View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Archives of the Faculty of Veterinary Medicine UFRGS

Deterministic and efficient minimal perfect hashing schemes

Leandro M. Zatesko [1]
Jair Donadelli [2]

Abstract: This paper presents deterministic versions to the hashing schemes of Botelho, Kohayakawa and Ziviani (2005) and Botelho, Pagh and Ziviani (2007), also proves a statement left as open problem in the former work, related to the correctness proof and to the complexity analysis of their scheme. Our deterministic variants have been implemented and executed over datasets with up to 25,000,000 keys and have brought equivalent performance results between the deterministic and the original randomized algorithms.

Resumo: Neste trabalho apresentamos versões determinísticas para os esquemas de *hashing* de Botelho, Kohayakawa e Ziviani (2005) e de Botelho, Pagh e Ziviani (2007). Também respondemos a um problema deixado em aberto no primeiro dos trabalhos, relacionado à prova da corretude e à análise de complexidade do esquema por eles proposto. As versões determinísticas desenvolvidas foram implementadas e testadas sobre conjuntos de dados com até 25.000.000 de chaves, e os resultados verificados se mostraram equivalentes aos dos algoritmos aleatorizados originais.

1 Introduction

A *minimal perfect hashing scheme*, as defined in [1, 2, 3], is an algorithm that, given a set S with n keys from an universe U, constructs a hash function $h\colon U \to \{0,\dots,n-1\}$ which maps without collision S to $\{0,\dots,n-1\}$. We are interested only in hashing schemes whose outputs are hash functions with $O(1)$ lookup time. For our purposes, every key x is assumed to be a chain of at most L symbols taken from a finite alphabet Σ, for a fixed constant L. As an example, keys can be URLs of length at most L which we are trying to map to memory addresses, and the alphabet would then contain decimal digits, Latin letters and some special characters like `/` and `?`.

Hashing is a widely studied topic in Computer Science. Mapping n objects bijectively to hash table addresses $\{0,\dots,n-1\}$ is a very often problem, and minimal perfect hash functions, in particular, are useful in situations related to "efficient storage and fast retrieval of items from static sets, such as words in natural languages, reserved words in programming

[1]Departamento de Informática, UFPR, Brazil, PO Box 19081
leandro@inf.ufpr.br
[2]Centro de Matemática, Computação e Cognição, UFABC, Brazil, CEP 09210-170
jair.donadelli@ufabc.edu.br

languages or interactive systems, universal resource locations (URLs) in Web search engines, or item sets in data mining techniques" [1].

Derandomization is an important subject of Computational Complexity and a way to understand whether randomness in algorithms is necessary. Formally, a central problem about complexity of randomized algorithms is "$\mathcal{P} = \mathcal{BPP}$?". For example, the problem of polynomial identity testing is in $\mathcal{BPP}$, which means that it is solvable by a polynomial-time *Monte Carlo* algorithm [4], an algorithm whose answer can be wrong with bounded probability. Derandomizing polynomial identity tests has deep consequences in Computational Complexity [5]. On the other hand, the celebrated polynomial-time deterministic algorithm for primality testing [6] is a successful derandomization of a *Monte Carlo* polynomial-time algorithm [7]. Differently, the hashing schemes we study are known as *Las Vegas* algorithms [4], which means that their answer is always right, but the time complexity is a random variable. Problems solvable by *Las Vegas* algorithms with expected polynomial-time complexity form the class $\mathcal{ZPP} \subseteq \mathcal{BPP}$.

We shall present derandomized versions to the hashing schemes of Botelho, Kohayakawa and Ziviani (2005) and of Botelho, Pagh and Ziviani (2007), from now on referred as BMZ and BDZ, respectively. These schemes are *Las Vegas* algorithms that, given a set with n keys, construct in expected time $O(n)$ a hash function which in $O(1)$-evaluation time maps without collision the keys to the set $\{0,\ldots,n-1\}$. The problem of constructing a minimal perfect $O(1)$-evaluation time hash function given a set with n keys is, of course, in $\mathcal{P}$ [8], therefore our work just shows that these *very practical* algorithms didn't need to be randomized to achieve a good average performance. Actually we derandomize the schemes in a very simple manner, and the resulting algorithms are schemes of $O(n)$ average-case time complexity. Additionally, we also give a proof for a question left open in [1] (Equation 2 below), closing the complexity analysis and the correctness proof of the BMZ scheme.

In what follows, unless stated otherwise, we use the term *graph* to refer to a simple graph, that is an unweighted, undirected graph containing no loops or multiple edges. The term *critical subgraph* of a graph refers to the maximal subgraph with minimum degree $\delta \geqslant 2$. The term *parent* of a vertex in a graph search is used according to the traditional meaning, as could be found in [9], in any graph search the term *tree edge* refers to an edge in which one endpoint is the parent of the other, and the term *back edge* refers to an edge which is not a tree edge. Also, we write $a(n) \approx b$ if $a(n) \to b$ as $n \to \infty$.

This paper is organized as follows. In Section 2 we shall give a brief review on related works, in Section 3 we shall present deterministic versions of the schemes BMZ and BDZ, in Section 4 we shall prove a graph theoretical result related to the complexity analysis and to the correctness proof of BMZ scheme, in Section 5 we shall give performance comparisons between the randomized and derandomized schemes and finally in Section 6 we shall close with some considerations about hashing and our results.

2 Related works

It is known that finding a perfect hash function for sets with n keys cannot be done in $o(n)$ time [3], although many $O(n)$-time perfect hashing schemes are known from literature [2, 10, 11, 12]. For example, the randomized hashing scheme presented in [3], which maps the n keys to the edges of an acyclic graph on $2.09n$ vertices and then uses a depth-first search to label the edges with the values $1,\dots,n$, constructs a minimal perfect hash function in $O(n)$ expected time. The constructed hash function requires $O(n\log n)$ bits to be stored, and this amount of space is proportional to the size of the graph. This important hashing scheme inspired the BMZ scheme [1], which, allowing the graph to be cyclic, reduces the number of vertices to $1.15n$.

