Hash, displace, and compress

Djamal Belazzougui[1], Fabiano C. Botelho *[2], and Martin Dietzfelbinger**[3]

[1] Ecole Nationale Supérieure d'Informatique, ENI Oued Smar, Algiers, Algeria
d_belazzougui@ini.dz
[2] Department of Computer Engineering, Federal Center for Technological Education of Minas Gerais, Belo Horizonte, Brazil
fabiano@decom.cefetmg.br
[3] Faculty of Computer Science and Automation, Technische Universität Ilmenau, P.O.Box 100565, 98684 Ilmenau, Germany
martin.dietzfelbinger@tu-ilmenau.de

Abstract. A hash function h, i.e., a function from the set U of all keys to the range range $[m] = \{0, \ldots, m-1\}$ is called a perfect hash function (PHF) for a subset $S \subseteq U$ of size $n \leq m$ if h is 1–1 on S. The important performance parameters of a PHF are representation size, evaluation time and construction time. In this paper, we present an algorithm that permits to obtain PHFs with representation size very close to optimal while retaining $O(n)$ construction time and $O(1)$ evaluation time. For example in the case $m = 2n$ we obtain a PHF that uses space 0.67 bits per key, and for $m = 1.23n$ we obtain space 1.4 bits per key, which was not achievable with previously known methods. Our algorithm is inspired by several known algorithms; the main new feature is that we combine a modification of Pagh's "hash-and-displace" approach with data compression on a sequence of hash function indices. That combination makes it possible to significantly reduce space usage while retaining linear construction time and constant query time. Our algorithm can also be used for k-perfect hashing, where at most k keys may be mapped to the same value. For the analysis we assume that fully random hash functions are given for free; such assumptions can be justified and were made in previous papers.

1 Introduction

In this paper, we study the problem of providing perfect hash functions, minimal perfect hash functions, and k-perfect hash functions. In all situations, a "universe" U of possible keys is given, and a set $S \subseteq U$ of size $n = |S|$ of relevant keys is given as input. The range is $[m] = \{0, 1, \ldots, m-1\}$.

Definition 1. (a) *A function $h\colon U \to [m]$ is called a* ***perfect hash function (PHF)*** *for $S \subseteq U$ if h is one-to-one on S.* (b) *A function $h\colon U \to [m]$ is*

* Research partially supported by the Brazilian National Institute of Science and Technology for the Web (grant MCT/CNPq 573871/2008-6), and for the InfoWeb Project (grant MCT/CNPq/CT-INFO 550874/2007-0).
** Research partially supported by DFG grant DI 412/10-1.

called a ***minimal perfect hash function (MPHF)*** *for $S \subseteq U$ if h is a PHF and $m = n = |S|$.* (c) *For integer $k \geq 1$, a function $h\colon U \to [m]$ is called a* ***k-perfect hash function (k-PHF)*** *for $S \subseteq U$ if for every $j \in [m]$ we have $|\{x \in S \mid h(x) = j\}| \leq k$.*

"A hash function construction" consists in the following: We consider algorithms that for a given set S construct a (static) data structure D_S such that using D_S on input $x \in U$ one can calculate a value $h(x) \in [m]$, with the property that h is a PHF (MPHF, k-PHF) for S. The evaluation time should be constant.

Perfect hashing can be used in many applications in which we want to assign a unique identifier to each key without storing any information on the key. One of the most obvious applications of perfect hashing (or k-perfect hashing) is when we have a small fast memory in which we can store the perfect hash function while the keys and associated satellite data are stored in slower but larger memory. The size of a block or a transfer unit may be chosen so that k data items can be retrieved in one read access. In this case we can ensure that data associated with a key can be retrieved in a single probe to slower memory. This has been used for example in hardware routers [21]. Perfect hashing has also been found to be competitive with traditional hashing in internal memory [1] on standard computers. Recently perfect hashing has been used to accelerate algorithms on graphs [8] when the graph representation does not fit in main memory.

The algorithm that we present in this paper applies equally well to perfect hashing and k-perfect hashing. However, in the analysis and in experiments we will mostly concentrate on perfect hashing.

1.1 Space Lower Bounds

One of the most important metrics regarding PHFs is the space required to describe such a function. The information theoretic lower bound to describe a PHF was studied in [10, 17]. A simpler proof of such a bound was later given in [22].

There is an easy way to obtain quite good lower bounds starting from the simple formulas in [17, Theorem III.2.3.6 (a)]. There it was noted that the bit length of the description of a perfect hash function for S must be at least $\log\left(\frac{\binom{u}{n}}{\left(\frac{u}{m}\right)^n \cdot \binom{m}{n}}\right)$. Using a simple argument that involves estimating sums of the form $\sum_{0 \leq i < n} \log(1 - \frac{i}{m})$ by an integral one obtains the lower bounds $(m - n)\log(1 - \frac{n}{m}) - \log n - (u - n)\log(1 - \frac{n}{u})$ (for PHFs) and $-(u - n)\log(1 - \frac{n}{u}) - \log n$ for MPHF.

Considering $u \gg n$, for PHFs where $m = 1.23n$ this gives a value of approximately $0.89n$ bits per key, and the lower bound for MPHFs ($n = m$) is approximately $n/\ln 2 \approx 1.44n$ bits.

It does not seem to bee easy to derive similar bounds for k-perfect hashing and $km = (1+\varepsilon)n$. For $n = km$ (corresponding to MPHF with $k > 1$) one can argue similarly as in [17, Theorem III.2.3.6 (a)], to obtain the space lower bound $\log\left(\frac{\binom{u}{n}}{\binom{u/(n/k)}{k}^{n/k}}\right)$. This leads to the lower bound $-(u-n)\log(1-\frac{n}{u}) - \log n + (n/k)\cdot\log(k!/k^k)$, or, for $u \gg n$ and n large, the lower bound is $n\cdot(\log e + \log(k!/k^k)/k - o(1))$. For example for $k = 4$ we have the lower bound $n\cdot(\log e - 0.854) \approx 0.589n$; for $k = 8$ the bound is $\approx 0.355n$; and for $k = 16$ the lower bound is $\approx 0.208n$.

1.2 Known Constructions

Many constructions for perfect hash functions are known. The first construction of functions that need space $O(n)$ were by Schmidt and Siegel [25]. Hagerup and Tholey [13] gave an asymptotically optimal construction for MPHF, without assumptions about random hash functions, but the scheme does not seem to be practical. In 1999, Pagh [20] suggested an extremely clear suboptimal scheme (using only very simple hash functions) that needed $(2+\varepsilon)\log n$ bits per key (analyzed), in experiments it was shown that values near $0.35\log n$ bits per key could be achieved. By splitting tricks (see e.g. [6]) the space may be reduced to $\delta\log n$ per key, for arbitrary constant $\delta > 0$. The past five years have seen some surprising new developments. In [4] it was implicit how simple perfect hash functions can be constructed, and finally in [2] an explicit construction was given that, for very large n, m can construct a PHF with fewer than 2 bits per key in a very simple and lucid way. The last two constructions assume that random hash functions are given for free.

