Rate-distorted cryptographic perfect hash functions

A theoretical analysis on space efficiency, time complexity, and entropy

Alexander Towell
lex@metafunctor.com

Abstract

We analyze a theoretical *perfect hash function* that has three desirable properties: (1) it is a cryptographic hash function; (2) its *in-place* encoding obtains the theoretical lower-bound for the expected space complexity; and (3) its *in-place* encoding is a random bit string with maximum entropy.

Keywords: perfect hash function, random oracle, space complexity, maximum entropy, cryptographic hash function

Contents

1 Prior Art

Perfect hash functions have been extensively studied in the literature. Early foundational work by Dietzfelbinger et al. [4] established theoretical bounds for space-efficient hash tables with worst-case constant access time. Czech, Havas, and Majewski [3] introduced a family of perfect hashing methods that became widely influential.

More recent practical algorithms have focused on minimal perfect hash functions (MPHFs), which achieve a load factor of exactly 1. Botelho, Pagh, and Ziviani [2] presented simple and

space-efficient constructions, while Belazzougui, Botelho, and Dietzfelbinger [1] developed the Compress, Hash, and Displace (CHD) algorithm, which achieves excellent space-time tradeoffs for large datasets.

Our work differs by focusing on the theoretical properties of cryptographic perfect hash functions with maximum entropy encodings, rather than minimal space or construction speed.

2 Perfect hash functions

A *set* is an unordered collection of distinct elements. If we know the elements in a set, we may denote the set by these elements, e.g., $\{a, c, b\}$ denotes a set whose membes are exactly a, b, and c.

A *finite set* has a finite number of elements. For example, $\{1, 3, 5\}$ is a finite set with three elements. When sets $\mathbb{A}$ and $\mathbb{B}$ are *isomorphic*, denoted by $\mathbb{A} \cong \mathbb{B}$, they can be put into a one-to-one correspondence (bijection), e.g., $\{b, a, c\} \cong \{1, 2, 3\}$. Since there exists at least one bijection between isomorphic sets, we can losslessly convert one to the other and thus, isomorphic sets are in some sense equivalent.

The cardinality of a finite set $\mathbb{A}$ is the number of elements in the set, denoted by $|A|$, e.g., $|\{1, 3, 5\}| = 3$. A *countably infinite set* is isomorphic to the set of *natural numbers* $N = \{1, 2, 3, 4, 5, \ldots\}$.

Given two elements a and b, an ordered pair of a then b is denoted by (a, b), where $(a, b) = (c, d)$ *if and only if* $a = c$ and $b = d$. Ordered pairs are non-commutative and non-associative, i.e., $(a, b) \neq (b, a)$ if $a \neq b$ and $(a, (b, c)) \neq (b, (a, c))$.

Related to the ordered pair is the Cartesian product.

Definition 2.1. *The set* $\mathbb{X} \times \mathbb{Y} = \{(x, y) : x \in \mathbb{X} \wedge y \in \mathbb{Y}\}$ *is the Cartesian product of sets* $\mathbb{X}$ *and* $\mathbb{Y}$.

By the non-commutative and non-associative property of ordered pairs, the Cartesian product is non-commutative and non-associative. However, they are isomorphic, i.e., $\mathbb{X} \times \mathbb{Y} \cong \mathbb{Y} \times \mathbb{X}$.

A *tuple* is a generalization of order pairs which can consist of an arbitrary number of elements, e.g., $(x_1, x_2, \ldots, x_n)$.

Definition 2.2 (n-fold Cartesian product)**.** *The n-ary Cartesian product of sets* $\mathbb{X}[1], \ldots, \mathbb{X}[n]$*, is given by* $\mathbb{X}[1] \times \cdots \times \mathbb{X}[n] = \{(x_1, \ldots, x_n) : x_1 \in \mathbb{X}[1] \wedge \cdots \wedge x_n \in \mathbb{X}[n]\}$.

Note that $\mathbb{X}[1] \times \mathbb{X}[2] \times \mathbb{X}[3] \cong \mathbb{X}[1] \times (\mathbb{X}[2] \times \mathbb{X}[3]) \cong (\mathbb{X}[1] \times \mathbb{X}[2]) \times \mathbb{X}[3]$, thus we may implicitly convert between them without ambiguity.

If each set in the n-ary Cartesian product is the same, the power notation may be used, e.g., $\mathbb{X}^3 \equiv \mathbb{X} \times \mathbb{X} \times \mathbb{X}$. As special cases, the 0-ary (nullary) Cartesian product is defined to be $\{\emptyset\}$, and the 1-ary (unary) Cartesian product is the identity, e.g., $\mathbb{X}^1 = \mathbb{X}$.