In 1984 Fredman and Komlós [13] proved that $n\lg\mathrm{e} + \lg\lg u + O(\log n)$ is a lower bound for the space of $O(1)$-evaluation time hash functions built by a minimal perfect scheme on an universe with u objects. Remark that this means about $\lg\mathrm{e} \cong 1.443$ bit per key. In addition, Melhorn [14] presented in 1984 a theoretical scheme with which proved the lower bound to be tight. His scheme however was an exponential-time algorithm. Both schemes of [3] and [1], although perfect, minimal, practical and efficient in time, construct hash functions represented by an undesirable amount of space, if we take into account that it is possible to have minimal perfect hashing schemes whose output hash functions require only $O(n)$ bits [14]. Even the space of the latter being smaller than that of the former, it does not escape from the asymptotic $O(n\log n)$.

The practical, minimal, perfect and $O(n)$-expected time BDZ scheme [2], presented in 2007, not only achieves the $O(n)$ space to the representation of the constructed hash function but also gets this amount to be $2.62n$, just a little greater than $1.443n$. More recently, better practical hashing schemes were proposed [15, 16]. The one by Belazzougui, Botelho and Dietzfelbinger [15], known as CHD, generates in $O(n)$ time a minimal perfect $O(1)$-evaluation time hash function which requires just about 2.06 bits per key. The authors' experiment show that CHD is more efficient than other schemes concerning to running and evaluation time too. Nevertheless, one can still set CHD parameters to obtain better results according to each application. For example, if one does not need the hash function to be minimal, CHD can map without collision the n keys to addresses $1,\dots,m$, where $m = 1.23n$, in a way that the generated hash function requires about 1.4 bits per key. Evenmore, if one sets $m = 2n$, one gets 0.67 bits per key. CHD also shows up very efficient for k-perfect hashing, where at most k keys can be mapped to the same address. BMZ, BDZ, CHD and other hashing schemes were implemented in CMPH (C Minimal Perfect Hashing) library, developed and maintained at `SourceForge.net` by the authors themselves.

Below, for sake of completeness, we give a short review of the BMZ and BDZ schemes, though we strongly recommend [1, 2] for more details.

2.1 BMZ

This is a hashing scheme proposed in [1] which constructs in $O(n)$ expected time a minimal perfect hash function given a set S with n keys. It maps S to the set $E(G)$ of the n edges of a graph G on $1.15n$ vertices and then tries to find a way to assign labels to the vertices so that the edges labels, defined to be the sum of endpoints labels, will be the whole set $\{0,\dots,n-1\}$. Two properties about G are required:

Property $\mathscr{P}_1$: The critical subgraph of G, denoted by G_{crit}, must be connected.

Property $\mathscr{P}_2$: $|E(G_{\text{crit}})| \leqslant \frac{1}{2}|E(G)|$.

Both Properties $\mathscr{P}_1$ and $\mathscr{P}_2$ occur in a random graph on $1.15n$ vertices and n edges with probability $p \approx 1$ [1].

BMZ is a three-step hashing scheme: first is the *mapping step*, when the keys are mapped to the edges of a graph; second is the *ordering step*, when G_{crit} is found; and third is the *searching step*, when the edges are labeled, starting at those in G_{crit}.

In the *mapping step* the set S is mapped to the edge set of a random graph G by picking up two random functions $h_1, h_2 \colon S \to V(G)$, so a key $x \in S$ is mapped to the edge

$$e(x) = \begin{cases} \{h_1(x), h_2(x)\}, & \text{if } h_2(x) \neq h_1(x), \\ \{h_1(x), 2h_1(x)+1\}, & \text{otherwise,} \end{cases} \tag{1}$$

and, if we have $e(x) = e(y)$ for distinct x and y, we simply pick up another pair of functions (h_1, h_2). The expected number of iterations of this procedure is about 2.13 [1].

In the *ordering step* BMZ finds G_{crit} by successively removing from G the edges incident to vertices of degree at most 1. See Example 1 below.

In the *searching step* BMZ first labels the edges in G_{crit}. After that, the other ones can receive the non-used labels in the *critical part*. Labeling the critical edges can be done by a simple greedy strategy performed in a breadth-first search on G_{crit}. We assign to the initial vertex u_0 of the search the label $g(u_0) = 0$ and initialize a counter variable i with 1, so each searched vertex $v \neq u_0$ is labeled with $g(v) = i$, whereupon i is incremented each time is used. Whenever we assign the label i to a vertex and this assignment *fails*, we increment i and try again on the same vertex, never changing earlier assigned labels. As each edge label $h(\{u,v\})$ is the sum $g(u)+g(v)$, an assignment can *fail* if, when trying to assign i to v, we find a neighbor w of v such that the label $h(\{w,v\}) = g(w)+i$ collides with the label of another edge previously labeled.

Let us denote by N_{t} the total number of vertices *reassignments* in the search on G_{crit}, and by N_{bedges} the number of back edges of G_{crit} according to the search. In [1] the authors showed that under the hypotheses of Properties $\mathscr{P}_1$ and $\mathscr{P}_2$ and of

$$N_{\text{t}} \leqslant N_{\text{bedges}}, \tag{2}$$

the labeling procedure never causes an edge e to have $h(e) > n-1$, assuring minimality to the hash function we are constructing (see Theorem 1 below). Notwithstanding, if some edge e receives $h(e) > n-1$, because of the infinitesimal probability of G not satisfying some of Properties $\mathscr{P}_1$ and $\mathscr{P}_2$, the whole scheme is restarted. At the end of the process, the bijection $h \circ e \colon S \to \{0,\ldots,n-1\}$, which maps each x in S to an address between 0 and $n-1$, is the desired minimal perfect hash function.

