creates at most w_l *strongly connected components* (SCC) such that each node in the SCC is congruent modulo w_l to the others. Thus, we obtain w_l small systems with independent sets of variables. This has two practical advantages:

- we can use parallel solving.
- variable index compression.

We found that the first approach is detrimental in practice. Moreover, HEM can be easily parallelized without effort (each chunk can be analyzed independently), so it is not necessary

[2]Of course, in principle a binary search would provide asymptotically logarithmic decoding time. However, due to practical values of ℓ being very small, the complex logic of such a search yields results that are never competitive with a linear search. Even more complex techniques are described in [27].



!htbp

Figure 4.5: Fraction of unsolvable systems in a test dataset with geometric value distribution, $k = 3$, $\delta = 1.21$, depending on the maximum codeword length in the chunk.

to exploit parallelism. The latter combined with broadword programming can be advantageous. The next step reduces the index of the variable subtracting the residue $\mod w_l$ and divide the result by w_l. Using smaller indexes, the bit array representing each equation is smaller. Thus row operations are faster.

The problem, which is hard to detect from the theoretical side, is that the kXORSAT bounds we are using have a different asymptotic behavior than, say, acyclicity thresholds for random graphs. The asymptotic behavior starts becoming typical at much larger values (in the acyclicity case, just a few dozens of vertices are sufficient). Since we are solving linear systems chunk by chunk, but we are using a global code, some chunks happen to have a quite large maximum codeword w_l, in particular in the case of skewed distributions. For these chunks, the size of the systems and the number of variables are both scaled down by a factor of w_l, bringing them out of the "good" region. In Figure 4.5 we show the fraction of the unsolvable system with respect to the length of the maximum codeword (for a standard Huffman code) in a chunk for a dataset with geometric distribution with parameter $p = 0.5$ (the worst case): the described behavior is evident. In particular, with the smallest chunk size, finding solvable systems becomes almost impossible. Another technical issue is that the length of the longest keyword in each chunk must be stored, and m/w_l needs to be computed at query time. In the experimental part, we will show the results that motivate our decision to fall back and use the same random hash function we utilized for the other constructions.

4.4 Conclusions

In this chapter, we presented different improvements that can be added to the construction techniques based on linear system solving. These allowed us to make the construction proposed by Dietzfelbinger and Pagh [20] feasible. In 4.2 we discussed the effect of each major improvement to the time of the construction technique. This experiment was made using the MWHC construction using $r = 3$, $c_{2,3} = 1.10$ and log chunk size $n' = 10$. We determined different classes: just peeling process (P), broadword computation (B), lazy Gaussian elimination (G) or a combination of these. When we do not use lazy Gaussian elimination, we use standard Gaussian elimination. Without broadword computation, we use a sparse representation of the system with arraylist of arraylist. As we can see in 4.2 the best performance is obviously given by a combination of all

All	BG	GP	G	BP	B	P	None
0%	+13%	+57%	+98%	+296%	+701%	+2218%	+5490%
best							worst

Table 4.2: Time deterioration without each technique for the noncompressed case.

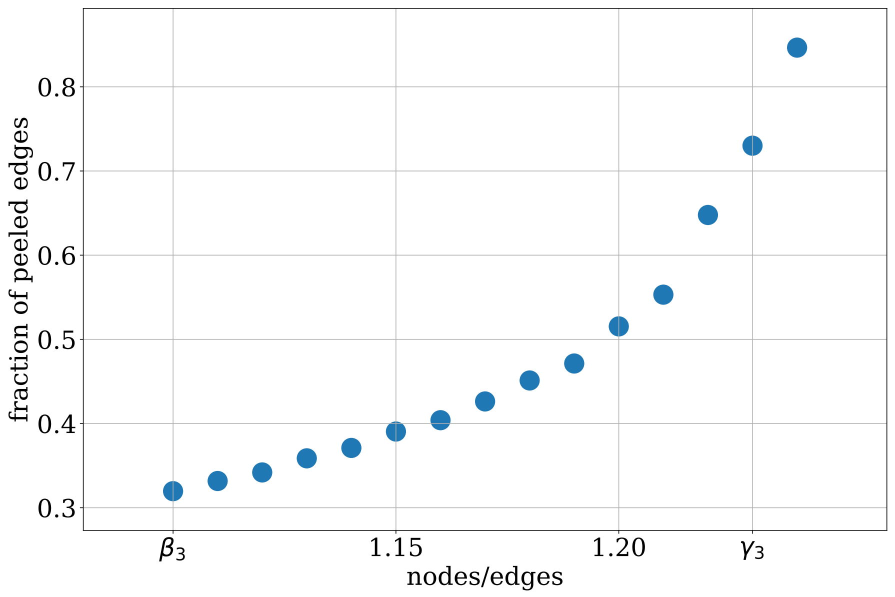


Figure 4.6: Fraction of peeled edges for a random 3hypergraph with 1024 edges.

three improvements. However, the impact of the lazy Gaussian elimination is clearly essential. This improvement is present in all the best results.

We experimented how the size of the 2core varies depending on the ratio between nodes and edges. In Figure 4.6 we show that the number of peeled equations grows quickly as we increase the number of variables. The more close we get to γ_3, the more equations are peeled.

The effectiveness of lazy Gaussian elimination is also due to the fact that the number of equations remaining in the nonempty 2core after the peeling procedure is high, as we shown in Figure 4.6. This is also confirmed by the poor performance of the peeling process when used by itself.

Part II

Experiments

Contents

Introduction

In this part of the thesis, we will present experimental results for different construction techniques based on linear systems. We start the analysis of results from static functions. We will compare the results obtained using the classical MWHC construction and the ones obtained with the improvements described in Chapter 4. To build static hash function, we will use both $r = 3$ and $r = 4$. We will see how lookup time increases depending on r.

Then, in Chapter 6 we will move to minimal perfect hash functions. We add our improvements also to the BPZ construction technique. We will compare the results for minimal perfect hash functions created using MWHC, BPZ, and HDC. For HDC construction technique we use the available code on the website of the project.

The last chapter is about compressed static functions. We describe the datasets, and we will analyze the results obtained for each dataset. We will also show that for skewed datasets the construction is scalable, and suggest that one can switch to MWHC-style triangulation otherwise.

Datasets

We performed experiments using two datasets derived from the `eu-2015` crawls gathered by BUbiNG [11]. The smaller dataset is the list of hosts that contains 11 264 052 keys. Each key is long ≈ 22 bytes. The larger dataset is the list of pages with 1 070 557 254 keys; in this case, keys are longer, that is, ≈ 80 bytes. The crawl data is publicly available at the LAW website[3]. We will see later that the length of keys has a significant impact on lookup speed.

The dataset is stored on secondary memory in plain text format. The sizes are about 80 GB for the list of URLs and 233 MB for the list of hosts. In Chapter 7 we will analyze the performance of compressed static functions, and we will describe the list of values associated with each key.

We also measure the scalability of our implementation. We create subsets of the URLs dataset for different sizes $\{2^i \mid 22 \leq i \leq 27\}$. We can state that construction is scalable if the construction speed and the space usage per key are constant.

[3]`http://law.di.unimi.it/`

Experiments

In this section, we will describe how we perform the experiments. The language we used is Java 8. The latest improvements of the JVM made bit manipulation very fast, and the performance is similar to faster language such as C/C++. The only experiment done using a different language is the one involving HDC construction. For that specific experiment, we used the implementation proposed by the authors written in C. Each experiment is performed on an Intel® Core™ i7-7770 CPU @3.60GHz (Kaby Lake), with 64 GiB of RAM, and Linux 4.10.12.

For each construction technique we measure:

- the construction speed;
- the final space required to store the function on disk;
- the lookup speed.

The time required to build and store the function is measured using the Bash command `time`. This command outputs three different times *real*, *user* and *sys*. For all the experiments we used the *real* time value, or wall time, that is, the time from the start to the end of the command execution. We decided to use to not measure timing inside the application because we want to include starting overhead for the JVM and the time required to load the keys from the datasets and arrange them using HEM. We express the time necessary for the construction as the time needed for each key. More precisely we divide the time required by the number of the elements in the keyset. We report the number of microseconds per key.

In our implementation, each data structure is stored using object serialization. Each object in Java that implements a particular interface can be stored as a sequence of bytes. The sequence of byte includes all the object data and all the information about the type of the object. All serialized objects can be loaded from secondary memory and used. We measure the size of the final file. We use as a unit of measurement bits per key. We are conscious that this is not the real size of the data structure and may contains some redundancy given by object serialization, but we think that is more fair to use this measure for experiment space usage in real case, indeed once the file is stored it contains all the data required to perform lookup queries. Furthermore is easier to compare different implementation.

Our data structures are designed to perform random access in constant time. So we fix the number of accesses that we want to perform, then extract a subset of the keyset, and shuffle it. Once fixed the subset, we perform 13 rounds; in each round, we access all the keys of the shuffled subgroup. We use the first three rounds as warm-up rounds for the just-in-time compiler. Then we measure the time required for each round, and at the end, we compute the average of the results. If the cardinality of the subset is at least 10^7, then the variance of the results for each round is small. It is important to notice that in this case, we do not measure the time required to load the object from the secondary memory and the time necessary to pre-process the dataset.