1.3 The Full Randomness Assumption

We assume throughout that for all ranges $[r]$ that occur in the construction we have access to a family of fully random hash functions with range $[r]$. This means that naming r and an index $q \geq 1$, say, there is a hash function $h_{r,q}\colon U \to [r]$ that is fully random on U, and so that all the hash functions with distinct pairs (r,q) are independent. Moreover, given $x \in U$ and r and q, the value $h_{r,q}(x)$ can be found in $O(1)$ time. This looks like quite a strong assumption, but there are simple ways (the "split-and-share trick") to justify such an assumption without wasting more than $n^{1-\Omega(1)}$ (bits of) space. For details, see [6].

In our experimental implementations, we did not use split-and-share, but just used simple indexed families of hash functions that are reported to behave very well in practice. "Indexed" here means that one can choose new hash functions whenever they are needed, while not using too much space. (More about the hash functions is said below in Appendix A.) In all constructions occurring here success can be checked at runtime. Whenever this is the case one may try to get by with cheap hash functions for several attempts and switch to the more sophisticated functions that guarantee the analysis to go through only if this did not succeed. (No such case occurred in the experiments.)

2 The Data Structure and its Construction

We start with the description of the PHF construction. (The MPHF construction is derived from a PHF with range $[(1+\varepsilon)n]$ by applying some standard compression tricks on the range, just as in [2].) The data structure consists of two levels. We choose some size r of an "intermediate" table. A "first level hash function" g maps U into $[r]$, and thus splits S into r "buckets"

$$B_i = \{x \in S \mid g(x) = i\},\ 0 \le i < r.$$

For each bucket there is a second hash function $f_i \colon U \to [m]$, picked by the construction algorithm from a sequence $(\phi_1, \phi_2, \phi_3, \ldots)$ of independent fully random hash functions. To name f_i, one only has to know the index $\sigma(i)$ such that $f_i = \phi_{\sigma(i)}$. We want to obtain a mapping $h \colon U \to [m]$ defined by

$$h(x) = f_{g(x)}(x) = \phi_{\sigma(g(x))}(x)$$

that is perfect for S. Since g and the family $(\phi_1, \phi_2, \phi_3, \ldots)$ are assumed to be given for free, the data structure only has to store the sequence $\Sigma = (\sigma(i), 0 \le i < r)$, and make sure that $\sigma(i)$ can be retrieved in $O(1)$ time.

Following Pagh [20], whose construction was based on an idea of Tarjan and Yao [26], we now sort the buckets in falling order according to their size, and then find a "displacement value" for the buckets one after the other, in this order. But while in Pagh's construction the displacements were values in $[r]$, and $r > n$ was required for the analysis, we deviate here from his approach, and utilize the power of our fully random functions. For each bucket B_i we find a suitable index $\sigma(i)$, in the following manner:

Algorithm 1 ***Hash, displace, and compress***

(1) Split S into buckets $B_i = g^{-1}(\{i\}) \cap S$, $0 \le i < r$;
(2) Sort buckets B_i in falling order according to size $|B_i|$
($O(n)$ time, since numbers to be sorted are small);
(3) Initialize array `T[0...m-1]` with 0's;
(4) for all $i \in [r]$, in the order from (2), do
(5) for $\ell = 1, 2, \ldots$ repeat forming $K_i = \{\phi_\ell(x) \mid x \in B_i\}$
(6) until $|K_i| = |B_i|$ and $K_i \cap \{j \mid \texttt{T[}j\texttt{]} = 1\} = \emptyset$;
(7) let $\sigma(i) =$ the successful ℓ;
(8) for all $j \in K_i$ let `T[`j`]` $= 1$;
(9) Transform $(\sigma(i))_{0 \le i < r}$ into compressed form, retaining $O(1)$ access.

(Clearly, if $B_i = \emptyset$, then $\sigma(i) = 1$.) The output of this procedure is the sequence $\Sigma = (\sigma(i), 0 \le i < r)$. By replacing the 0-1-valued array `T[0...m-1]` with an array of counters for counting from 0 to k and the obvious modifications we obtain an algorithm for constructing a k-perfect hash function.

We will see below that if $m = (1+\varepsilon)n$ for some constant $\varepsilon > 0$ then computing the sequence Σ will succeed in expected linear time. Also, from that analysis

it follows directly that with high probability the values $\sigma(i)$ can be bounded by $C \log n$ for some constant C, so that each number $\sigma(i)$ can be represented using $\log\log n + O(1)$ bits. Packing the numbers $\sigma(i)$ in fixed size frames of size $\log\log n + O(1)$ will lead to space requirements of $n(\log\log n + O(1))$ bits, while constant evaluation time for h is retained. (This idea for obtaining a MPHF, communicated to the third author by Peter Sanders [24], was explored in the master's thesis of Lars Dietzel [5].)

However, we can go one step further. We will show below that the numbers $\sigma(i)$, $0 \le i < r$, are small, in that $\sigma(i)$ is geometrically distributed with bounded expectation, and that $\mathbf{E}(\sigma(i))$ is bounded by a constant. From this it follows that by using a compression scheme like that described in [11] the whole sequence $(\sigma(i), 0 \le i < r)$ can be coded in $O(n)$ bits, in a way that random access is possible, and so the property is retained that h can be evaluated in $O(1)$ time.

3 Analysis

We show that our scheme has the theoretical potential to approximate the optimum space of $n \cdot \log e$ bits for a MPHF up to a small constant factor—if we use sufficiently strong coding schemes. The calculation is carried out under the assumption that $\varepsilon = 0$. Introducing $\varepsilon > 0$ in the calculations is easy and only has the effect that the space requirements *decrease*. (But note that the algorithm cannot simply be used for $\varepsilon = 0$, because then the construction time is $\Theta(n \log n)$ (coupon's collector's problem!) and we need $\Theta(n)$ hash functions. In order to obtain a MPHF, one has to use $\varepsilon > 0$ and add some compression method as mentioned in [2] or use $\varepsilon = 0$ and treat buckets of size 1 separately.) We will use the following, which is a special case of Jensen's inequality (since $\log x$ is a concave function).

Fact 1 $\mathbf{E}(\log_2(\sigma(i))) \le \log_2(\mathbf{E}(\sigma(i)))$.