Example 1 ([1])**:** As shown in the Figure 1, by successively removing edges incident to vertices of degree at most 1 we get

$$G_{\text{crit}} = \big(\{0,3,4,7,8\}, \big\{\{0,8\},\{8,3\},\{3,4\},\{4,8\},\{8,7\},\{7,0\}\big\}\big)$$

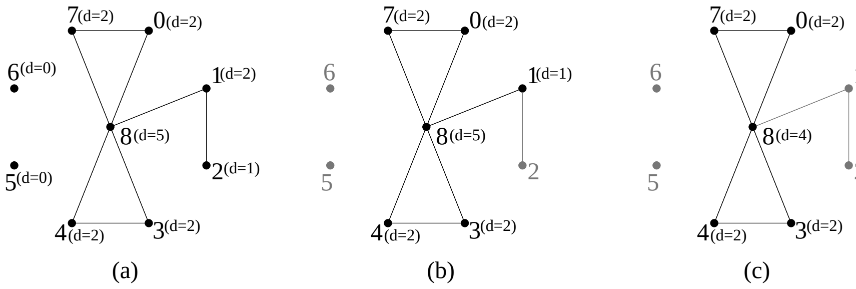

Figure 1. Finding the critical subgraph of a graph

For an example of BMZ searching step, let's start at vertex 8 a breadth-first search on G_{crit} indicated in Figure 1(c), assigning 0 to $g(8)$. Next vertex searched is 0, and we make $g(0) = i = 1$. Incrementing i, we search for 3, which is assigned to label $g(3) = i = 2$. Now $i = 3$, and 4 is searched and assigned smoothly to $g(4) = i = 3$. But, when searching for 7, if we make $g(7) = i = 4$, a collision occurs between labels $h(\{7,0\}) = 4+1$ and $h(\{3,4\}) = 2+3$. Hence, we try a *reassignment* incrementing i. Anyhow, if we make $g(7) = i = 5$, a collision takes place between $h(\{8,7\})$ and $h(\{3,4\})$. Another reassignment comes to pass, but this time, making $g(7) = i = 6$, we finally get the labels to all critical edges, as Figure 2(a) illustrates. Now we can simply perform a depth-first search on the non-critical edges to assign them the labels in $0,\ldots,n-1$ not used for G_{crit}, in our example 0 and 4, as shown in Figure 2(b).

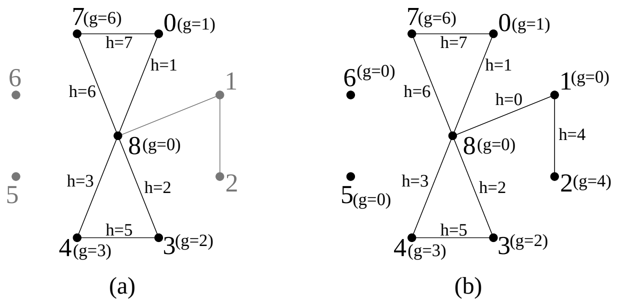

Figure 2. Searching for minimal perfect hash function h

The core result obtained in [1] runs as follows.

Theorem 1 (Botelho, Kohayakawa and Ziviani [1])**:** *If $N_{\text{t}} \leqslant N_{\text{bedges}}$, and if G_{crit} satisfies both Properties $\mathscr{P}_1$ and $\mathscr{P}_2$, then the maximum label $\max A_E$ assigned to a critical edge by BMZ searching step is at most $n-1$.*

Sketch of proof. As variable i is incremented $|V(G_{\text{crit}})| + N_{\text{t}}$ times, the biggest value assigned to a critical vertex is at most $|V(G_{\text{crit}})| + N_{\text{t}} - 1$, the second biggest value is at most $|V(G_{\text{crit}})| + N_{\text{t}} - 2$. Thus, $\max A_E \leqslant 2|V(G_{\text{crit}})| - 3 + 2N_{\text{t}}$. If $N_{\text{t}} \leqslant N_{\text{bedges}}$, then $\max A_E \leqslant 2|V(G_{\text{crit}})| - 3 + 2N_{\text{bedges}}$. G_{crit} is connected, so the number of tree edges in the search is $|V(G_{\text{crit}})| + 1$, and, consequently, $N_{\text{bedges}} = |E(G_{\text{crit}})| - |V(G_{\text{crit}})| + 1$. Therefore, $\max A_E \leqslant 2|E(G_{\text{crit}})| - 1$, and the theorem follows from $|E(G_{\text{crit}})| \leqslant \frac{1}{2}|E(G)|$ and $|E(G)| = n$. ☐

2.2 BDZ

This is a hashing scheme proposed in [2] which constructs in $O(n)$ expected time a minimal perfect hash function that requires only about 2.62 bits per key, very close to the tight lower bound result of about 1.443 bit per key. It maps S to the edge set of a 3-partite 3-hypergraph G with $t = 1.23n$ vertices and n edges. Though BDZ is originally defined to be performed using a r-partite r-hypergraph for any $r \geqslant 2$, the best results are obtained when $r = 3$. BDZ is not a generalization nor an expansion of BMZ. While in BMZ the label of an edge is the sum of the labels of the vertices belonging to that edge, in BDZ the label of an edge is *the* label of the vertex *assigned* to that edge, as we shall define. Remark that BMZ does not require the graph to be bipartite.