	P	C	A	Total
HOST (ns/key)	47.62	≈ 9.59	≈ 46.08	103.29
URLS (ns/key)	105.45	≈ 17.94	≈ 150.28	273.67

Table 4.3: Lookup speed contribution for each unit

The lookup time is measured in nanoseconds per key. For simplicity we divide the lookup time in four units operation:

- (P)rocess the string
- (C)ompute three indexes of the variables in the equation
- (A)ccess three times the data structure (cache miss)

These units are common for all the construction; furthermore the minimal perfect hash functions and compressed static function require one more operation: the ranking for the former, decoding process for the latter.

Table 4.3 shows the impact of each unit using GOV3Function class. The time required by the first unit depends on the length of the keys, URLs are typically longer than than host names, as we can see the impact in the first column of Table 4.3. Access to the bitvector depends just on its final length.

For compressed static function we notice that decoding the canonical Huffman code has a very marginal impact, as it requires $\approx 10\,\text{ns}$ or less ($\approx 2\,\text{ns}$ in the uniform case, as the decoding table contains just one entry).

Parameters

In every construction technique based on linear system solving, few parameters can be tuned. We can decide the size n' of each chunk of the HEM (see Section 4.3.1). We can also fix the ratio between the number of nodes and the number of hyperedges, as we saw in Section 2.4, to speed up computation. Only for compressed static functions, we can set the maximum table length ℓ and the fraction α as explained in Subsection 4.3.3. We will see that all parameters have a significant impact on the construction speed and final space, but they have a small impact on lookup speed.

Chapter 5

GOV3 and GOV4

5.1 Introduction

In this chapter, we will analyze the results obtained by appling our improvements to construction for static functions, and in particular focunsing on the optimization of the chunk size. We will use the following implementations:

- MWHC;
- GOV3Function;
- GOV4Function.

These classes are available as part of the open source Java project Sux4J.

The MWHC function is the same as the one presented in 3.1.2. GOV3Function is our data structure based on linear systems (see Section 3.1.3) plus the improvements described in 4. As indicated by its name, this implementation uses random 3-hypergraphs. GOV4Function has the same improvements of GOV3Function but uses random 4-hypergraphs.

5.2 Results

The first analysis that we performed concerns the choice of the chunk size. This parameter defines the number of keys for each bucket, that is, the number of equations for each system. This setting influences both construction speed and space usage. In general small buckets require less time to be processed. On the other hand, smaller buckets need more space. Remember that for each bucket we build a static function, and that each of these has some extra information to store. Both GOV3Function and GOV4Function create a data structure that can represent for any static function. In all experiments, we store the function $f(k_i) = i$, which implies that we store exactly one integer equal to the dataset size minus one.

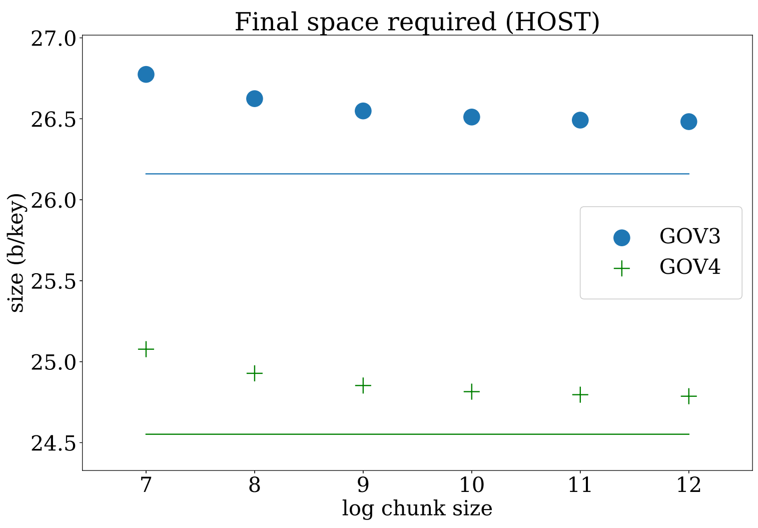


Figure 5.1: Space usage for GOV3 and GOV4 on host dataset

It is interesting to see how the measures of interest vary with the chunk size. Figure 5.1 shows how the number of bits per element and construction time varies with the chunk size for GOV3 and GOV4. To avoid being too close to the threshold, we use the constant 1.10 in the case $r = 3$

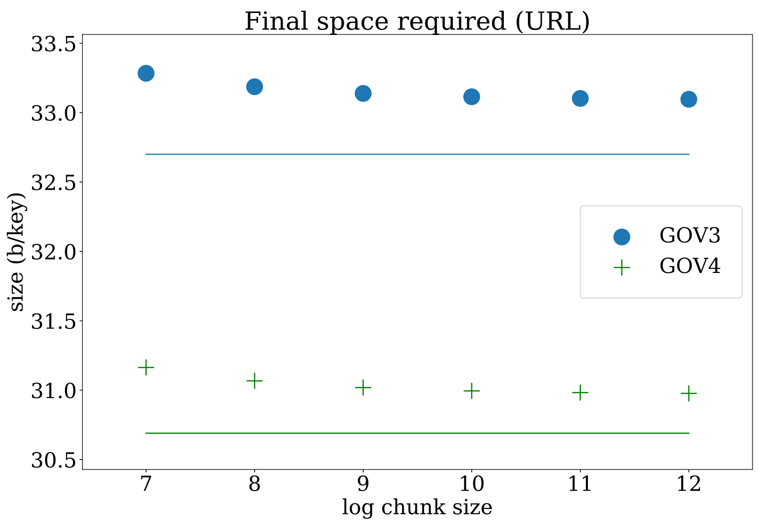


Figure 5.2: Space usage for GOV3 and GOV4 on URL dataset

and 1.03 in the case $r = 4$ (the thresholds are ≈ 1.09 and ≈ 1.023, respectively). In Figure 5.1 and Figure 5.2 we highlight the theoretical space usage with a line. The theoretical space usage represent the thoeretical size of the array of bits and it is computed using this formula:

$$c_{2,r} \cdot \log(m)$$

where m is the maximum value of the output values. As you can see the theoretical space usage is not influenced by the HEM size. We remark that the difference between theoretical value and file size is at most 0.61 bits/key and 0.52 bits/key for GOV3Function and GOV4Function rispectively. In real case, as chunks gets larger the number of bits per key slightly decreases (as the impact of the offset structure is better amortized); at the same time in Figure 5.3 we can see that construction time increases because the Gaussian elimination process is superlinear (very sharply after chunk size 2^{11}).

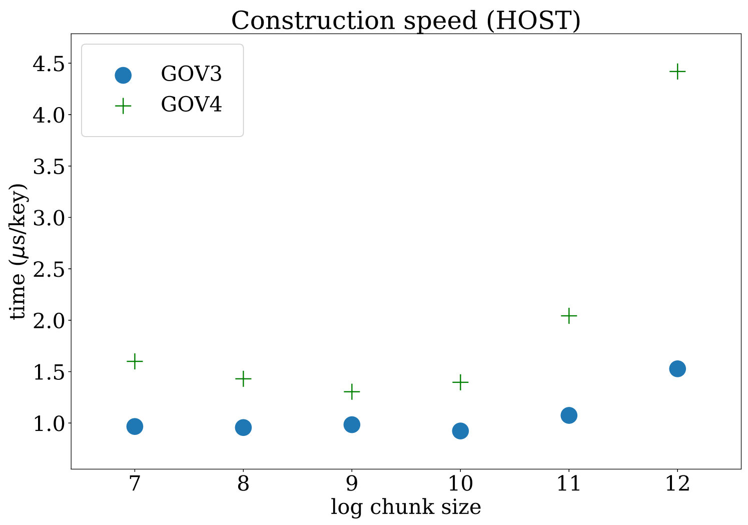


Figure 5.3: Construction time for GOV3 and GOV4 on host dataset

In Figure 5.3 we can see the behavior of construction speed. For both GOV3Function and GOV4Function, for chunk size equal to 2^7 we obtained the best result. Of course, GOV3Function is in general faster than GOV4Function (as we have seen in Figure 4.1, lazy Gaussian elimination is less efficient when the number of variables per equation increases, as highlited in fig 5.4.). The average gap between GOV3Function and GOV4Function is 0.96μs/ key, but it becomes evident for chunks bigger than 2^{11}. In both datasets, the worst case the GOV4Function is about four times slower than the best case. Nonetheless, the worst case for GOV3Function is about just 1.5 times slower.