The *hash function* is given by the following definition.

Definition 2.3. Hash functions *of type* $\mathbb{X} \mapsto \mathbb{Y}$ *are just total functions, normally with a finite codomain, with the weak assumption that they will be used as a device to assign elements from* $\mathbb{X}$ *a value from* $\mathbb{Y}$ *without requiring any particular rule.*

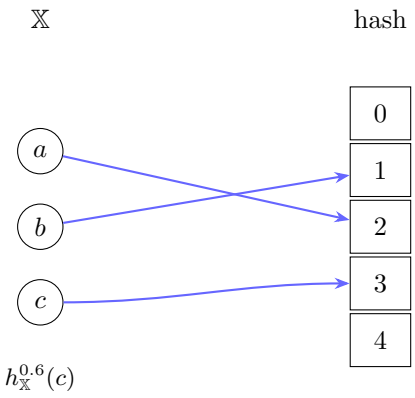
For a given bit string x and hash function hash, $y = \text{hash}(x)$ is denoted the *hash* of x.

We are particularly interested in *perfect* hash functions as given by the following definition.

Definition 2.4 (Perfect hash function)**.** *A* perfect hash function *over the set* $\mathbb{A} \subseteq \mathbb{X}$*, denoted by*

$$\text{h}_{\mathbb{A}} : \mathbb{X} \mapsto \mathbb{Y}, \tag{1}$$

Figure 1: A perfect hash function $\text{hash}_{\mathbb{X}}^{0.6} : \mathbb{U} \mapsto \{0, 1, 2, 3, 4\}$.



is an injective function when restricted to $\mathbb{A}$.[1]

Assumption 1. *Perfect hash functions are* surjective.

The *load factor* is given by the following definition.

Definition 2.5. *The proportion of hashes mapped to by* $\text{hash}_{\mathbb{A}} : \mathbb{X} \mapsto \mathbb{N}$ *over subset* $\mathbb{A}$ *is denoted the load factor. Specifically, the load factor of* $\text{hash}_{\mathbb{A}}$ *is a rational number given by*

$$\frac{|\text{hash}_{\mathbb{A}}(\mathbb{A})|}{|\text{image}(\text{hash}_{\mathbb{A}})|}. \tag{2}$$

Notation. *A perfect hash function of type* $\mathbb{X} \mapsto \mathbb{Y}$ *over* $\mathbb{A}$ *with a load factor* r *may be denoted by* $\text{hash}_{\mathbb{A}}^{r}$. *If* $m = |\mathbb{A}|$ *and we are interested in drawing attention to the cardinality of the perfect hash function, we may also denote it by* hash_{m}^{r}.

Example 1 *Consider the set* $\mathbb{X} = \{x_1, x_2, x_3\}$. *A perfect hash function of type* $\mathbb{U} \mapsto \mathbb{N}$ *over* $\mathbb{X}$ *with a load factor* $r = \frac{3}{5}$ *is denoted by* $\text{hash}_{\mathbb{X}}^{0.6}$ *or* $\text{hash}_{3}^{0.6}$. *Given the load factor* r *and* $\mathbb{X}$, *we may deduce the codomain of* $\text{hash}_{\mathbb{X}}^{r}$ *to be precisely* $\{0, 1, 2, 3, 4\}$. *See Figure 1 for an illustration.*

Definition 2.6. *A* minimal *perfect hash has a load factor* 1.

Every hash function in $\mathbb{X} \mapsto \mathbb{Y}$ is a perfect hash function over some subset of $\mathbb{X}$, e.g., every hash function is trivially a perfect hash function of $\emptyset$ and singleton sets.

Assuming $\mathbb{X}$ and $\mathbb{Y}$ are finite, the set of hash functions of type $\mathbb{X} \mapsto \mathbb{Y}$, which may also be denoted by $\mathbb{Y}^{\mathbb{X}}$, has a cardinality

$$|\mathbb{Y}|^{|\mathbb{X}|}. \tag{3}$$

The set of *perfect* hash functions over $\mathbb{A} \subseteq \mathbb{X}$ is a subset of $\mathbb{X} \mapsto \mathbb{Y}$ with the predicate that no collisions may occur on any pair of elements in $\mathbb{A}$. The set of perfect hash functions over $\mathbb{A}$ has a cardinality

$$\text{permutations}(|\mathbb{Y}|, |\mathbb{A}|)|\mathbb{Y}|^{|\mathbb{X}|-|\mathbb{A}|}. \tag{4}$$

[1] There are no collisions among elements of $\mathbb{A}$, $\text{hash}_{\mathbb{A}}(x) \neq \text{hash}_{\mathbb{A}}(y)$ for all $x, y \in \mathbb{A}$, $x \neq y$.