3.1 Heavy Buckets

Let $\lambda = n/r$ be the load factor of the intermediate table. Then the average size of B_i is λ, and $|B_i|$ is $\mathrm{Bi}(n, 1/r)$-distributed, with expectation λ. Let $\mathsf{Poisson}(\lambda)$ denote the distribution of a random variable X with $\mathbf{Pr}(X = t) = e^{-\lambda} \cdot \lambda^t / t!$. It is well known that for $n, r \to \infty$ with $n/r = \lambda$ the binomial distribution $\mathrm{Bi}(n, 1/r)$ converges to $\mathsf{Poisson}(\lambda)$. In our cases, $\lambda \ge 1$ will be some constant integer[4], and n (and r) will be assumed to be large. In this case, for each fixed t and each $i \in [r]$:

$$\mathbf{Pr}(|B_i| = t) = \frac{e^{-\lambda} \cdot \lambda^t}{t!} \cdot (1 + o(1)). \tag{1}$$

Lemma 1. $\mathbf{Pr}(X \ge t) \le e^{-\lambda}(e\lambda/t)^t$, *for* $t \ge \lambda$. For a proof see [18, p. 97].

[4] The assumption that λ is integral is only made for notational convenience.

Using (1) and Azuma's inequality, it is easy to establish the following estimates.

Lemma 2. *For each fixed $t \leq 2\lambda + 2$, with high probability ($1 - n^{-c}$ for an arbitrary $c > 0$) we have:*

$$a_{\geq}(\lambda, t) = |\{i \mid |B_i| \geq t\}| \qquad = r \cdot \mathsf{Poisson}(\lambda, \geq t) \cdot (1 + o(1));$$
$$b_{\geq}(\lambda, t) = |\{x \in S \mid |B_{g(x)}| \geq t\}| = n \cdot \mathsf{Poisson}(\lambda, \geq t-1) \cdot (1 + o(1)).$$

Assume now $t = |B_i| \geq 2\lambda + 1$. At the moment when the algorithm tries to find a function $f_i = \phi_{\sigma(i)}$ for this bucket, at most $b_{\geq}(\lambda, t)$ keys are stored in $\mathtt{T}[0 \ldots m{-}1]$. By the remarks above and Lemmas 1 and 2 we have that $b_{\geq}(\lambda, t) = n \cdot \mathsf{Poisson}(\lambda, \geq t-1) \leq n \cdot e^{-\lambda}(e\lambda/(t-1))^{t-1}$.

This entails that the success probability when we try some ϕ_ℓ is bounded below by $\left(1 - (n/m) \cdot e^{-\lambda}(e\lambda/(t-1))^{t-1}\right)^t$. The latter bound is increasing in t, for $t \geq 2\lambda{+}1$. So the success probability is bounded below by $(1-(e/4)^\lambda)^{2\lambda+1}$. Thus, $\mathbf{E}(\sigma(i)) < (1-(e/4)^\lambda)^{-(2\lambda+1)}$, which is bounded by the constant $(1-(e/4))^{-3} < 31$. By Fact 1 we conclude $\mathbf{E}(\log(\sigma(i))) < \log(31) < 5$. Both the expected time for finding $\sigma(i)$ and the space for storing $\sigma(i)$ for these heavy buckets is bounded by $a_{\geq}(\lambda, t) \cdot O(1) < r \cdot \mathsf{Poisson}(\lambda, \geq 2\lambda) \cdot O(1) \leq e^{-d\lambda} \cdot n$, for some constant $d > 0$, which shows that asymptotically, for larger λ, the influence of these buckets is negligible.

3.2 Overall Analysis

In the following, we estimate $\mathbf{E}(\sum_{1 \leq i < r} \log(\sigma(i)))$. Assuming that g has been applied and the buckets B_i have been formed we define:

$$\begin{aligned}
T_t &= \text{number of buckets of size } t;\\
L_{=t} &= t \cdot T_t = \text{number of keys in buckets of size } t;\\
L_{\geq t} &= \text{number of keys in buckets of size } t \text{ or larger};\\
\beta_{=t} &= L_{=t}/n = \text{fraction of keys in buckets of size } t;\\
\beta_{\geq t} &= L_{\geq t}/n = \text{fraction of keys in buckets of size } t \text{ or larger}.
\end{aligned}$$

Assume in Algorithm 1 $s-1$ buckets of size t have been treated already, and bucket B_i is next. In the table $\mathtt{T}$ exactly $L_{\geq t+1} + (s-1)t$ keys have been placed. We check the hash values of keys in B_i one after the other; the probability that they all hit empty places in $\mathtt{T}$ is at least $\left(\frac{n-(L_{\geq t+1}+(s-1)t+(t-1))}{n}\right)^t = \left(1 - \beta_{\geq t+1} - \frac{st-1}{n}\right)^t$. Thus, $\mathbf{E}(\sigma(i)) \leq \left(\frac{1}{1-\beta_{\geq t+1}-\frac{st-1}{n}}\right)^t$. By Fact 1 we conclude $\mathbf{E}(\log(\sigma(i))) \leq t \cdot \log\left(\frac{1}{1-\beta_{\geq t+1}-\frac{st-1}{n}}\right)$. By summing over all buckets of size t we get

$$\sum_{i\,:\,|B_i|=t} \mathbf{E}(\log(\sigma(i))) \leq t \cdot \sum_{1 \leq s \leq L_{=t}} \log\left(\frac{1}{1 - \beta_{\geq t+1} - \frac{st-1}{n}}\right). \tag{2}$$

The sum in (2) we may estimate by an integral plus a correction term:

$$t \cdot \int_0^{L_{=t}} \log\left(\frac{1}{1-\beta_{\geq t+1}-\frac{xt-1}{n}}\right) dx + t \cdot (C_t - C_{t+1}), \tag{3}$$

where $C_t = \log(1/(1-\beta_{\geq t}+\frac{1}{n}))$.