In Figure 5.5 we show the lookup speed for host dataset, it is always constant when the chunk size varies. Obviously, GOV3Function is faster than GOV4Function, because the latter requires one additional cache miss. Values are quite stable for both constructions. As it is quite evident,

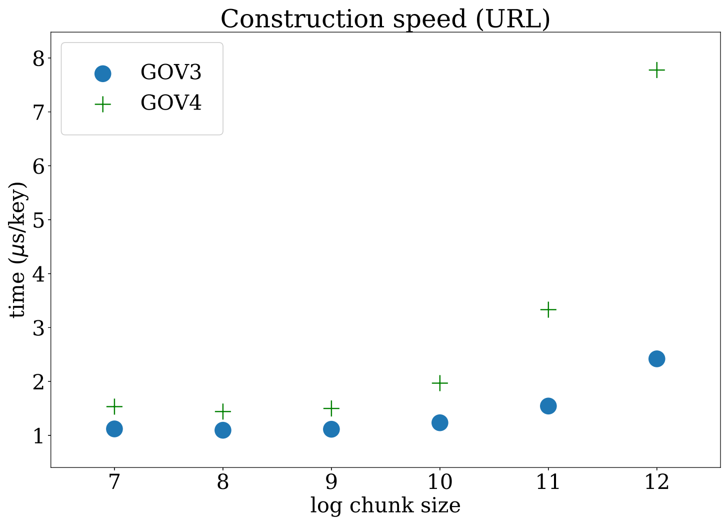


Figure 5.4: Construction time for GOV3 and GOV4 on URL dataset

the chunk size has basically no impact on this measure. Similar results are obtained for the URL dataset and displayed in Figure 5.2, 5.4 and 5.6.

It is interesting to notice that using chunks larger than 2^{10} yield a slightly improved lookup time, as the offset array becomes small enough to fit into the L3 cache. This phenomenon occours only for big datasets because for small datasets the offset array always fits the in the L3 cache. It explains why in Figure 5.6 we have better performances increasing the chunk size.

Summarizing the analysis preented so far, we see that for chunk size 2^{10} we obtain the best results in terms of space usage and construction time. Indeed construction time increases rapidly for chunks bigger than 2^{10} and the space usage decrease slowly. So we decided to fix the chunk size to 2^{10}. Finally we can compare GOV3Function and GOV4Function with a classical hypergraph-based MWHCFunction. In Table 5.1 we report all measurement for both datasets.

	Host			URLs		
	MWHC	GOV3	GOV4	MWHC	GOV3	GOV4
Lookup (ns/key)	184.08	160.68	171.57	368.01	320.73	339.05
Construction ($\mu s/key$)	0.46	0.92	1.39	0.89	1.23	1.97
Size ($bits/key$)	29.80	26.51	24.81	37.17	33.11	30.99

Table 5.1: A comparison of per-key construction time, lookup time and size for static function implementation.

As expected, the space per key required by the classic MWHC construction is bigger than both GOV3Function and GOV4Function.

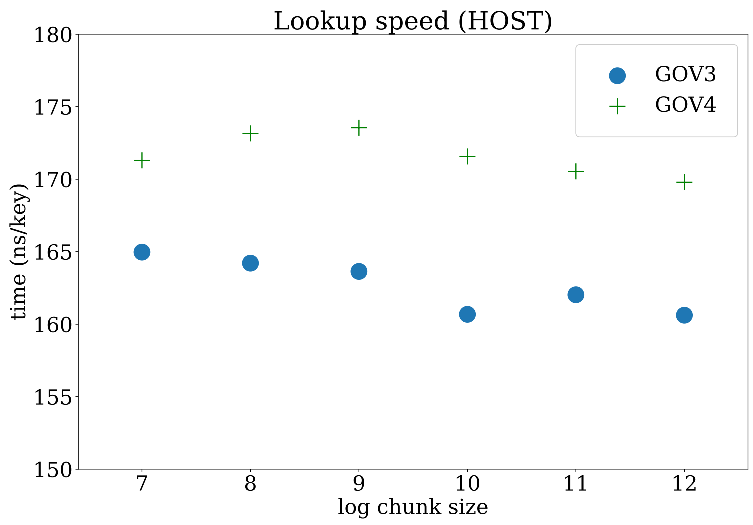


Figure 5.5: Lookup speed for GOV3 and GOV4 on host dataset

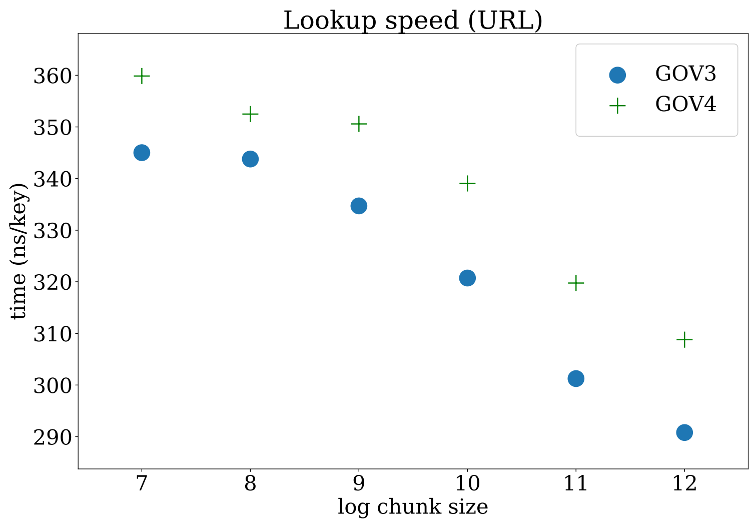


Figure 5.6: Lookup speed for GOV3 and GOV4 on URLs dataset

5.3 Conclusions

In this chapter, we analyzed results for different implementation of static functions. Our experiments show that our method give improvements in both space usage and lookup speed. In the next chapter we will present the results for minimal perfect hash functions. We will analyze BPZ, HDC and a version of BPZ that includes our improvements. In Chapter 7 we will compare GOV3Function and GOV4Function with compressed static function.

Chapter 6

Minimal Perfect Hash Functions

Introduction

As we have seen in Chapter 3, minimal perfect hash functions have been historically more important than static functions. This chapter focuses on the application of our improvements to MPHF, applying an analysis similar to the one we showed in the previous chapter. As before, we focus on the size of the chunk that gives the best performances. We have different implementations available for the construction technique base on linear system solving. We have one for BPZ algorithm and one based on BPZ that includes our improvements. The last one will be called GOVMPHF. We also compared our implementation with the one proposed by the author of HDC.[1]

For minimal perfect hash functions, we choose to use just $r = 3$. As we have seen in Section 3.1.2 there are good reason to use $r = 3$. Indeed for $r = 4$ we need three bits to store each value because two bits are fully covered by the index of the hinge, and one is required for the ranking data structure. Moreover, we need results analogous to the k-XORSAT bounds for the solvability of modular systems, and they are available only for $r = 3$.

6.1 Results

As before, the aim of the first experiment is to find out the best size for the chunks. In Figure 6.1 and Figure 6.2 we show how the number of bits per element vary with the chunk size for $r = 3$. Figure 6.1 shows the results obtained by using the host dataset while Figure 6.2 the ones obtained using the URL dataset. The values for GOVMPHF ranges from 2.55 bit/key to 2.21 bit/key, and from 2.45 bit/key to 2.21 bit/key for the host and URL dataset respectively. These values show that the size of the dataset does not influence the final space as expected by theory.

[1]We used their implementation, this is written in C and is available at [18].

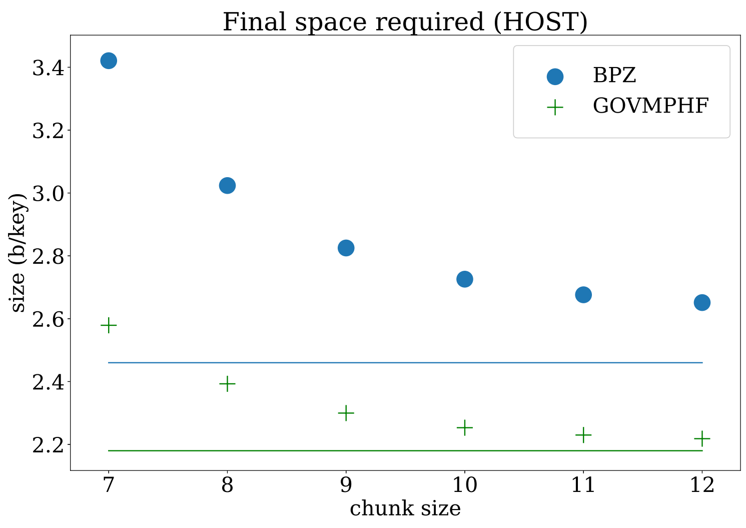


Figure 6.1: Size for BPZ and GOVMPHF on host dataset.

We can see that we can use less space than the classical BPZ. For chunk size equal to 2^{10} we move from 2.72 bits/key required by BPZ construction to 2.25 bit/key required by GOVMPHF.