3 A theoretical *cryptographic* perfect hash function

The bit set $\{0,1\}$ is denoted by $\{0,1\}$. The set of all bit strings of length n is therefore $\{0,1\}^n$. The cardinality of $\{0,1\}^n$ is 2^n. In the case of $\{0,1\}$, we denote $\{0,1\}^0$ by ϵ. The set of all bit strings of length n or less is denoted by $\{0,1\}^{\leq n}$ with a cardinality $2^{n+1}-1$, e.g., $\{0,1\}^{\leq 2} = \epsilon \cup \{0,1\} \cup \{0,1\}^2$. The countably infinite set of all bit strings, $\lim_{n\to\infty}\{0,1\}^{\leq n}$, is denoted by $\{0,1\}^*$. A tuple $(x_1, x_2 \ldots) \in \{0,1\}^*$ is denoted a *bit string*, and we typically drop the angle brackets when specifying stings, e.g., $(x_1, x_2) \equiv x_1 x_2$.

An important operation that is closed over the free semigroup of $\{0,1\}$ is *concatentation*, $\# : \{0,1\}^* \times \{0,1\}^* \mapsto \{0,1\}^*$, which is defined as $a_1 \ldots a_n \# b_1 \ldots b_m = a_1 \ldots a_n b_1 \ldots b_m$ with special cases $x \# \epsilon = \epsilon \# x = x$.

Most hash functions are of the type $\{0,1\}^* \mapsto \{0,1\}^n$ where n is some finite natural number.

Another useful function is the binary *padding* function $\mathrm{pad} : \{0,1\}^* \times \mathbb{Z}_{\geq 0} \mapsto \{0,1\}^*$ is defined as $\mathrm{pad}(x,k) = x \# 0^{k-\mathrm{BL}(x)}$ with the special case $\mathrm{pad}(x,0) = \epsilon$.

The binary *truncation* function $\mathrm{trunc} : \{0,1\}^* \times \mathbb{N} \mapsto \{0,1\}^*$ is defined as $\mathrm{trunc}(a_1 \ldots a_k \ldots a_n, k) = a_1 \ldots a_k$, with the special case $\mathrm{trunc}(x,0) = \epsilon$.

The *bit length* function $\mathrm{BL} : \{0,1\}^* \mapsto \mathbb{N}$ is defined as $\mathrm{BL}(a_1 \ldots a_n) = n$, e.g., if $b \in \{0,1\}^n$ then $\mathrm{BL}(b) = n$.

Since $\{0,1\}^* \cong \mathbb{N}$, they may put into one-to-one correspondence. A convenient one-to-one correspondence between them is given by the following definition.

Definition 3.1. *The sets* $\{0,1\}^*$ *and* $\mathbb{N}$ *have a bijection given by*

$$b_1, \ldots, b_m \longleftrightarrow 2^m + \sum_{j=1}^{m} 2^{m-j} b_j. \tag{5}$$

We denote the mapping described by Definition 3.1 with the postfix function $' : \{0,1\}^* \mapsto \mathbb{N}$ and $' : \mathbb{N} \mapsto \{0,1\}^*$, which are inverse functions, i.e., $x'' = x$. An important observation of this mapping is that a natural number n maps to a bit string n' of length $\mathrm{BL}(n') = \lfloor \log_2 n \rfloor$.

More generally, if we have some set $\mathbb{U}$, in a physical computer there must be a way to map the elements of $\mathbb{U}$ to bit strings that *repesent* the values. We provide this mapping with *encoder* and *decoder* functions denoted respectively by $\mathrm{e}[\mathbb{U}] : \mathbb{U} \mapsto \{0,1\}^*$ and $\mathrm{d}[\mathbb{U}] : \{0,1\}^* \mapsto \mathbb{U}$ such that $\mathrm{d}[\mathbb{U}] \circ e_{\mathbb{U}} = \mathrm{id}_{\mathbb{U}}$ and $e_{\mathbb{U}} \circ d_{\mathbb{U}} = \mathrm{id}_{\{0,1\}^*}$.

A special case of the encoder and decoder is given by letting any bit string represent either the sequence of bits $a_1 \ldots a_n$ or as the binary coded decimal (BCD), e.g., $010 \longleftrightarrow 4$. Thus, any operation on non-negative integers may be applied to bit strings without ambiguity, e.g., $010 + 01 = 11$.

Given a function of $g : \mathbb{X} \mapsto \mathbb{Y}$, the *domain* of g may be denoted by $\mathrm{dom}(g) = \mathbb{X}$ and the *codomain* may be denoted by $\mathrm{codom}(g) = \mathbb{Y}$.