To evaluate the integral, we substitute $y = 1-\beta_{\geq t+1}-\frac{xt-1}{n}$ and obtain

$$t \cdot \int_{1-\beta_{\geq t+1}+1/n}^{1-\beta_{\geq t}+1/n} \log\left(-\frac{n}{ty}\right) dy = n \cdot \int_{1-\beta_{\geq t}+1/n}^{1-\beta_{\geq t+1}+1/n} \log\left(\frac{1}{y}\right) dy. \tag{4}$$

Since $\int \log(1/y)\, dy = -y(\log y - \log e)$, we get from (2), (3), and (4) that

$$\sum_{i\,:\,|B_i|=t} \mathbf{E}(\log(\sigma(i))) \leq n \cdot [-y(\log y - \log e)]_{1-\beta_{\geq t}+1/n}^{1-\beta_{\geq t+1}+1/n} + t \cdot (C_t - C_{t+1}). \tag{5}$$

We sum (5) up to the first t_0 with $\beta_{t_0+1} = 0$, hence $C_{t_0+1} = -\log(1+\frac{1}{n})$, to obtain

$$\sum_{0\leq i<r} \mathbf{E}(\log(\sigma(i))) \leq n \cdot [-y(\log y - \log e)]_{1/n}^{1+1/n} + \sum_{1\leq t\leq t_0+1} C_t. \tag{6}$$

A little calculation, using $(1+\frac{1}{n})\ln(1+\frac{1}{n}) > \frac{1}{n}$, shows that

$$n \cdot [-y(\log y - \log e)]_{1/n}^{1+1/n} < n\log e - \log n - 2\log e. \tag{7}$$

This yields

$$\sum_{0\leq i<r} \mathbf{E}(\log(\sigma(i))) < n\log e - \log n - 2\log e + \sum_{1\leq t\leq t_0+1} C_t. \tag{8}$$

The C_t-sum depends on the distribution of the sizes of the buckets (and on ε). For $\varepsilon = 0$, the first summand C_1 equals $\log n$, which cancels against the term $-\log n$ in (7). (If $\varepsilon > 0$, it is not hard to verify that all C_t are $O(\log(1/\varepsilon))$.) For estimating $\sum_{2\leq t\leq t_0+1} C_t$ we must make an assumption about the sequence $\beta_2, \beta_3, \ldots$. In case of the Poisson distribution we have $C_t < \log(1/(1-\beta_{\geq t})) \leq \log(1/(1-\beta_{\geq 2})) = \log(1/e^{-\lambda}) = \lambda \log e$. As seen in Section 3.1, the total contribution of buckets of size $2\lambda+2$ and larger to $\sum_{0\leq i<r} \mathbf{E}(\log(\sigma(i)))$ is $\leq e^{-d\lambda} \cdot n$, and we may estimate $\sum_{2\leq t\leq 2\lambda+1} C_t$ roughly by $O(\lambda^2)$.

Summing up, we obtain the following result. (The estimate for $\varepsilon > 0$ is derived in a similar way.)

Theorem 1. *In Algorithm 1 we have* $\sum_{0\leq i<r} \mathbf{E}(\log(\sigma(i))) \leq n(\log e + e^{-d\lambda}) + O(\lambda^2)$.

The compression scheme from [11] will thus give an expected space bound of $n(\log e + O((\log(\lambda) + 1)/\lambda)) + O(\lambda^2)$ (the O-notation refers to growing λ), as shown in Appendix A.2. More sophisticated compression schemes for the index table `T`, which can reduce the effect of the fact that the code length must be an integer while $\log(\sigma(i))$ is not, will be able to better approximate the optimum $n \log e$.

Finally, we also obtained a simpler result that explicitly shows the dependence of construction time on λ and ε. Theorem 2 shows the result that is fully proved in Appendix B.

Theorem 2. *The algorithm "Compressed hash-and-displace" requires expected time* $O(n \cdot (2^\lambda + (1/\varepsilon)^\lambda)$ *and space* $O(\log(1/\varepsilon)n)$ *(no dependence on* λ*).*

4 Experimental Results

The purpose of this section is to evaluate the performance of the compressed hash-and-displace algorithm, referred to as CHD algorithm from now on. The implementation used for the experiments is described in Appendix A. We also compare it with the algorithm proposed by Botelho, Pagh and Ziviani [2], which is the main practical perfect hashing algorithm we found in the literature and will be referred to as BPZ algorithm from now on.

The experiments were carried out on a computer running the Linux operating system, version 2.6, with a 1.86 gigahertz Intel Core 2 processor with a 4 megabyte L2 cache and 4 gigabyte of main memory. The algorithms were implemented in the C language and are available at `http://cmph.sf.net` under the GNU Lesser General Public License (LGPL).

To compare the algorithms we used the following metrics: (i) The amount of time to generate PHFs or MPHFs. (ii) The space requirement for storing the resulting PHFs or MPHFs. (iii) The amount of time required by a PHF or an MPHF for each retrieval. All results are averages on 50 trials and were statistically validated with a confidence level of 95%.

In our experiments we used two key sets: (i) a key set of $5,000,000$ unique query terms extracted from the AllTheWeb[5] query log, referred to as AllTheWeb key set; (ii) a key set of $20,000,000$ unique URLs collected from the Brazilian Web by the TodoBr[6] search engine, referred to as URL key set; Table 1 shows the main characteristics of each key set, namely the shortest key length, the largest key length, the average key length in bytes and the key set size in megabytes.

[5] AllTheWeb (`www.alltheweb.com`) is a trademark of Fast Search & Transfer company, which was acquired by Overture Inc. in February 2003. In March 2004 Overture itself was taken over by Yahoo!.

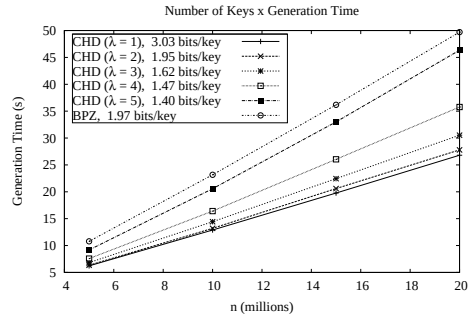
[6] TodoBr (www.todobr.com.br) is a trademark of Akwan Information Technologies, which was acquired by Google Inc. in July 2005.

Key Set	n	Shortest Key	Largest Key	Average Key Length	Key Set Size (MB)
AllTheWeb	5,000,000	2	31	17.46	91
URL	20,000,000	8	496	58.77	2,150

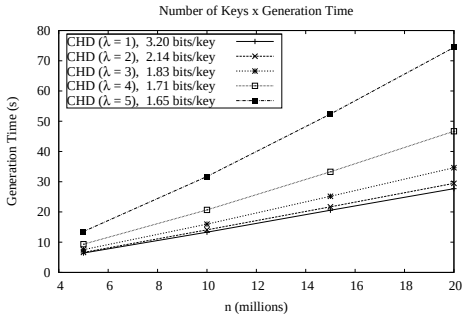
Table 1: Characteristics of the key sets used for the experiments.