Besides of 3-partiteness, BDZ requires the 3-hypergraph G another property:

Property $\mathscr{P}_3$: The edges set $E(G)$ must be orderable in a list $\boldsymbol{L} = [e_1, \ldots, e_n]$ such that every edge e_j has at least one vertex not belonging to any $e_{j'}$ for all $j' > j$.

Every acyclic hypergraph satisfies this property, of course. Furthermore, according to [17, 3], acyclicness occurs in a random 3-partite 3-hypergraph on $1.23n$ vertices and n edges with probability $p \approx 1$. Thus, in its *mapping step*, BDZ use three random functions $h_j \colon S \to V_j$, for $j = 0, 1, 2$, to map each key x to the edge $\{h_0(x), h_1(x), h_2(x)\}$, where

$$V_j = \left\{ \left\lfloor \frac{jt}{3} \right\rfloor, \ldots, \left\lfloor \frac{(j+1)t}{3} - 1 \right\rfloor \right\} \tag{3}$$

are the parts of $V(G)$. The hypergraph G cannot contain multiple edges, so we redraw the functions h_j when $\{h_0(x), h_1(x), h_2(x)\} = \{h_0(y), h_1(y), h_2(y)\}$ for distinct x and y. As exemplified in Figure 3, ordering the edges in list $\boldsymbol{L}$ can be done by simply removing successively from G the edges incident to vertices of degree at most 1, until we have no edges to remove. If edges remain such that none of them can be removed, the mapping step draw another set of random functions. In [2] was shown that the probability of redrawing h_j, due to multiple edges or due to fail while ordering edges in $\boldsymbol{L}$, is $p \approx 0$.

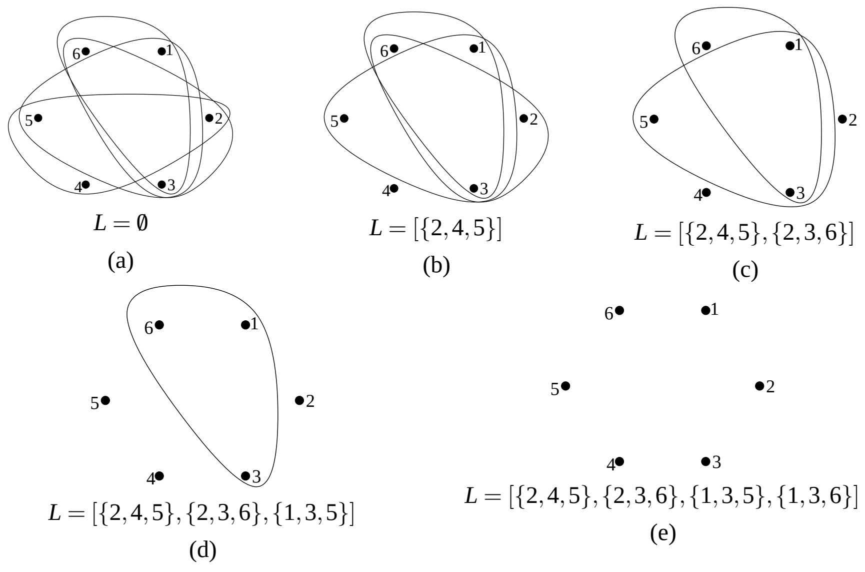
6
1
5
2
4
3
$L = \emptyset$
(a)
$L = [\{2,4,5\}]$
(b)
$L = [\{2,4,5\},\{2,3,6\}]$
(c)
$L = [\{2,4,5\},\{2,3,6\},\{1,3,5\}]$
(d)
$L = [\{2,4,5\},\{2,3,6\},\{1,3,5\},\{1,3,6\}]$
(e)

Figure 3. Ordering the edges of a hypergraph in a list, according to Property $\mathscr{P}_3$

BDZ *assigning step* finds a function ρ which maps injectively $E(G)$ to $V(G)$, in order to assign the vertex $\rho(e)$ to the edge e. We get ρ by traversing the edges in the reverse order

$e_n, \ldots, e_1$ with respect to $\boldsymbol{L}$. Every vertex is initially labeled with 3. Then, for each edge $e = \{u_0, u_1, u_2\}$ traversed, where $u_0 \in V_0$, $u_1 \in V_1$ and $u_2 \in V_2$, we take exactly one vertex $u_i \in e$ not yet *visited*, labeling it with

$$g(u_i) = \left(i - \sum_{\substack{v \in e_j \\ v \text{ visited}}} g(v) \right) \bmod 3, \tag{4}$$

and then *visit* all vertices in e. At the end of assigning step the sum modulo 3 of the labels for all vertices in an edge $e = \{u_0, u_1, u_2\}$ will be the index i of the vertex which is assigned to e by function ρ. As well, a label $g(u)$ of a vertex u is not equal to 3 if and only if u is assigned to an edge. In other words, if and only if u is image of some edge by the function ρ.

Compounding mapping and assignment steps gives us a perfect hash function for set S. Each key, mapped to an edge of the hypergraph G, is mapped by ρ to an address in $V(G) = \{0, \ldots, 1.23n - 1\}$. For the sake of making minimal this hash function, BDZ *ranking step* computes a rank table to achieve a function rank: $V(G) \to \{0, \ldots, n-1\}$, injective for $\rho(E(G))$, defined by $\text{rank}(u) = |\{v \in V(G) : v < u \text{ e } g(v) \neq 3\}|$. The authors suggest the work of [18] for implementing efficiently this rank table.