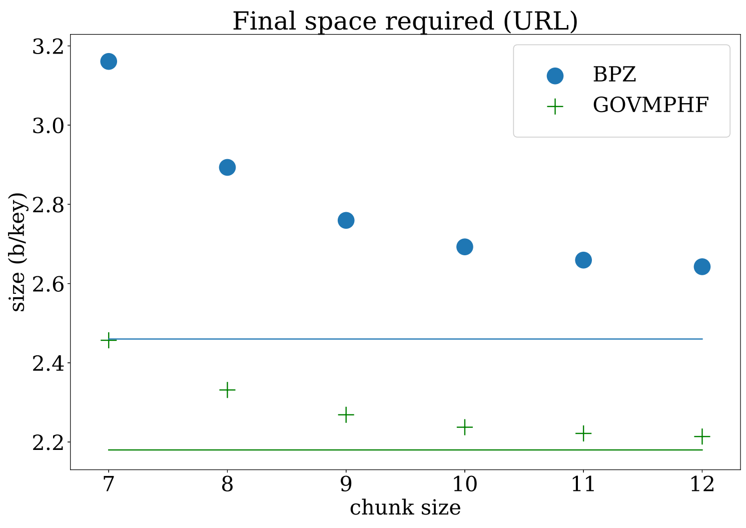


Figure 6.2: Size for BPZ and GOVMPHF on URL dataset.

We can see from Figure 6.1 and Figure 6.2 that we reduced the gap between the theoretical lower bound and the real size of the file. We remark that the the theoretical lower bound is

$2 \cdot c_{2,3}$, as we explained in Subsection 3.1.2 we need two bits for each key to create the ranking data structure.

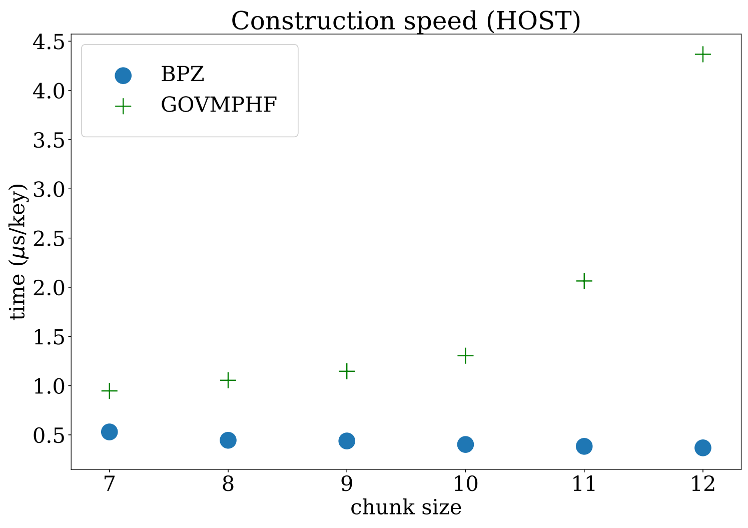


Figure 6.3: Time for BPZ and GOVMPHF on host dataset.

The construction time for GOVMPHF sharply increases as the chunk size passes 2^{10}. As we have seen for GOV3Function, this behavior is typical in constructions based on the solution of linear systems, rather just on peeling (triangulation).

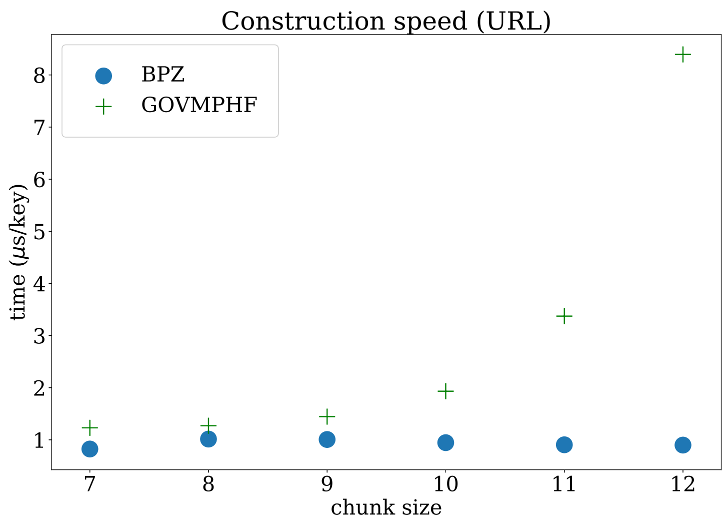


Figure 6.4: Time for BPZ and GOVMPHF on URL dataset.

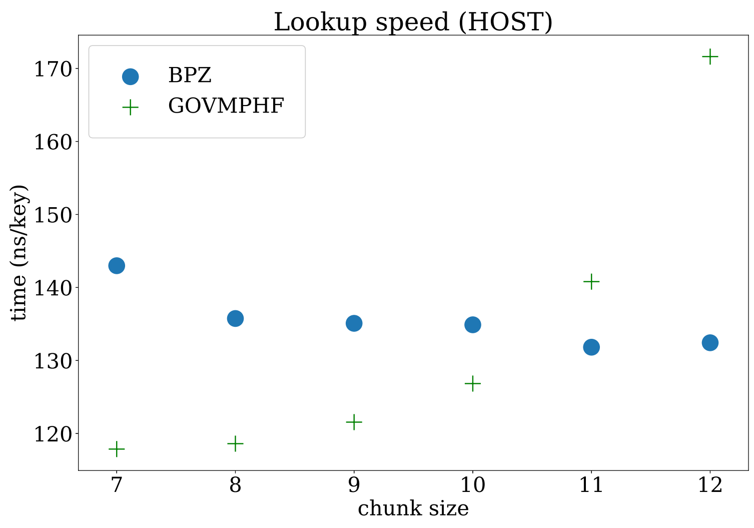


Figure 6.5: Lookup speed for BPZ and GOVMPHF on host datasets

As we can see in Figure 6.5 and Figure 6.6, larger chunks cause slower lookup speed. The substitute of the rank function does more work to compute the inside-chunk rank that we described in Section 4.3.1. Anyway, for small chunks, the improvement that we proposed is two times faster than the rank data structure for BPZ.

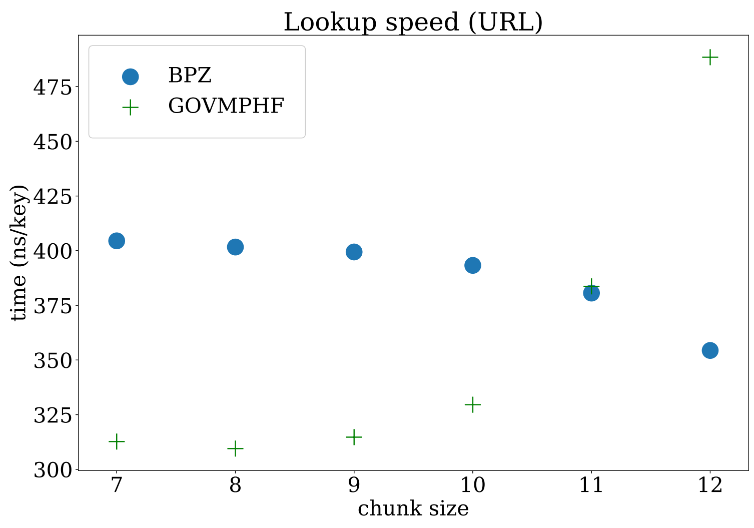


Figure 6.6: Lookup speed for BPZ and GOVMPHF on URLs datasets

In Table 6.2 we show the lookup, construction time and final size of our chosen chunk size, 2^{10}, with respect to the data reported in [2] for the same space usage (i.e., additional 1.10 b/key), and to the C code for the HDC technique made available by the authors (`http://cmph.sourceforge.net/`) when $\lambda = 3$, in which case the number of bits per key is similar to ours. We tried to set the parameter $\lambda = 2$, but the space obtained was 2.44. For completeness, we report all the results for both values of λ in Table 6.1.

	eu-2015-host		eu-2015	
	$\lambda = 3$	$\lambda = 2$	$\lambda = 3$	$\lambda = 2$
Lookup (ns)	371.44	372.33	897.50	1054.90
Construction (μs)	0.75	0.59	2.08	1.58
Size	2.18	2.45	2.18	2.45

Table 6.1: Per-key construction, evaluation time and size for HDC construction with different λ.

We remark that for minimal perfect hash functions is required edge orientation. As for the GOV3Function and GOV4Function, the gap in speed is quite stable with respect to the key size: testing the same structures with very short (less than 8 bytes) random keys provides, of course, faster lookup, but the ratio between the lookup times remain the same.

6.2 Conclusions

In this chapter, we presented the experimental results for minimal perfect hash function. As for the GOV3Function and GOV4Function, the main challenge was to reduce the overhead caused by Gaussian elimination. We show that our construction speed is comparable with the BPZ implementation. We also reduced the gap between the space required and the theoretical lower bound by removing the ranking data structure. As we have seen in Figure 6.5 and Figure 6.6, this choice does not influence lookup speed for small chunks. We have extremely competitive lookup speed and better scalability.