4.1 Comparing the CHD and BPZ Algorithms

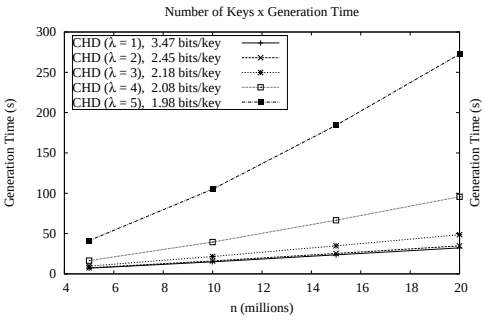
In this section we show that the CHD algorithm is very competitive in practice. It generates in linear time the most compact PHFs and MPHFs we know of and those functions can also be computed in constant time. We have experimented with four different values for the load factor: $\alpha = 0.81, 0.90, 0.99$ and 1.00. MPHFs are generated when $\alpha = 1.00$. For each α we vary the average number of keys per bucket (λ) in order to obtain a tradeoff between generation time and representation size. For the experiments, we have used the heuristic described in Appendix A.1 as it gave consistently better space usage and only degrades the generation time by 25%.



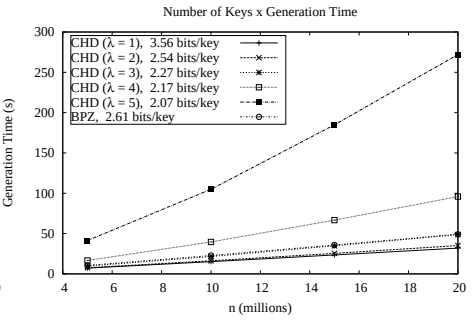
(a) $\alpha = 0.81$, Space lower bound = 0.88.



(b) $\alpha = 0.90$, Space lower bound = 1.07.



(c) $\alpha = 0.99$, Space lower bound = 1.38.



(d) $\alpha = 1.00$, Space lower bound = 1.44.

Fig. 1: Number of URLs versus generation time for both the CHD algorithm with $\lambda \in [1, 5]$ and $\alpha = 0.81, 0.90, 0.99$ and 1.00, and the BPZ algorithm with $\alpha = 0.81$ and 1.00.

Figure 1 shows a comparison with the BPZ algorithm [2]. We remark that the BPZ algorithm can generate PHFs only with $\alpha = 0.81$, whereas the CHD algorithm can achieve much higher load factors ($\alpha = 0.99$), see Figure 1 (a), (b) and (c). Moreover CHD algorithm is more flexible offering a full trade-off between space usage and load factors obtaining 0.67 bits per key for $\alpha = 0.5$ (we did not include this case in the experiments since we find those cases of higher load facors the most interesting) against 1.98 bits per key for $\alpha = 0.99$. To obtain MPHFs with $\alpha = 1.00$ both algorithms need to use a succinct data structure that supports rank and select operations and this requires a few more bits as

shown in Figure 1 (d). The CHD algorithm can be tuned to outperform the BPZ algorithm for both generation time and description size of the resulting functions. For instance, it is always the fastest algorithm when $\lambda \leq 3$. Also, with the CHD algorithm it is possible to obtain PHFs and MPHFs that are far from the information theoretic lower bound by a factor of 1.43 at the expense of spending more time to generate those functions. There is no other algorithm in the perfect hashing literature that can get so close to the information theoretic lower bound and even so to run linear time.

We now compare the CHD and BPZ algorithms considering the time to evaluate 5×10^6 keys of the AllTheWeb collection and 2×10^7 keys of the URL collection. Table 2 shows that the functions generated by the BPZ algorithm are slightly faster than the ones generated by the CHD algorithm. However, all functions require less than 0.8 microseconds when their description fit in cache, which is the case for the AllTheWeb collection, and less than 1.4 microseconds when their description do not fit in cache, which is the case for the URL collection. We remark that functions with a larger description size are slightly slower due to cache effects (more cache misses). The MPHFs are also slightly slower because they require the extra computation of rank and select operations.

Algorithms	λ	AllTheWeb ($n = 5 \times 10^6$) Evaluation Time (sec)			URL ($n = 2 \times 10^7$) Evaluation Time (sec)		
		$\alpha = 0.81$	$\alpha = 0.99$	$\alpha = 1.0$	$\alpha = 0.81$	$\alpha = 0.99$	$\alpha = 1.0$
CHD	1	3.53	3.59	4.14	24.06	24.24	26.99
	2	3.41	3.46	4.01	23.24	23.70	26.26
	3	3.40	3.49	4.04	22.71	23.47	26.05
	4	3.42	3.44	4.01	22.58	23.06	25.65
	5	3.43	3.45	4.02	22.41	22.98	25.59
BPZ	1	2.80	–	3.19	19.76	–	22.12

Table 2: Comparing the CHD and BPZ algorithms considering the time to evaluate 5×10^6 keys of the AllTheWeb collection and 2×10^7 keys of the URL collection.

The evaluation time of the PHFs and MPHFs generated by CHD algorithm depends on the compression technique used. For instance, it is possible to generate faster functions using Elias-Fano scheme (see [27]) instead of the one we used for the experiments [11] at the expense of generating functions with a slightly larger description size (we obtained PHFs that require 2.08 bits per key instead of 1.98 bits per key for $\alpha = 0.99$ and $\lambda = 5$).

4.2 Results for *k*-perfect hashing

In this section we present results showing that the CHD algorithm can generate very compact k-perfect hash functions. These functions can be used as an index stored in fast memory for data stored in slow memory. Usually, memory blocks can contain more than one element. In the case where the number of elements per block is a small constant k, we can construct a k-perfect hash function that uses less space than a perfect hash function. The main advantage of k-perfect hashing is that it requires a single random access to the slow memory in the worst case. This is not the case for other schemes like the linear hashing method proposed by Litwin [15] and the bucketed cuckoo hashing [9].

We have only experimented with load factor $\alpha = 0.99$ as this case is closer to the situation for which we derived lower bounds in Section 1.1. Table 4.2 presents

the results for generation time and space usage. We omited evaluation time as the results are too similar to the ones presented in Table 2.

k	AllTheWeb ($n = 5 \times 10^6$)						URL ($n = 2 \times 10^7$)					
	Generation Time(sec)			Space (bits/key)			Generation Time(sec)			Space (bits/key)		
	Avg. Bucket Size (γ)			Avg. Bucket Size (γ)			Avg. Bucket Size (γ)			Avg. Bucket Size (γ)		
	2	4	8	2	4	8	2	4	8	2	4	8
4	4.59	5.30	57.63	1.70	1.20	1.03	26.69	32.72	455.18	1.70	1.20	1.03
8	4.28	4.48	14.51	1.50	0.98	0.77	24.74	26.50	91.56	1.50	0.98	0.77
16	4.14	4.18	6.65	1.37	0.83	0.60	23.79	24.28	37.70	1.37	0.83	0.60

Table 3: Generation time and description size of k-perfect hash functions with $\alpha = 0.99$.