3 A simple strategy for derandomizing BMZ and BDZ

In mapping step BMZ scheme draws the functions $h_1, h_2 \colon S \to \{0, \ldots, t-1\}$, for $t = 1.15n$, by filling randomly two tables T_1 and T_2. Each table has $L \times |\Sigma|$ numbers in the set $\{0, \ldots, t-1\}$. Recall that a key $x \in S$ is a sequence $x = x_1 x_2 \cdots x_{|x|}$ of $|x| \leqslant L$ symbols in Σ, thus the lines of the tables correspond to the positions in the key, as the columns to the symbols. Thus, for $j = 1, 2$, the value of $h_j(x)$ is defined by

$$h_j(x) = \left(\sum_{i=1}^{|x|} T_j[i, x_i] \right) \bmod t. \tag{5}$$

We can show [1, Section 3.1] that the probability of a pair (h_1, h_2) giving a simple graph is about $\mathrm{e}^{-1/1.15^2} \cong 0.469$. This means that approximately 0.469 of all possible pairs (T_1, T_2) are *good* pairs: they generate h_1 and h_2 whereby the graph obtained doesn't have loops or multiple edges. Our deterministic version of BMZ simply establishes an ordering to search for all possible pairs (T_1, T_2) in a way that the expected number of probes until finding a *good* one is at most 3.

From now on, we look at each pair (T_1, T_2) as a number T with $2L|\Sigma|$ digits in base t:

the j-th digit, $0 \leqslant j < 2L|\Sigma|$, is $T_a[b,c]$, where the a, b and c are given by:

$$\begin{aligned}
a &= \begin{cases} 1, & \text{if } j < L|\Sigma|; \\ 2, & \text{otherwise;} \end{cases} \\
b &= \begin{cases} \left\lfloor \dfrac{j}{|\Sigma|} \right\rfloor + 1, & \text{if } j < L|\Sigma|; \\ \left\lfloor \dfrac{j}{|\Sigma|} \right\rfloor + 1 - L, & \text{otherwise;} \end{cases} \\
c &= (j \bmod |\Sigma|) + 1.
\end{aligned} \tag{6}$$

Moreover, we need a constant with $2L|\Sigma|$ digits such that added successively to T modulo $N+1$ gives all possible $N+1$ numbers with $2L|\Sigma|$ digits. We show in Proposition 1 that if $t-1$ is divisible by 3, then we can fix the constant as $N/3$. This implies that we must consider

1. $t = \min\{m \in \mathbb{N} \colon m \geqslant 1.15n \text{ and } m-1 \text{ is divisible by } 3\}$ instead of $1.15n$;
2. that, if a pair of tables is not *good*, we take next simply adding $(t-1)/3$ modulo t to each position in table, propagating carry from each position to another, in view of $N/3$ in base t has all digits equal to $(t-1)/3$.

In Proposition 1, we assume without loss of generality that the first number in sequence σ is 0, but actually, in practice, our first pair of tables is obtained by filling the tables with $0,1,2,\dots,t-1,0,1,2,\dots$

Proposition 1: *If* $t-1$ *is divisible by* 3*, and if* $\sigma_0 = 0$*, and if*

$$\sigma_{j+1} = \left(\sigma_j + \frac{N}{3}\right) \bmod (N+1) \tag{7}$$

for all $j > 0$*, then* $\{\sigma_0, \dots, \sigma_N\} = \{0, \dots, N\}$.

Proof. Let us suppose that $t-1$ is divisible by 3, thus

$$N = (t-1)t^0 + (t-1)t^1 + (t-1)t^2 + \cdots + (t-1)t^{2L|\Sigma|-1} \tag{8}$$

also is divisible by 3, furthermore $N/3$ is an integer whose prime factors are factors of N. Moreover, 3 is the only prime factor of N that might not be factor of $N/3$. Because none of the primes which divide N divides $N+1$, we have that $N/3$ and $N+1$ are coprimes.

For all $j \in \{0,\dots,N\}$, $\sigma_j = (jN/3) \bmod (N+1)$. But $N+1$ and $N/3$ are coprimes, and $N+1$ never divides $jN/3$ for any $0 < j \leqslant N$. By consequence, $\sigma_j \neq 0$ for all $0 < j \leqslant N$.

Now, let us assume that there exists some repetition in $\{\sigma_0, \ldots, \sigma_N\}$, being σ_j the first repeated element, equal to some $\sigma_{j'}$ for some $j' < j$. We must have $j' > 0$. However,

$$\left((j - j')\frac{N}{3}\right) \bmod (N+1) = 0, \tag{9}$$

and $\sigma_{j-j'} = 0$, a contradiction, since $j - j' > 0$. $\square$

Remark that Proposition 1 means that taking successively next pair of tables (T_1, T_2) when one fails guarantees us that no pair will be repeated until all pairs have been taken. Evenmore, as 0.469 of the pairs are *good*, we might affirm that in the average case our deterministic approach finds a *good* pair in about 2.13 iterations. Trying to make one pair different the most from the next, till we have similar pairs only after $3 > 2.13$ steps, can be viewed as a way of *jumping* enough in the search space, since we can fairly believe that small changes in the tables T_1 and T_2 would not make great differences in the structure of the generated graph.

We use the very same strategy for derandomizing BDZ. According to [2], BDZ uses the works of [19, 20] to draw h_0, h_1 and h_2 by filling with random bits a matrix $\mathsf{A}_{\gamma\times L}$, where γ is a constant. The matrix A gives a function h' which maps each key x to a chain $h'(x)$ of γ bits, from which we obtain h_0, h_1 and h_2 as follows: if $\boldsymbol{x}$ is the binary representation of key x, $h'(x)$ is given by $h'(x) = \mathsf{A}\boldsymbol{x}^{\mathrm{T}}$, and each h_j, for $j = 0, 1, 2$, is given by

$$h_j(x) = \left(h'(x)[j\beta..(j+1)\beta - 1]\right) \bmod \left(\frac{t}{3}\right) + j\left(\frac{t}{3}\right), \tag{10}$$

where β is the chosen constant that defines the number of bits used from h' for computing each h_j. By the way, we use $h'(x)[a..b]$ to denote the natural number whose binary representation is the subchain of $h'(x)$ starting at position a and ending at position b.