For small sizes, performing the construction entirely in main memory, as HDC does, is an advantage, but as soon as the dataset gets large, our approach scales better. We also remark that our code is a highly abstract Java implementation based on *strategies* that turn objects into

Table 6.2: A comparison of per-key construction and evaluationn time, $r = 3$. HDC is from [8], ADR is from [2].

	eu-2015-host			eu-2015			ADR
	BPZ	GOVMPHF	HDC	BPZ	GOVMPHF	HDC	SF
Lookup (ns)	134.89	126.84	371.44	393.26	329.55	897	?
Construction (μs)	0.40	1.30	0.75	0.96	1.93	2.07	270
Size	2.72	2.25	2.18	2.69	2.24	2.18	1.10

bit vectors at runtime: any object can thus be used as a key. A tight C implementation able to hash only byte arrays, such as that of HDC, would be significantly faster. Indeed, from the data reported in [4] we can estimate that it would be about twice as fast.

Finally, one must consider that HDC, at the price of a much greater construction time, can further decrease its space usage, but just a 9% decrease in space increases construction time by an order of magnitude, which makes the tradeoff unattractive for large datasets.

Chapter 7

Compressed Static Functions

Introduction

In this chapter we will analyze the results of our implementation for compressed static functions. We will use the theoretical construction proposed in [35]. As we have seen in Section 3.2, this construction reaches a space per key proportional to the empirical entropy of the list of output values. We pay this space reduction by multiplying the number of equations that must be solved approximately by the entropy itself.

Thus, it is not obvious that previously known techniques are sufficient to make this construction scalable. Indeed, to the best of our knowledge, no one has ever engineered, implemented and published a practical code for building a compressed static function for a large number of keys, whereas several constructions for the non-compressed case are available.

As in the case of static functions, this construction technique is based on linear system solving. Thus, we tested our implementation for $r = 3$ and $r = 4$ as in Chapter 5. In both minimal perfect hash function and static functions, we focused on the behavior of the implementation varying the chunk size. Instead, for the compressed static function, we focus on the list of values; we wanted to find out which distribution has the best performance using CSF construction. We compared our construction with the implementation provided by the classes GOV3Function and GOV4Function.

7.1 Datasets

Before analyzing the results, we will describe the datasets and the lists of values. As in the previous chapters, we consider a list of 11 264 052 hostnames, and a list of 1 070 557 254 URLs. However, in this case, for each key set we generate three synthetic lists of values using different distributions:

- a geometric distribution with probability $p = 1/2$;

- a Zipfian (i.e., finite power-law) distribution with $N = 10^6$ values and exponent $s = 2$;
- a discrete uniform distribution with 64 values.

Moreover, as real-world case we consider the mapping from each URL (host) to its indegree in the web (host, respectively) graph.

The geometric and uniform distributions are the most skewed and the less skewed distribution with exactly matching optimal codes, and the Zipfian distribution sits somewhere in the middle. Note that in principle the geometric distribution has un unlimited range, but in practice for n keys the largest generated value is $O(\log n)$ with high probability.

Table 7.1 reports the empirical value range and empirical entropy of the eight resulting combinations. Figure 7.1 represents the Pareto chart for each list of values for the host dataset. To make plots more readable, we limited the cumulative sum at 0.97 of total value for the distribution with a long tail.

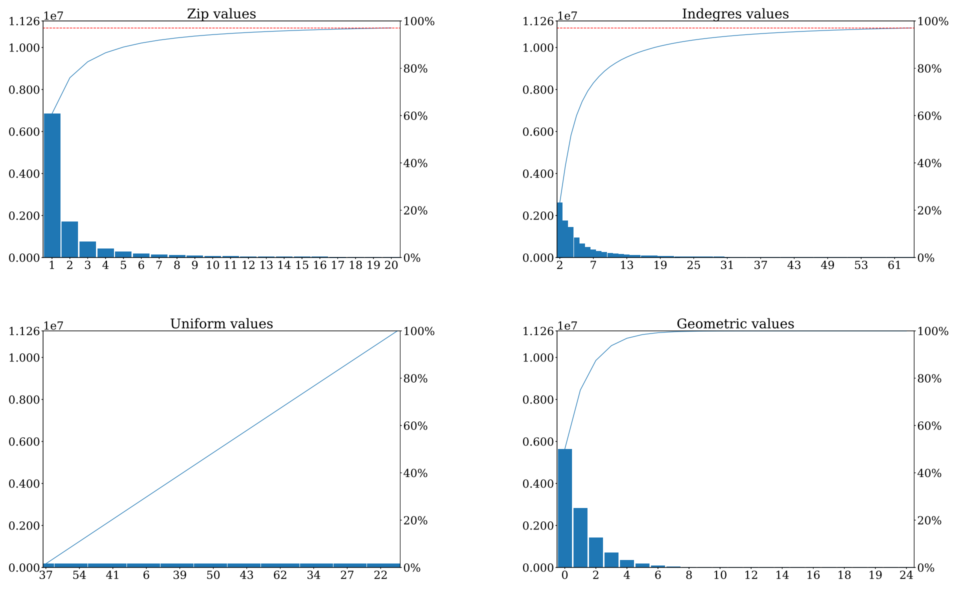


Figure 7.1: Pareto chart for each value list for host dataset

Dataset	Geometric	Zipfian	Uniform	Indegree
Hosts	$[0 \dots 23]$, 2.0	$[1 \dots 774599]$, 2.36	$[0 \dots 63]$, 6.0	$[1 \dots 174433]$, 4.22
URLs	$[0 \dots 31]$, 2.0	$[1 \dots 997570]$, 2.36	$[0 \dots 63]$, 6.0	$[1 \dots 20252239]$, 5.10

Table 7.1: Actual ranges and empirical entropy of our synthetic and real-world datasets.

7.2 Results

In Table 7.2 and 7.3 we report the results of our eight combinations of datasets and output values, coupled with similar results for the non-compressed data structures distributed with Sux4J.

- If the distribution is skewed and the entropy is small, we obtain a significant compression, very close to $c_{2,r} \cdot H_0$ bits per key, as expected. The compression is more impressive in the case of a distribution with a long tail, as in that case for large datasets a few very large values occur, which forces the standard implementation to choose a large number of bits per value. In the case of the geometric distribution, this might happen, too, but with very small probability. Construction time increases, in some cases very significantly (e.g., URL indegree, $r = 4$).

- In the uniform case, construction time starts to grow significantly, and we have no space gain. Nonetheless, since we can use a value of $c_{2,3}$ very close to the best possible, we do not increase the space usage. Lookup time is slightly slower due to decoding, but by using a flat binary code, it would be equivalent to the standard case.

- Lookup times for Zipfian and Indegree dataset are (sometimes significantly) shorter. It is due to the reduced size of the bit array. Since the bit array is smaller we have less stress on the cache and on the Translation Lookahead Buffer.

- concerning the case $r = 3$, the case $r = 4$ exhibits slightly increased access time, a few percent reduction in space (as expected) and much longer construction time, due to the growing density of the linear systems.

Host dataset	$r = 3$							
	Geometric		Zipfian		Uniform		Indegree	
	Std	Comp	Std	Comp	Std	Comp	Std	Comp
Size (b/key)	5.56	2.27	22.10	2.75	6.66	6.67	19.89	4.78
Constr. (μs/key)	0.82	1.57	0.82	2.06	1.06	11.25	0.80	5.35
Lookup (ns/key)	103.97	97.62	152.74	100.08	114.11	116.70	152.38	119.96
	$r = 4$							
	Geometric		Zipfian		Uniform		Indegree	
	Std	Comp	Std	Comp	Std	Comp	Std	Comp
Size (b/key)	5.21	2.12	20.69	2.59	6.23	6.24	18.62	4.48
Constr. (μs/key)	1.27	3.38	1.42	4.88	1.29	44.90	1.33	18.30
Lookup (ns/key)	109.02	108.00	164.72	108.04	121.02	123.50	163.45	124.01

Table 7.2: Comparison between the non-compressed implementations from Sux4J (Std) and our compressed implementation (Comp) for the host dataset.

URL dataset	$r = 3$							
	Geometric		Zipfian		Uniform		Indegree	
	Std	Comp	Std	Comp	Std	Comp	Std	Comp
Size (b/key)	5.55	2.25	22.09	2.72	6.65	6.65	27.60	5.72
Constr. (μs/key)	1.46	2.58	1.47	3.29	1.46	19.81	1.49	13.83
Lookup (ns/key)	270.42	271.92	315.56	277.28	276.47	275.55	318.47	302.72
	$r = 4$							
	Geometric		Zipfian		Uniform		Indegree	
	Std	Comp	Std	Comp	Std	Comp	Std	Comp
Size (b/key)	5.191	2.11	20.67	2.55	6.22	6.23	25.83	5.36
Constr. (μs/key)	2.17	5.96	2.17	8.75	2.17	86.20	2.23	55.70
Lookup (ns/key)	277.68	280.05	325.98	284.48	285.96	291.06	339.42	309.49

Table 7.3: Comparison between the non-compressed implementations from Sux4J (Std) and our compressed implementation (Comp) for the URL dataset.