5 Conclusions

We have presented in this paper a novel approach for the compressed hash-and-displace method. The CHD algorithm generates the most compact PHFs and MPHFs we know of in $O(n)$ time. The time required to evaluate the generated functions is constant (in practice less than 1.4 microseconds). The storage space of the resulting PHFs and MPHFs are distant from the information theoretic lower bound by a factor of 1.43. The closest competitor is the algorithm by Martin and Pagh [7] but their algorithm do not work in linear time. Furthermore, the CHD algorithm can be tuned to run faster than the BPZ algorithm [2] (the fastest algorithm available in the literature so far) and to obtain more compact functions. The most impressive characteristic is that it has the ability, in principle, to approximate the information theoretic lower bound while being practical.

6 Acknowledgements

The third author wishes to thank Peter Sanders for sharing with him the idea of using random functions for the buckets in Pagh's hash-and-displace, which was subsequently explored in Lars Dietzel's M.Sc. thesis [5] at TU Ilmenau, which led to a solution with superlinear space.

References

1. F. C. Botelho, H. R. Langbehn, G. V. Menezes, and N. Ziviani. Indexing internal memory with minimal perfect hash functions. In *Proc. of the 23rd Brazilian Symposium on Database (SBBD'08)*, pages 16–30, October 2008.
2. F.C. Botelho, R. Pagh, and N. Ziviani. Simple and space-efficient minimal perfect hash functions. In *Proc. of the 10th Workshop on Algorithms and Data Structures (WADs'07)*, pages 139–150. Springer LNCS vol. 4619, 2007.
3. F.C. Botelho and N. Ziviani. External perfect hashing for very large key sets. In *Proc. of the 16th ACM Conference on Information and Knowledge Management (CIKM'07)*, pages 653–662. ACM Press, 2007.
4. Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, and Ayellet Tal. The bloomier filter: An efficient data structure for static support lookup tables. In *Proc. of the 15th annual ACM-SIAM symposium on Discrete algorithms (SODA'04)*, pages 30–39, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics.

5. L. Dietzel. Speicherplatzeffiziente perfekte hashfunktionen, November 2005. Master's Thesis (in German).
6. Martin Dietzfelbinger. Design strategies for minimal perfect hash functions. In *SAGA*, pages 2–17, 2007.
7. Martin Dietzfelbinger and Rasmus Pagh. Succinct data structures for retrieval and approximate membership. In *Proc. of the 35th international colloquium on Automata, Languages and Programming (ICALP'08)*, pages 385–396, Berlin, Heidelberg, 2008. Springer-Verlag.
8. Stefan Edelkamp, Peter Sanders, and Pavel Simecek. Semi-external ltl model checking. In *CAV*, pages 530–542, 2008.
9. Ulfar Erlingsson, Mark Manasse, and Frank Mcsherry. A cool and practical alternative to traditional hash tables. In *Proc. of the 7th Workshop on Distributed Data and Structures (WDAS'2006)*, pages 1–6, Santa Clara University, Santa Clara, California, USA, 2006.
10. M. L. Fredman and J. Komlós. On the size of separating systems and families of perfect hashing functions. *SIAM Journal on Algebraic and Discrete Methods*, 5:61–68, 1984.
11. Kimmo Fredriksson and Fedor Nikitin. Simple compression code supporting random access and fast string matching. In *Proc. of the 6th International Workshop on Efficient and Experimental Algorithms (WEA'07)*, pages 203–216, 2007.
12. Rodrigo Gonzalez and Gonzalo Navarro. Statistical encoding of succinct data structures. In *CPM*, pages 294–305, 2006.
13. T. Hagerup and T. Tholey. Efficient minimal perfect hashing in nearly minimal space. In *Proc. of the 18th Symposium on Theoretical Aspects of Computer Science (STACS'01)*, pages 317–326. Springer LNCS vol. 2010, 2001.
14. Bob Jenkins. Algorithm alley: Hash functions. *Dr. Dobb's Journal of Software Tools*, 22(9), september 1997.
15. Witold Litwin. Linear hashing: a new tool for file and table addressing. In *Proc. of the sixth international conference on Very Large Data Bases (VLDB'80)*, pages 212–223. VLDB Endowment, 1980.
16. B.S. Majewski, N.C. Wormald, G. Havas, and Z.J. Czech. A family of perfect hashing methods. *The Computer Journal*, 39(6):547–554, 1996.
17. K. Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*. Springer-Verlag, 1984.
18. M. Mitzenmacher and E. Upfal. *Probability and Computing*. Cambridge University Press, 2005.
19. Michael Mitzenmacher and Salil Vadhan. Why simple hash functions work: Exploiting the entropy in a data stream. In *Proc. of the nineteenth annual ACM-SIAM symposium on Discrete algorithms (SODA'08)*, 2008.
20. R. Pagh. Hash and displace: Efficient evaluation of minimal perfect hash functions. In *Workshop on Algorithms and Data Structures (WADS'99)*, pages 49–54, 1999.
21. B. Prabhakar and F. Bonomi. Perfect hashing for network applications. In *Proc. of the IEEE International Symposium on Information Theory*. IEEE Press, 2006.
22. J. Radhakrishnan. Improved bounds for covering complete uniform hypergraphs. *Information Processing Letters*, 41:203–207, 1992.
23. Kunihiko Sadakane and Roberto Grossi. Squeezing succinct data structures into entropy bounds. In *SODA*, pages 1230–1239, 2006.
24. P. Sanders. Personal communication, 2004.
25. J. P. Schmidt and A. Siegel. The spatial complexity of oblivious k-probe hash functions. *SIAM Journal on Computing*, 19(5):775–786, October 1990.
26. R. E. Tarjan and A. C. C. Yao. Storing a sparse table. *Communications of the ACM*, 22(11):606–611, November 1979.
27. Sebastiano Vigna. Broadword implementation of rank/select queries. In *Proc. of the 7th International Workshop on Efficient and Experimental Algorithms (WEA'08)*, pages 154–168, 2008.

A A Practical Version

In practice it has been observed that simple hash functions often behave similarly to really random hash functions (see [19] for a theoretical study of this fact). Therefore, we do not use real random hash functions, but just use the heuristic hash function proposed by Jenkins [14]. This function presents very good performance in practice [2, 3], and outputs a 12 byte long integer, which is an interesting property for our implemention described below.