As demonstrated in Proposition 2, ordering all possible $N+1$ matrices A in a deterministic sequence is achieved if $\gamma \cdot L$ is even. We likewise consider a matrix as a number with $\gamma \cdot L$ bits, and again add modulo $N+1$ each number in sequence to $N/3$, whose binary representation is $(0101\ldots01)$, as Equation 13 states. In other words, if a matrix is not *good*, we take next simply adding 1 modulo 2 to odd positions and 0 modulo 2 to even positions, propagating carry from each position to another. Proposition 2 assumes the first number of the sequence to be 0 by convenience, but the initial matrix is actually the one obtained by filling its cells with $100110011001\ldots$

Proposition 2: *If $\gamma \cdot L$ is even, and if $\sigma_0 = 0$, and if*

$$\sigma_{j+1} = \left(\sigma_j + \frac{N}{3}\right) \bmod (N+1) \tag{11}$$

for all $j > 0$, then $\{\sigma_0, \ldots, \sigma_N\} = \{0, \ldots, N\}$.

Proof. If $\gamma \cdot L$ is even, then

$$N = \sum_{j=0}^{\gamma \cdot L-1} 2^j = 3 \sum_{\substack{j=0 \\ j \text{ even}}}^{\gamma \cdot L-1} 2^j, \tag{12}$$

and, thereupon,

$$\sum_{\substack{j=0 \\ j \text{ even}}}^{\gamma \cdot L-1} 2^j = \frac{N}{3}. \tag{13}$$

Equation 13 means that N is divisible by 3, so the proof follows analogous to proof for Proposition 1. □

As a matrix A has probability $p \approx 0$ of being bad, since this is the probability of re-drawing h_j, almost all matrices in the deterministic ordering sequence generate a hypergraph without multiple edges satisfying Property $\mathscr{P}_3$.

4 A proof for Equation 2

Recall BMZ searching step labels the critical edges of a graph G by performing a breadth-first search on the critical subgraph of G. Each time an unlabeled vertex v is discovered, the search assigns to v the current value of the variable i, and if this assignment fails, i is incremented by one and a new attempt is made. We call each such attempt a *reassignment*. As we have transcribed in Theorem 1, BMZ authors [1] show that, inasmuch as G satisfies two properties about G_{crit} that almost all graph satisfies and Equation 2, which was left open in [1], holds, we will never label an edge with a value greater than $n-1$. Using Lemma 2, in Theorem 3 we present a proof for Equation 2, closing the theoretical analysis of BMZ.

Lemma 2: *In critical part of BMZ searching step, whenever we assign to a tree edge a label, say, j, for sure j is greater than any label of any other already labeled tree edge.*

Proof. Let us consider the moment in depth-first search the tree edge e is assigned to label j, no matter this assignment fails or not. This is the moment when some $v \in e$ is assigned to label i, where i is the counter variable of BMZ searching step on G_{crit}. Then $j = i + g(v_0)$, where v_0 is the parent of v in the search. As e is a back edge, $e = \{v_0, v\}$.

Suppose, for the sake of contradiction, that there is a previously labeled tree edge $\{v_1, v_2\}$ such that $h(\{v_1, v_2\}) = g(v_1) + g(v_2) \geqslant j$. We can assume

$$g(v_0) < g(v_1) < g(v_2) < i. \tag{14}$$

It follows that v_0 was dequeued before v_1, which was dequeued before v_2, which was dequeued before v, therefore, these vertices were queued respecting the sequence v_0, v_1, v_2, v. But when v_2 was queued, v_1 had already been dequeued, because v_1 is the parent of v_2 in search. Thus, when v_2 was queued, v_0 had already been dequeued, queueing v even before v_2 being queued, a contradiction. □

We proof $N_{\mathrm{t}} \leqslant N_{\mathrm{bedges}}$ by showing an injection from the set of reassignments to the set of back edges. Thereunto we take a set B, initially empty, and show a way how each reassignment puts in B a back edge which was not there before. As B is a subset of the set of all back edges, we have our injection. Our proof is an overlap of two proofs by induction, whereas outer induction is on reassignments and inner induction finds for each reassignment the back edge to add to B.

Theorem 3 (conjectured to be true in [1])**:** $N_{\mathrm{t}} \leqslant N_{\mathrm{bedges}}$.

Proof. Let r be a reassignment which occurs when assigning a value i to a vertex v fails. As v is obviously not the initial vertex of the search, let v_0 be the parent of v in search. The reassignment occurs due to a vertex w neighbor of v such that $g(w) + i = j = g(u_1) + g(u_2)$ for some edge $\{u_1, u_2\}$ previously labeled, where $g(u_1) < g(u_2)$ without losing of generality. We will demonstrate that r puts a back edge in B which was not there before.

If r is the first reassignment in the whole search and $w = v_0$, the edge $\{w, v\}$ is a tree edge and, from Lemma 2, $\{u_1, u_2\}$ is the back edge we put in B, at this time empty. Otherwise, if r is the first reassignment but $w \neq v_0$, then $\{w, v\}$ is the back edge we put in B, since $\{v_0, v\}$ is a tree edge.