7.3 Scalability

We tested scalability using sequence of subsets of the URL datasets with sizes increasing exponentially from 2^{22} to 2^{27}, with geometrically distributed values. Figure 7.2 shows that there is essentially no difference in per-key construction time and space usage. Indeed, due to HEM once

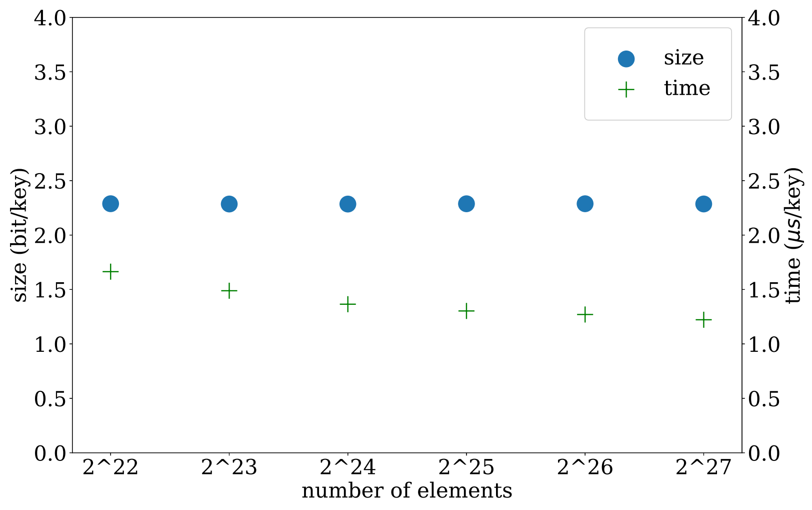


Figure 7.2: Size and construction time for increasing size of the key set (geometrically distributed output, $r = 3$).

we fix the chunk log size the only non-linear part of the construction is the sorting phase, which accounts for just a few percent of the overall construction time. Maybe surprisingly, one can witness a small *decrease* in per-key construction time due to some constant costs (JVM startup, etc.) being amortized on a larger number of keys.

7.4 Switching to Triangulation.

Since we try to triangulate the system before solving it by lazy Gaussian elimination, for $r = 3$ when we move from $c_{2,3}$ to γ_3 we can triangulate the system in linear time with high probability (as in the MWHC case). In practice, with a 12% increase in space, we can *guarantee* that the increase in construction time will be at most multiplied by the entropy. In fact, as we show in Table 7.4, the increase is usually much less, as the triangulation process is responsible for a small fraction of the overall construction time. Maybe surprisingly, because of the reduced memory footprint the construction time for the geometric case is *faster* than the standard construction. As we state in Chapter 2, note that γ_3 s *minimum* among the γ_r, so there is no sense in trying this approach for $r \neq 3$.

URL dataset	$r = 3$			
	Geometric	Zipfian	Uniform	Indegree
Size (b/key)	2.51	3.03	7.43	6.39
Constr. (μs/key)	1.04	1.60	1.94	1.95
Lookup (ns/key)	272.68	278.52	285.95	305.44

Table 7.4: Data for $\delta = 1.23$, using triangulation to solve linear systems.

7.5 Table-Limited Huffman Code

As stated before it is possible to use different a static code to encode values. We tested the performance of the *Unary* code. This code is optimal if the frequencies of the output values follow a geometric distribution with $p = 0.5$. In our case we encode each value $f(x_i)$ with the bit sequence $0^{f(x_i)}1$, and the decoding can be performed counting the leading zeros. This is a common operation in browadword programming and has been introduced in *SSE4.2* Intel instruction set in 2007. As expected the final size and the construction time are similar to the one we reported in Table 7.2 and Table 7.3. The lookup time for the host dataset is ≈ 90 ns/key and ≈ 204ns/key for the URLs dataset. The values do not differs too much from the reported in Table 7.2 and Table 7.3 for the same datasets. The main drawback of the use of static Unary code is its limited applications; indeed, it is very uncommon to store functions such that the output values follows exactly a geometric distribution.

The last experiment that we performed shows the behavior of the function when we change the length of the decoding table of the table-limited Huffman code (see Section 4.3.3). For this

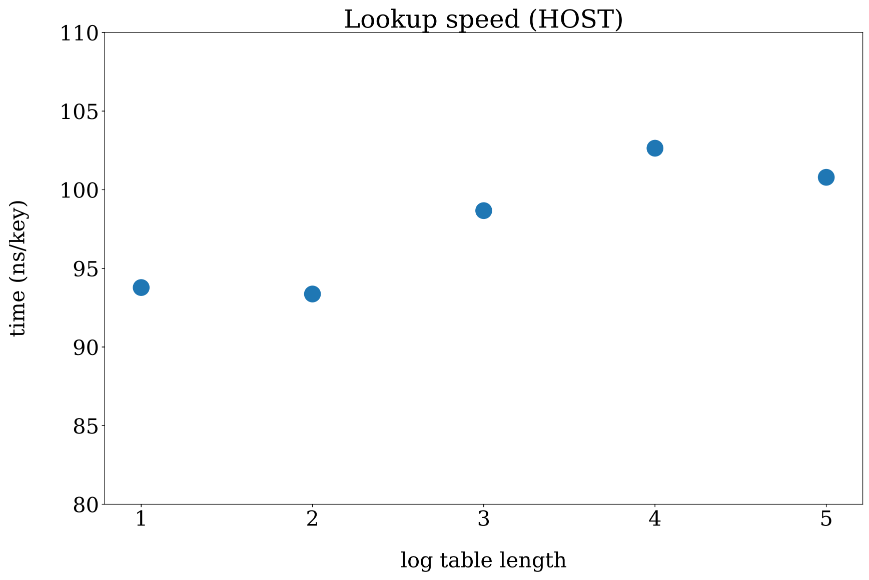


Figure 7.3: Lookup speed vs. length of the decoding table (geometrically distributed output).

experiment we will use the host dataset and the list of values generated using the geometric distribution with $p = 0.5$. This distribution is the most skewed and exploits the changes in the decoding table. Then we changed the parameter $\ell = 2^i, i \in \{1, 2, 3, 4, 5\}$, and we measured

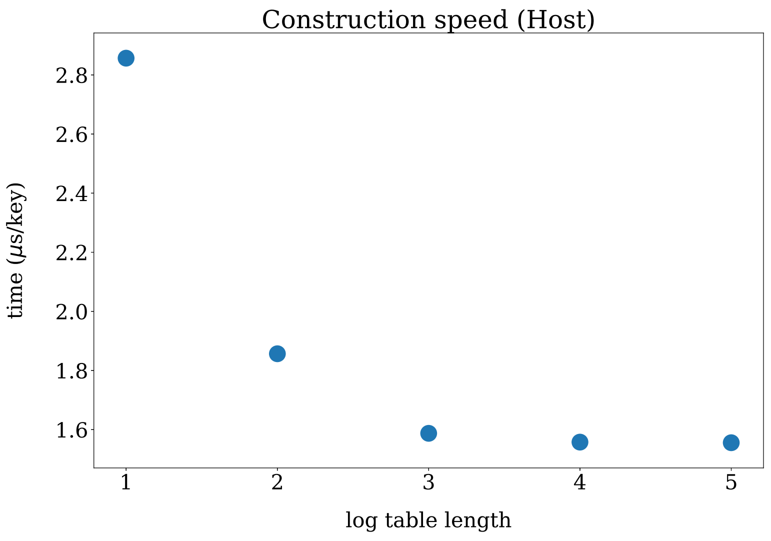


Figure 7.4: Construction time vs. length of the decoding table.

construction speed, space usage and lookup speed. We expect that for values of ℓ beyond 8 the impact of the table-limited Huffman code should be marginal.

Note that for $\ell = 32$ there is no difference from table-limited Huffman code and canonical Huffman code, beacuse in the geometric-distributed list of value the longest codeword has length equal to 23.

In theory, we need to use a table-limited Huffman code to perform decoding in constant time. In practice, however, the decoding time is dominated by the time required to compute the r hash functions h_i. As we can see in Figure 7.3, the speed values slowly increases for different table lengths, but never more than 10%.

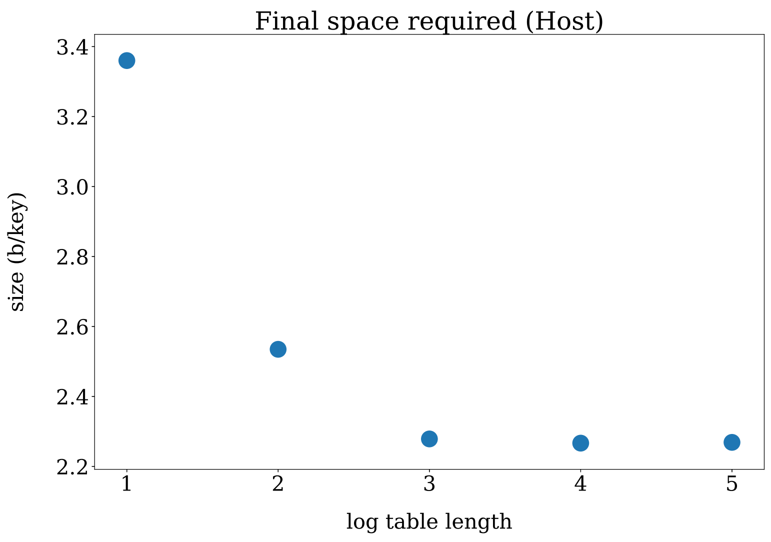


Figure 7.5: Size vs. length of the decoding table.