Our implementation needs to find an injective mapping $x \to (g(x), f_1(x), f_2(x))$ for all $x \in S \subseteq U$, where $g : U \to [r]$, $f_1 : U \to [m]$, and $f_2 : U \to [m]$ are three hash functions computed by a single Jenkins' function, four bytes for each one of the three hash functions. It is well known that it is possible to obtain this injective mapping with probability $O(1 - 1/m)$, see, e.g, [16]. Thus, we will have to do $1 + o(1)$ trials on average.

Function g will map each key to one of the r buckets. Then, for each bucket B_i, $0 \leq i < r$, we will assign a pair of displacements (d_0, d_1) so that each key $x \in B_i$ is placed in an empty bin given by $(f_1(x) + d_0 f_2(x) + d_1) \bmod m$. For each bucket B_i we will try different pairs (d_0, d_1) until one of them successfully places all keys in B_i. In each trial we use a pair from the sequence $(0, 0), (0, 1), \ldots, (0, m-1), (1, 0), (1, 1), \ldots, (1, m-1), \ldots, (m-1, m-1)$. Instead of storing a pair (d_0, d_1) for each bucket B_i, we store the index of the first pair in that sequence that successfully places all keys in B_i.

A.1 Greedy Strategy to Reduce Entropy

We use the following greedy strategy to further reduce the entropy of the index sequence: first we try to place all buckets of a given size s using the index value 0, which represents the pair $(0, 0)$. Then we try to place the remaining buckets with the second index value 1, which represents the pair $(0, 1)$, and son on. Of course we maintain a list of non placed buckets, and each time we place a bucket, we remove it from the list.

The generation time for this greedy heuristic is longer by about 25% than when we try to place the buckets of a given size s one by one, exploring all possible pairs (d_0, d_1) when we are placing a given bucket B_i, $0 \leq i < r$. However, the greedy strategy allows to obtain functions that are up to 5% more compact.

A.2 Compression schemes

To compress the index values used to place the buckets there exist many methods that achieve space proportional to the empirical entropy with an additional sublinear space. In our case, we are only interested in entropy H_0. In our scheme this entropy is $O(1)$ bits per key. It is possible to find in the literature [23, 12] compression schemes that are able to compress an array of n values with

space $nH_0 + o(n)$ bits while supporting constant time access to any element of the array. However, the hidden constants in the $o(n)$ and $O(1)$ notation may not be so good in practice. For a more practical solution we can use the method proposed by Fredriksson and Nikitin [11]. This solution is particularly interesting for our case because it does not need to use any encoding or decoding tables.

Fredriksson-Nikitin Encoding In this scheme, we encode each number of the sequence separately using a variable length encoding. We store in a contiguous array all encoded numbers from the sequence of displacements. Then, we store the lengths of each encoded number using an Elias-Fano encoding. For example, we will encode the number 1 using 0 bits, the numbers 2 and 3 using 1 bit, the numbers 4,5,6,7 using 2 bits, and so on. More generally, we will use t bits to encode any number in range $[2^t, 2^{t+1} - 1]$. For a number x we will use $\lceil \log(x + 1) \rceil - 1$ bits.

Given a sequence of displacements $x_0, x_1, \ldots, x_{r-1}$. We first encode each x_i as described above giving a string y_i, and we collate all encoded strings giving the string $y_0 y_1 \ldots y_{r-1}$. Then we use an Elias-Fano to store the sequence $|y_0|, |y_1|, \ldots, |y_{r-1}|$. This will give us for any i, the beginning of encoding of y_i as well as $|y_i|$.

Using Elias-Fano encoding any sequence $s_0, s_1 \ldots, s_{m-1}$ will be encoded in at most

$$m \left(2 + \left\lceil \log \left(\frac{\sum_{i=0}^{m-1} s_i}{m} \right) \right\rceil \right)$$

bits. Thus, for a sequence $x_0, x_1, \ldots, x_{r-1}$, we will use a total of $Y + Z$ bits of space where we have

$$Y = \sum_{i=0}^{r-1} |y_i| = \sum_{i=0}^{r-1} (\lceil \log(x_i + 1) \rceil - 1)$$

and Z is at most

$$Z = r(2 + \lceil \log(Y/r) \rceil).$$

Now we can compute space usage by Fredriksson-Nikitin encoding of the sequence $\sigma(0), \sigma(1), \ldots, \sigma(r-1)$ with Theorem 1, which gives:

$$\sum_{0 \le i < r} \mathbf{E}(\log(\sigma(i))) \le n(\log e + e^{-d\lambda}) + O(\lambda^2).$$

On the other hand Fredriksson-Nikitin encoding uses space $Y + Z$ where Y is at most

$$\sum_{0 \le i < r} (\lceil \log(\sigma(i) + 1) \rceil - 1) \le \sum_{0 \le i < r} (\log(\sigma(i)))$$

So, from Theorem 1, expected value of Y is at most

$$n(\log e + e^{-d\lambda}) + O(\lambda^2) = n(\log e + O(1/e^{\lambda}))$$

We also have that Z is on expectation less than:

$$r(2 + \lceil \log(Y/r) \rceil) \leq r(2 + \lceil \log((n(\log e + e^{-d\lambda}) + O(\lambda^2))/r) \rceil)$$

Simplifying and replacing r by n/λ, we obtain that Z is at most:

$$\frac{n}{\lambda}(2 + \lceil \log(\lambda(\log e + e^{-d\lambda}) + O(\lambda^3/n)) \rceil).$$

Summing up, we get that $Y + Z$, the space used by Fredriksson-Nikitin encoding is in fact:

$$n(\log e + O((\log(\lambda) + 1)/\lambda)) + O(\lambda^2)$$

A.3 Input Parameters

We have implemented a version of the compressed hash-and-displace algorithm that can generate PHFs, MPHFs and k-PHFs. The algorithm takes a key set S as input as well as the following parameters:

1. λ: the number of keys per bucket. This parameter determines the tradeoff between description size and generation speed of the resulting functions. It is also used to compute the number of buckets via $r = \lceil \frac{n}{\lambda} \rceil$.
2. k: the maximum number of keys per bin when we are generating k-PHFs. It is set to 1 in case we are generating a PHF.
3. α: the hash table load factor. This parameter determines the number of bins (hash function range). In case of k-perfect hashing, the number of bins obeys the formula $m = \lceil \frac{n}{k \times \alpha} \rceil$.

B Simpler Analysis

We will see below that if $m = (1 + \varepsilon)$ for some constant $\varepsilon > 0$ then computing the sequence Σ will succeed in expected linear time. We will also show that the numbers $\sigma(i)$, $0 \leq i < r$, are small, in that $\sigma(i)$ is geometrically distributed with bounded expectation, so that $\mathbf{E}(\sigma(i))$ is bounded by a constant. From this it follows that by using a compression scheme like that described in [11] the whole sequence $(\sigma(i), 0 \leq i < r)$ can be coded in $O(n)$ bits, in a way that random access is possible, and so the property is retained that h can be evaluated in constant time.