If r is not the first reassignment, let's assume by induction that each reassignment r' before r satisfies the property of having put in B a back edge which has not been there yet:

1. If $w = v_0$, then edge $\{w, v\}$ is a tree edge and $f_0 = \{u_1, u_2\}$ is a back edge. If f_0 is already in B when r happens, it's because of another edge f_1 labeled after f_0 but before $\{w, v\}$ such that we had unsuccessfully tried to assign the label $h(f_0) = j$ to f_1. Analogously, if f_1 is already in B when r happens, it's due to another edge f_2 labeled after f_1 but before $\{w, v\}$ such that we had unsuccessfully tried to assign the label $h(f_1) > j$ to f_2. Inductively, there is an edge f_k, for some $k \geqslant 0$, which is not in B when r happens. Moreover, from Lemma 2, f_k is a back edge, because its label is $h(f_k) \geqslant j$. Therefore, f_k is the back edge we put in B.

2. Finally, if $w \neq v_0$, then edge $f_0 = \{w, v\}$ is itself a back edge. If f_0 is already in B when r happens, it's because of a reassignment before r when we had tried to assign to v the label $i' < i$ but it had failed due to an edge $f_1 \neq f_0$ for which $h(f_1) = g(w) + i' = j' < j$.

But if f_1 is already in B when r happens, it's because of another edge f_2 labeled after f_1 but before f_0 such that we had tried to assign to f_2 the label $h(f_1) = j'$. Analogously, if f_2 is already in B when r happens, it's due to another edge f_3 labeled after f_2 but before f_0 such that we had tried to assign to f_3 the label $h(f_2) > j'$. Inductively, there is an edge f_k, for some $k \geqslant 0$, which is not in B when r happens. Moreover, from Lemma 2, f_k is a back edge, because its label is $h(f_k) \geqslant j'$. Therefore, f_k is the back edge we put in B. $\square$

5 Empirical results

BMZ and BDZ implementations in CMPH do not draw the functions in mapping step as described above. Both use the practical Jenkins hash functions [21] instead of tables T_1, T_2 and matrix A. Given a key x which is a string and a random seed of 32 bits, Jenkins program compute extremely fast three hash functions J_1, J_2 and J_3 that map x to three numbers of 32 bits each. Jenkins hash functions circumvent the problem of, in practice, keys being quite similar to each other, far from uniform distribution assumed in theory. For example, in case where keys are URLs, they follow a very specific pattern. As Jenkins functions use all bits in x to influence each bit in $J_i(x)$, they generally map even similar, whilst distinct, keys to very different addresses. BMZ drawing of h_1 and h_2 is actually drawing of a random seed for two Jenkins hash functions. BDZ drawing of h_0, h_1 and h_2 is actually drawing of a random seed for the three Jenkins hash functions. We extend our derandomization strategy for Jenkins hash functions by ordering all numbers of 32 bits: initial seed is defined to be $\lfloor n/3 \rfloor$, and $(2^{32} - 1)/3$ is the chosen coprime of 2^{32} which we add (modulo 2^{32}) to the current seed in each iteration.

We have tested our deterministic versions[3] of BMZ and BDZ by simply making small changes in the original source codes in CMPH library. Both original codes and our variants have been executed in the same machine, an AMD Athlon$^{\text{TM}}$ 3500+ with 64 kiB of L1 cache, 512 kiB of L2 cache and 1 GiB of RAM. Running the same algorithm many times for same input can be interesting for randomized algorithms, but useless for deterministic ones. We have decided therefore each test case to consist of 50 instances of key sets, and what we present is the arithmetic average of all 50 obtained results. Both original and deterministic versions have been input with the very same 50 key sets, so they could be compared fairly. Tests have been executed over sets with up to 25,000,000 keys, artificial distinct URLs generated by a script. Our results are shown in Table 1, where our deterministic versions of BMZ and BDZ are called respectively D-BMZ and D-BDZ. We have compared not only time, but also

[3]available at `http://professor.ufabc.edu.br/~jair.donadelli/D-BMZ-BDZ.tar.bz2`.

number of iterations of mapping step. One can observe that there is no significant difference between our deterministic versions and the original ones.

$n =$	6,250,000		12,500,000		25,000,000	
scheme	iterations	time (s)	iterations	time (s)	iterations	time (s)
BMZ	1.8800	30.4516	2.4400	70.6006	2.4200	151.2864
D-BMZ	2.3600	32.9100	1.9000	65.1682	1.8800	144.6608
BDZ	1	23.2462	1	47.8154	1	101.2546
D-BDZ	1	23.0818	1	49.0736	1	102.4540

Table 1. An empirical comparison between our algorithms and the original ones

6 Final remarks

Despite of more recent and better results like those in [15, 16], we pick up only BMZ and BDZ schemes to derandomize, though we believe that our simple strategy can as well be applied to other randomized hashing schemes. BMZ [1] and BDZ [2] are practical and time-efficient minimal perfect hashing algorithms. Moreover, BDZ hash functions require a very small amount of space to be stored. It is about only 2.62 bits per key, a result which is very close to the tight lower bound of about 1.44 bit per key [13, 14]. BMZ and BDZ are randomized algorithms, but we present a simple strategy for removing all randomness of both schemes, and the empirical results show to be equivalent to the original ones. Our goal in derandomizing these schemes was not, of course, to obtain better time results, as one could think. Our contribution to these important hashing schemes is a deterministic behavior. In particular, executions for the same input always produce the same output whereas, for randomized schemes two distinct executions for same input can produce distinct outputs. We believe this strategy can be useful for developing dynamic hashing schemes based on BMZ and BDZ.

Static hashing schemes, like BMZ and BDZ, construct a hash function given a static set S with n keys. A dynamic hashing scheme is a scheme where operations like insertion and deletion of keys in S are available [3]. Dynamic hashing schemes are very useful to model data structures, specially due to $O(1)$ lookup time, in contrast to $O(\log n)$ time in data structures based on trees [9]. A very known dynamic hashing scheme, presented in [22], is based on the classic deterministic static hashing scheme FKS [8], though the dynamic version is not deterministic. Actually, we cannot have deterministic dynamic hashing with $o(\log n)$ lookup time [22], but determinism in the static scheme can help to build the dynamic one.