Nonetheless, the size of the table deeply affects the construction speed and space usage.

If we use a non-optimal code for the frequency of values, the number of equations increases and this presents two drawbacks: the system is bigger and the number of total variables grows. It is obvious that if we use a non-optimal code the space usage increases. We report these results in Figure 7.5. The space usage ranges from 3.36,b/key to 2.27 b/key. As shown in Figure 7.4, when we have more equations the time required to solve the system increases. We can nearly halve the construction time, from 2.86 μs/key to 1.56 μs/key;

Conclusions

In this thesis, we investigated construction techniques for static functions, that is, maps from a fixed subset of the universe to a set of integers. This family of functions can be divided in several subclasses: perfect hash function, minimal perfect hash function, order preserving minimal perfect hash function and compressed static functions. Each of these classes has different purpose: for instance, to retrieve the position of a key in a give list of keys we use order preserving minimal perfect hash functions. In general, minimal perfect hash functions can be used as building block of space efficient data structures. As we have seen in Chapter 3, several construction techniques have been proposed in the literature. We selected the ones that have the widest range of applications, which are based on the solution of random linear system. The first formulation of this construction technique takes advantage of the graph theory. As we have seen in Section 2.3, a random linear system can be represented using hypergrahs. In particular, in a random uniform hypergraphs if the ratio between nodes and edges is greater than the threshold γ_r the system can be triangulated in linear time. Our ambition was to beat the γ_r multiplicative factor in front of space bound without increasing significantly the construction time, as below this factor the system cannot be triangulated in linear time, and we need to solve it. To do so, we used the $c_{k,r}$ threshold described in [19].

In Chapter 4 we have presented how we can engineer data structures for construction based on linear systems. The most significant improvement is given by Lazy Gaussian Elimination 4.2. This heuristic derives from a sparse large system a small, denser one, which can be solved using standard Gaussian elimination. To speed up this process we represented equations using bit arrays and used *broadword programming* to manipulate them.

In Section 4.1.2, we described the algorithms to perform row operation for equations over $\mathbf{F}_3$. In the same chapter we introduced minor improvements, such as HEM to reduce the amount of main memory required during construction. In minimal perfect hash function representation we can remove the ranking as we explained in 4.3.1. For compressed static functions we introduced a new table-limited canonical Huffman code 4.3.3 and we trade optimality for constant-time decoding. We also discussed the drawback of the structured hash functions proposed in [35]. The asymptotic behavior of the new threshold causes the failure of many small linear system affecting construction speed.

In the experimental part of the thesis we compared the results for all the construction techniques based on linear system solving. Firstly, we described how we perform the experiments and the dataset that we used. Then in Chapter 5 we presented the results for GOV3Function and GOV4Function, the implementations for static function. We analyzed the behavior of the construction technique for different size of the linear system. We focused on final space required and construction speed. We choose the size that provides the best trade off between these two measures. Indeed, the number of equations does not influence the lookup speed. We compared the results with the hypergraph-based construction. Remarkably, our study demonstrate that our method can improve signficantly lookup speed with respect to previous construction.

In Chapter 6, we performed experiments on our minimal perfect hash function implementation. After the elimination of the ranking data structure we analyzed how the size of the linear system influence the lookup speed and the final size. The lookup speed increases as the size of the linear system, but for sizes less than 2^{10} the lookup speed grows slowly. Under that value, lookup for GOVMPHF is faster than BPZ construction. We also compared our new construction with one of the best construction technique available, HDC. This construction has similar performances in terms of final size and construction speed. Anyway, we can state that GOVMHPF is about 2.5 times faster in lookup speed than HDC construction as reported in Table 6.2. We consider this parameter crucial for the design of these data structure.

The last chapter of the experimental part is about compressed static hash functions 7. As far as we know, this is the first engineered implementation of the theoretical construction proposed in [35] for compressed static functions. For this construction technique the number of equations in the linear system is not equal to the size of the keyset. Rather, it depends on the 0th order entropy of the list of the outputs of the function. So we experimented different list of values as described in 7.1. We have seen that if the list of values has a small entropy this construction technique outperform GOV3/GOV4Function. If the distribution of the values has a long tail, but high entropy, we can reduce the space usage and improve lookup speed with respect to GOV3/GOV4Function. However, construction speed is slower than GOV3/GOV4Function. Note that to circumvent the high-entropy case one can measure at construction time the entropy of the output values and just switch to the construction based on linear-time system triangulation, as showed in Table 7.4.

We remark that the code for GOV3/GOV4/MinimalPerfectHashFunctions is already included in the Sux4J library. This is an open source project and we hope that it can be used as term of comparison with future construction techniques. We also plan to release the code for compressed static function: it will use the inherent parallelism of HEM to decrease the construction time by an order of magnitude if sufficient cores are available. For experiment comparing with previous literature we believe that a single-threaded approach gives less biased results.

It is also easy to use distributed computational frameworks to solve independently each chunk. For compressed static functions we suggest the $k = 3$ for a practical data structure, as construction time for $k = 4$ becomes too high, with minimal space gains.

If the distribution of values is known in advance, Huffman codes can in principle be replaced by a standard instantaneous code such as unary or Elias's γ. This choice reduces the space usage since the decoding structure is no longer required.

We remark that one of the advantages of the theoretical approach we chosen for our algorithmic engineering efforts [35] is that the number of memory accesses to compute a value is identical to that of the non-compressed case we are comparing with: indeed, in our tests we found a minor lookup slowdown in the uniform case, and a speedup in some skewed cases, in spite of the time that is necessary for decoding, as the smaller memory footprint reduces the out-of-cache accesses and the stress on the Translation Lookaside Buffer.

Thanks

Here I wish to thank all the people without whom this work would have been impossible: first, Sebastiano Vigna, my advisor, for all the efforts he took to guide my research, since the very beginning of my experience as a PhD student; his ideas and experencie were contagiuos and motivational for me, even during tough times in the Ph.D. pursuit. Then, let me thank Paolo Boldi, my co-advisor, his patience and empathy helped me all along the doctorate. I am proud to have been a member of their Laboratory of Web Algorithmics in Milan.

I want to thank my reviewers Djamal Belazzougui, Rasmus Pagh and Rossano Venturini, for all the time and energy they spent to improve my work. Their suggestions were precise and helpful.

Let me thank all the other contributors to this work: Giuseppe Ottaviano from Facebook, who provided ideas and technical improvements to our code; Massimo Santini, again from the Laboratory of Web Algorithmics, for all of his scientific, technological, academic and human support over the years. Corrado Monti for being a perfect senior Ph.D. student.

I wish to thank Luca Prigioniero, Pierlauro Sciarelli, Claudio Ceruti and all the other professors, fellow colleagues and friends in via Comelico, for their friendship and their helpful and thought-provoking everyday discussions.

Finally, let me thank my family; my parents, my sister, then my girlfriend Francesca and my friend from Udine and Milan. Your role is so profound it cannot be put into words.

Bibliography

[1] V. L. Arlazarov, E. A. Dinic, M. A. Kronrod, and I. A. Faradžev, *On economical construction of the transitive closure of a directed graph*, Soviet Mathematics—Doklady **11** (1970), no. 5, 1209–1210.

[2] Martin Aumüller, Martin Dietzfelbinger, and Michael Rink, *Experimental variations of a theoretically good retrieval data structure*, Algorithms - ESA 2009, 17th annual european symposium, Copenhagen, Denmark, September 7-9, 2009. proceedings, 2009, pp. 742–751.

[3] Peter Ayre, Amin Coja-Oghlan, Pu Gao, and Noëla Müller, *The satisfiability threshold for random linear equations*, 2017.

[4] Djamal Belazzougui, Paolo Boldi, Giuseppe Ottaviano, Rossano Venturini, and Sebastiano Vigna, *Cache-oblivious peeling of random hypergraphs*, 2014 data compression conference (dcc 2014), 2014, pp. 352–361.

[5] Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna, *Monotone minimal perfect hashing: Searching a sorted table with $O(1)$ accesses*, Proceedings of the 20th annual ACM-SIAM symposium on discrete mathematics (SODA), 2009, pp. 785–794.

[6] ———, *Fast prefix search in little space, with applications*, Algorithms - ESA 2010, 18th annual european symposium, Liverpool, UK, september 6-8, 2010. proceedings, part I, 2010, pp. 427–438.

[7] ———, *Theory and practice of monotone minimal perfect hashing*, ACM Journal of Experimental Algorithmics **16** (2011), no. 3, 3.2:1–3.2:26.