Lemma 3. *[18, p. 97]. If X is $\mathsf{Poisson}(\lambda)$-distributed, where $\lambda \geq 1$ is an integer, then*

(a) $\mathbf{Pr}(X > \lambda) \leq \frac{1}{2}$;
(b) *for $t \geq \lambda$:* $\mathbf{Pr}(X \geq t) \leq e^{-\lambda}(e\lambda/t)^t$.

We will be interested in the following quantities:

$$a_=(\lambda,t) = |\{i \mid |B_i| = t\}|; \tag{9}$$
$$a_\geq(\lambda,t) = |\{i \mid |B_i| \geq t\}|; \tag{10}$$
$$b_=(\lambda,t) = |\{x \in S \mid |B_{g(x)}| = t\}|; \tag{11}$$
$$b_\geq(\lambda,t) = |\{x \in S \mid |B_{g(x)}| \geq t\}|. \tag{12}$$

Using (1) and Azuma's inequality, it is easy to establish the following estimates.

Lemma 4. *For each fixed $t \leq 2\lambda + 2$, with high probability ($1 - n^{-c}$ for an arbitrary $c > 0$) we have:*

$$a_=(\lambda,t) = r \cdot \mathsf{Poisson}(\lambda,t)(1 + o(1)); \tag{13}$$
$$a_\geq(\lambda,t) = r \cdot \mathsf{Poisson}(\lambda,\geq t)(1 + o(1)); \tag{14}$$
$$b_=(\lambda,t) = n \cdot \mathsf{Poisson}(\lambda,t-1)(1 + o(1)); \tag{15}$$
$$b_\geq(\lambda,t) = n \cdot \mathsf{Poisson}(\lambda,\geq t-1)(1 + o(1)). \tag{16}$$

For simplicity of notation, in the following calculations we will suppress the $1+o(1)$ factor and rounding effects. It will always be clear that the error made will be of the order of $o(1)$. Thus, from here we assume that $a_=(\lambda,t) = r \cdot \mathsf{Poisson}(\lambda,t)$ and $a_\geq(\lambda,t) = \mathsf{Poisson}(\lambda,\geq t)$, for $0 \leq t \leq 2\lambda + 2$.

In our analysis, we are interested in two quantities:

(a) The number of trials for bucket B_i until a suitable hash function $f_{\sigma(i)}$ is found. This is measured by $\mathbf{E}(\sigma(i))$.
(b) The space needed for storing the PHF h, i. e., for storing all numbers $\sigma(i)$. As an estimate for this, we take $\lfloor \log_2(\sigma(i)) \rfloor + 1$ the binary length of $\sigma(i)$, which can be upper bounded by $\log_2(\sigma(i)) + 1$. (There are simple methods for storing these numbers using space $2\sum_{i<r}(\log_2(\sigma(i))+1)$, that allow random access in $O(1)$ time, given e. g. in [11].) Thus, $2r + O(\sum_{i<r}(\log_2(\sigma(i))))$ is an upper bound for the space needed to store the PHF h. We let the random variable s denote $\sum_{i<r} \log_2(\sigma(i))$ and estimate $\mathbf{E}(s)$.

B.1 Heavy Buckets

The influence of heavy buckets is negligible as proved in Section 3.1.

B.2 Bucket Sizes Above Average

Now we turn our attention to buckets of size t, where $\lambda + 1 < t \leq 2\lambda$. Assume B_i is such a bucket. When the algorithm tries to find a hash function $f_i = \phi_{\sigma(i)}$, the number of occupied places in $\mathtt{T}[0 \ldots m-1]$ is no more than

$$b_\geq(\lambda,t) \leq b_\geq(\lambda,\lambda+2) \leq n \cdot \mathsf{Poisson}(\lambda,\geq \lambda+1) \leq \frac{1}{2}n,$$

by Lemma 3(a) and 4. This means that the success probability for any one of the functions ϕ_ℓ is at least $(1/2)^{2\lambda}$, a constant. Hence

$$\mathbf{E}(\sigma(i)) \leq 2^{2\lambda} \quad \text{and} \tag{17}$$

$$\mathbf{E}(\log(\sigma(i))) \leq 2\lambda. \tag{18}$$

The number of buckets of size in the range $[\lambda + 2, 2\lambda]$ is of course bounded by $r = n/\lambda$. Hence the expected number of functions tested for such buckets is $O(r \cdot 2^{2\lambda})$; if we charge cost 2λ for one trial (we must evaluate between λ and 2λ hash functions), the overall cost here is $O(n \cdot 2^{2\lambda})$. The expected number of bits needed for storing the indices $\sigma(i)$ for hash functions ϕ_i for B_i in this category is $O(r \cdot \lambda) = O(n)$.

We note that this general space bound is independent of λ. We expect the space requirements to decrease with growing λ, but our analysis is too coarse to make this difference. Further we see that the intermediate sized buckets will increase the construction time exponentially in λ. This seems unavoidable.

B.3 Bucket Sizes Below Average

Note that buckets of size $t < \lambda$ also may cause problems because the table may have run almost full, and the probability for a single hash value to succeed will be small already. For these buckets it is essential that m is larger than n by a constant factor $(1+\varepsilon)$. For simplicity, in the following analysis we ignore the cells that are empty because keys from smaller buckets have not yet been inserted.

Assume bucket B_i is to be treated, with $1 \leq t = |B_i| \leq \lambda + 1$. The success probability for any one of the functions ϕ_ℓ is at least $(\varepsilon/(1+\varepsilon))^{\lambda+1}$. Hence

$$\mathbf{E}(\sigma(i)) \leq \Big(\frac{1+\varepsilon}{\varepsilon}\Big)^{\lambda+1} = O((1/\varepsilon)^\lambda) \quad \text{and} \tag{19}$$

$$\mathbf{E}(\log(\sigma(i))) = O(\log(1/\varepsilon) \cdot \lambda). \tag{20}$$

Taking into account that there are $r = n/\lambda$ buckets overall, the expected number of bits to store the indices $\sigma(i)$ for small buckets B_i is $O(n \log(1/\varepsilon))$.

Remark. A more detailed analysis shows that the upper bound for $\mathbf{E}(\sigma(i))$ can be lowered to $O((1/\varepsilon)^\lambda)$ for $\lambda \geq 2$ and even to $O(\log(1/\varepsilon))$ for $\lambda = 1$.

Empty buckets are assigned the hash function with the shortest name, in the simplest case this is ϕ_1. Summing up, we obtain the result of Theorem 2.