Notwithstanding running in $O(n)$ expected time, both BMZ and BDZ do not have any guarantee of halting (albeit this event has probability tending to 0), since drawing of functions h_j in mapping step is done with replacement. Our deterministic approach also gives the schemes a theoretical finite worst-case time, as Table 2 exposes, because we never repeat a pair of tables (T_1, T_2) nor a matrix A, according to Propositions 1 (p. 64) and 2 (p. 65). Notice that, in deterministic BMZ, the worst case is that when we try all $(1-0.469)(N+1) = O(n^{2L|\Sigma|})$ bad pairs of tables (T_1, T_2) until finding a *good* one. For each pair of tables tried, one $O(n)$-time iteration of mapping step is executed, what leads us to the $O(n^{2L|\Sigma|+1})$ worst-case time to the whole scheme. In deterministic BDZ, on the other hand, the worst case is that when we try all $p(N+1)$ bad matrices A. But $p \approx 0$, and then $p(N+1) = O(1)$, what gives the $O(n)$ worst-case time.

Scheme	BMZ	D-BMZ	BDZ	D-BDZ
Best-case time	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Worst-case time	$+\infty$	$O(n^{2L\|\Sigma\|+1})$	$+\infty$	$O(n)$
Average-case time	$O(n)$	$O(n)$	$O(n)$	$O(n)$

Table 2. Theoretical time complexity comparisons between the schemes

Acknowledgment: We thank Fabiano C. Botelho, who pleasingly answered our mails and presented us the BDZ algorithm.

References

[1] F. C. Botelho, Y. Kohayakawa, and N. Ziviani. A practical minimal perfect hashing method. In *Proceedings of the 4th International Workshop on Efficient and Experimental Algorithms*, volume 3503 of *LNCS*, pages 488–500, Santorini Island, Greece, 2005. WEA05, Springer.

[2] F. C. Botelho, R. Pagh, and N. Ziviani. Simple and space-efficient minimal perfect hash functions. In *Proceedings of the 10th Workshop on Algorithms and Data Structures*, volume 4619 of *LNCS*, pages 139–150, Halifax, Canada, 2007. WADS07, Springer.

[3] Zbigniew J. Czech, George Havas, and Bohdan S. Majewski. Perfect hashing. *Theoretical Computer Science*, 182(1–2):1 – 143, 1997.

[4] L. Babai. Monte-carlo algorithms in graph isomorphism testing. *Université de Montréal Technical Report DMS*, (79):1–33, 1979.

[5] V. Kabanets and R. Impagliazzo. Derandomizing polynomial identity tests means proving circuit lower bounds. In *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*, STOC '03, pages 355–364, New York, NY, USA, 2003. ACM.

[6] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. PRIMES is in P. *Ann. of Math.*, 160(2):781–793, 2004.

[7] M. Agrawal and S. Biswas. Primality and identity testing via Chinese remaindering. *J. ACM*, 50(4):429–443 (electronic), 2003.

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

[9] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, England, 1990.

[10] E. A. Fox, Q. F. Chen, and L. S. Heath. A faster algorithm for constructing minimal perfect hash functions. In *Proceedings of the 15th Annual International Conference on Research and Development in Information Retrieval, Data Structures*, pages 266–273, Dublin, Ireland, 1992. ACM.

[11] G. Havas, B. S. Majewski, N. C. Wormald, and Z. J. Czech. Graphs, hypergraphs and hashing. In *Proceedings of the 19th International Workshop on Graph-Theoretic Concepts in Computer Science*, volume 790 of *LNCS*, pages 153–165, Utrecht, Netherlands, 1993. WG1993, Springer.

[12] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, 2004.

[13] M. L. Fredman and J. Komlós. On the size of separating systems and families of perfect hash functions. *SIAM J. Alg. Disc. Meth.*, 5(1):61–68, 1984.

[14] K. Melhorn. *Data Structures and Algorithms I: Sorting and Searching*. Springer, Berlin, Germany, 1984.

[15] Djamal Belazzougui, FabianoC. Botelho, and Martin Dietzfelbinger. Hash, displace, and compress. In Amos Fiat and Peter Sanders, editors, *Algorithms - ESA 2009*, volume 5757 of *Lecture Notes in Computer Science*, pages 682–693. Springer Berlin Heidelberg, 2009.

[16] M. Dietzfelbinger and R. Pagh. Succinct data structures for retrieval and approximate membership (extended abstract). In *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part I: Tack A: Algorithms, Automata, Complexity, and Games*, volume 5125 of *Lecture Notes in Computer Science*, pages 385–396. Springer, 2008.

[17] P. Erdős and A. Rényi. On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci.*, 5(187):17–60, 1960.

[18] R. Pagh. Low redundancy in static dictionaries with constant query time. *SIAM Journal on Computing*, 31(2):353–363, 2001.

[19] Noga Alon, Martin Dietzfelbinger, Peter Bro Miltersen, Erez Petrank, and Gábor Tardos. Linear hash functions. *J. ACM*, 46(5):667–683, 1999.

[20] N. Alon and M. Naor. Derandomization, witnesses for Boolean matrix multiplication and construction of perfect hash functions. *Algorithmica*, 16(4-5):434–449, 1996.

[21] B. Jenkins. Algorithm alley: Hash functions. *Dr. Dobb's Journal of Software Tools*, 22(9), september 1997.

[22] Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert E. Tarjan. Dynamic perfect hashing: upper and lower bounds. *SIAM J. Comput.*, 23(4):738–761, 1994.