[8] Djamal Belazzougui, Fabiano C. Botelho, and Martin Dietzfelbinger, *Hash, displace, and compress*, Algorithms - ESA 2009, 17th annual european symposium, Copenhagen, Denmark, September 7-9, 2009. proceedings, 2009, pp. 682–693.

[9] Djamal Belazzougui and Gonzalo Navarro, *Alphabet-independent compressed text indexing*, Esa (1), 2011, pp. 748–759.

[10] Djamal Belazzougui and Rossano Venturini, *Compressed static functions with applications*, Soda, 2013, pp. 229–240.

[11] Paolo Boldi, Andrea Marino, Massimo Santini, and Sebastiano Vigna, *BUbiNG: Massive crawling for the masses*, Proceedings of the companion publication of the 23rd international conference on world wide web companion, 2014, pp. 227–228.

[12] Fabiano C. Botelho, Rasmus Pagh, and Nivio Ziviani, *Simple and space-efficient minimal perfect hash functions*, Proc. wads 2007, algorithms and data structures, 10th international workshop, 2007, pp. 139–150.

[13] ———, *Practical perfect hashing in nearly optimal space*, Inf. Syst. **38** (2013), no. 1, 108–131.

[14] Neil J Calkin, *Dependent sets of constant weight binary vectors* (1995).

[15] Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, and Ayellet Tal, *The Bloomier filter: an efficient data structure for static support lookup tables*, Proceedings of the fifteenth annual ACM-SIAM symposium on discrete algorithms, SODA 2004, 2004, pp. 30–39.

[16] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest, *Introduction to algorithms*, The MIT Press and McGraw-Hill Book Company, 1989.

[17] Zbigniew J. Czech, George Havas, and Bohdan S. Majewski, *Perfect hashing*, Theoretical Computer Science **182** (1997), no. 1-2, 1–143.

[18] Davi de Castro Reis, Djamel Belazzougui, Fabiano Cupertino Botelho, and Nivio Ziviani, *Cmph - c minimal perfect hashing library*. Accessed: 2017-09-18.

[19] Martin Dietzfelbinger, Andreas Goerdt, Michael Mitzenmacher, Andrea Montanari, Rasmus Pagh, and Michael Rink, *Tight thresholds for cuckoo hashing via XORSAT*, Automata, languages and programming, 2010, pp. 213–225 (English).

[20] Martin Dietzfelbinger and Rasmus Pagh, *Succinct data structures for retrieval and approximate membership (extended abstract)*, Automata, languages and programming, 35th international colloquium, ICALP 2008, proceedings, part I: Track A: Algorithms, automata, complexity, and games, 2008, pp. 385–396.

[21] Olivier Dubois and Jacques Mandler, *The 3-xorsat threshold*, Comptes Rendus Mathematique **335** (2002), no. 11, 963–966.

[22] Peter Elias, *On binary representations of monotone sequences*, Proc. sixth princeton conference on information sciences and systems, 1972, pp. 54–57.

[23] ———, *Universal codeword sets and representations of the integers*, IEEE Transactions on Information Theory **21** (1975), 194–203.

[24] Robert M. Fano, *On the number of bits required to implement an associative memory*, 1971. Memorandum 61, Computer Structures Group, Project MAC, MIT, Cambridge, Mass., n.d.

[25] Edward Fredkin, *Trie memory*, Commun. ACM **3** (September 1960), no. 9, 490–499.

[26] Michael L. Fredman and János Komlós, *On the size of separating systems and families of perfect hash functions*, SIAM J. Algebraic Discrete Methods **5** (1984), no. 1, 61–68.

[27] T. Gagie, G. Navarro, Y. Nekrich, and A. Ordóñez, *Efficient and compact representations of prefix codes*, IEEE Transactions on Information Theory **61** (2015Sept), no. 9, 4999–5011.

[28] Marco Genuzio, Giuseppe Ottaviano, and Sebastiano Vigna, *Fast scalable construction of (minimal perfect hash) functions*, Experimental algorithms - 15th international symposium, SEA 2016, St. Petersburg,Russia, June 5-8, 2016, proceedings, 2016, pp. 339–352.

[29] Marco Genuzio and Sebastiano Vigna, *Compressed static functions: Experimental results and analysis*, Draft, 2017.

[30] Andreas Goerdt and Lutz Falke, *Satisfiability thresholds beyond k -XORSAT*, Computer science — theory and applications: 7th international computer science symposium in russia, csr 2012. proceedings, 2012, pp. 148–159.

[31] Simon Gog and Matthias Petri, *Optimized succinct data structures for massive data*, Software: Practice and Experience (2014). To appear.

[32] Torben Hagerup and Torsten Tholey, *Efficient minimal perfect hashing in nearly minimal space*, Stacs 2001, 18th annual symposium on theoretical aspects of computer science, Dresden, Germany, February 15-17, 2001, proceedings, 2001, pp. 317–326.

[33] George Havas, Bohdan S. Majewski, Nicholas C. Wormald, and Zbigniew J. Czech, *Graphs, hypergraphs and hashing*, Proceedings of the 19th international workshop on graph-theoretic concepts in computer science, 1994, pp. 153–165.

[34] Daniel S. Hirschberg and Debra A. Lelewer, *Efficient decoding of prefix codes*, Commun. ACM **33** (April 1990), no. 4, 449–459.

[35] Jóhannes B. Hreinsson, Morten Krøyer, and Rasmus Pagh, *Storing a compressed function with constant time access*, Algorithms - ESA 2009, 17th annual european symposium, Copenhagen, Denmark, September 7-9, 2009. proceedings, 2009, pp. 730–741.

[36] Guy Jacobson, *Space-efficient static trees and graphs*, 30th annual symposium on foundations of computer science (FOCS '89), 1989, pp. 549–554.

[37] Donald E. Knuth, *The Art of Computer Programming. Pre-Fascicle 1A. Draft of Section 7.1.3: Bitwise Tricks and Techniques*, Addison–Wesley, 2007.

[38] S. Rao Kosaraju and Giovanni Manzini, *Compression of low entropy strings with lempel-ziv algorithms*, SIAM J. Comput. **29** (1999), no. 3, 893–911.

[39] Brian A. LaMacchia and Andrew M. Odlyzko, *Solving large sparse linear systems over finite fields*, Advances in cryptology: Crypt0'90, 1991, pp. 109–133.

[40] Lawrence L. Larmore and Daniel S. Hirschberg, *A fast algorithm for optimal length-limited huffman codes*, J. ACM **37** (July 1990), no. 3, 464–473.

[41] Bohdan S. Majewski, Nicholas C. Wormald, George Havas, and Zbigniew J. Czech, *A family of perfect hashing methods*, Comput. J. **39** (1996), no. 6, 547–554.

[42] Kurt Mehlhorn, *Data structures and algorithms 1: Sorting and searching*, EATCS Monographs on Theoretical Computer Science, vol. 1, Springer, 1984.

[43] Milidiú and Laber, *Bounding the inefficiency of length-restricted prefix codes*, Algorithmica **31** (2001Dec), no. 4, 513–529.

[44] Michael Molloy, *Cores in random hypergraphs and Boolean formulas*, Random Structures and Algorithms **27** (2005), no. 1, 124–135.

[45] Andrew M. Odlyzko, *Discrete logarithms in finite fields and their cryptographic significance*, Advances in cryptology, 1985, pp. 224–314.

[46] Rasmus Pagh, *Hash and displace: Efficient evaluation of minimal perfect hash functions*, Algorithms and data structures, 6th international workshop, WADS '99, Vancouver, British Columbia, Canada, August 11-14, 1999, proceedings, 1999, pp. 49–54.

[47] Mihai Patrascu, *Succincter*, 49th annual IEEE symposium on foundations of computer science, 2008, pp. 305–313.

[48] Boris Pittel, Joel Spencer, and Nicholas C. Wormald, *Sudden emergence of a giant*k*-core in a random graph*, J. Comb. Theory, Ser. B **67** (1996), no. 1, 111–151.

[49] Michael Rink, *Thresholds for matchings in random bipartite graphs with applications to hashing-based data structures*, Ph.D. Thesis, 2015.

[50] Jeanette P. Schmidt and Alan Siegel, *The spatial complexity of oblivious k-probe hash functions*, SIAM J. Comput. **19** (September 1990), no. 5, 775–786.

[51] Eugene S. Schwartz and Bruce Kallick, *Generating a canonical prefix encoding*, Commun. ACM **7** (March 1964), no. 3, 166–169.

[52] Volker Strassen, *Gaussian elimination is not optimal*, Numer. Math. **13** (August 1969), no. 4, 354–356.

[53] Robert Endre Tarjan and Andrew Chi-Chih Yao, *Storing a sparse table*, Commun. ACM **22** (1979), no. 11, 606–611.

[54] Sebastiano Vigna, *Broadword implementation of rank/select queries*, Experimental algorithms. 7th international workshop, wea 2008, 2008, pp. 154–168.

[55] D H Wiedemann, *Solving sparse linear equations over finite fields*, IEEE Trans. Inf. Theor. **32** (January 1986), no. 1, 54–62.