A random oracle as given by the following definition.

Definition 3.2. *A random oracle in the family* $\{0,1\}^* \mapsto \{0,1\}^\infty$ *is a theoretical* hash function *whose output is uniformly distributed over its codomain.*

The theoretical analysis of the perfect hash function makes the following assumption.

Assumption 2. *The hash function* $\mathrm{hash} : \{0,1\}^* \mapsto \{0,1\}^\infty$ *is a random oracle.*

Definition 3.3. *The* data type *for the* cryptographic *perfect hash function under consideration is defined as* $\mathrm{PH} = \{0,1\}^* \times \mathbb{N}$ *with a value constructor* $\mathrm{ph} : \mathcal{P}(\mathbb{U}) \times [0,1] \mapsto \mathrm{PH}$ *defined as*

$$\mathrm{ph}(\mathbb{X}, r) = (n', N) \tag{6}$$

where

$$\begin{aligned} m &= |\mathbb{X}|, \\ N &= \lceil m/r \rceil, \\ k &= \lceil \log_2 N \rceil, \\ \beta(x,n) &= \operatorname{trunc}\big(\operatorname{hash}(x' \# n'), k\big)' \mod N, \\ \mathbb{Y}_n &= \{\beta(x,n) \in \{0,\ldots,N-1\} | x \in \mathbb{X}\}, \\ n &= \min\{j \in \mathbb{N} | \mathbb{Y}_j \in 2^{\mathbb{N}} \wedge |\mathbb{Y}_j| = m\}. \end{aligned} \tag{7}$$

Since PH is a data type that purports to model perfect hash functions, its *computational basis* is given by the following set of functions.

By the assumption of surjectivity (and the perfect hash function is not rated-distorted), then the load factor is given by $\frac{|\mathbb{X}|}{N}$ where $N = \arg\max_{x \in \mathbb{X}} \operatorname{hash}_{\mathbb{A}}(x)$, i.e., the maximum hash (natural ordering of integers) of $\operatorname{hash}_{\mathbb{A}}$.

Definition 3.4. *The* minimum *and* maximum *hash of a value of type* PH *(that models a perfect hash function) are given respectively by* $\operatorname{min\_hash} : \mathrm{PH} \mapsto \mathbb{N}$ *and* $\operatorname{max\_hash} : \mathrm{PH} \mapsto \mathbb{N}$ *where*

$$\operatorname{min\_hash}(b, N) = 0 \text{ and } \operatorname{max\_hash}(b, N) = N - 1. \tag{8}$$

The most important function, the *perfect hash* mapping, $\operatorname{perfect\_hash} : \mathrm{PH} \times \mathbb{U} \mapsto \mathbb{N}$, is defined as

$$\operatorname{ph}((b, N), x) = q' \mod N \tag{9}$$

where

$$\begin{aligned} k &= \lceil \log_2 N \rceil, \\ r &= \operatorname{hash}(x' \# b), \\ q &= \operatorname{trunc}(r, k). \end{aligned} \tag{10}$$

Theorem 3.1. *A value of type* PH *constructed with* $\operatorname{ph}_{\{0,1\}^*}(\mathbb{A}, r)$ models $\operatorname{hash}^r_{\mathbb{A}} : \{0,1\}^* \mapsto \{0, 1, \ldots, k-1\}$ *where* $\mathbb{A} \subseteq \{0,1\}^*$ *and* $k = \frac{|\mathbb{A}|}{r}$.

Proof. Proof here. □

Suppose we have a function $f : \mathbb{U} \mapsto \{0,1\}^*$ such that $f|_{\mathbb{A}}$ is *injective*, e.g., an *encoder* for values of $\mathbb{A} \subseteq \mathbb{U}$. The composition $\operatorname{hash}_{\mathbb{A}} = \operatorname{hash}_{\mathbb{B}} \circ f$ where $\mathbb{B} = \{f(a) \in \{0,1\}^* | a \in \mathbb{A}\}$ is a perfect hash function of type $\mathbb{U} \mapsto \mathbb{N}$ over $\mathbb{A} \subseteq \mathbb{U}$. Thus, the rest of the material in this paper does not usually assume any particular domain of the perfect hash function since injections may always be constructed for any set.

3.1 Analysis

Note that since n' denotes the geometric code for n and we choose the smallest n that succeeds, where each choice of n is a geometrically distributed trial with probability p, the expected space complexity obtains the information-theoretic lower-bound of 1.44 bits per element.

We search *all* possible hash functions which are a function of hash, an approximate random oracle, and choose the *perfect hash function* on a specified set that has the *smallest* bit length. We describe the algorithm for performing this exhaustive search in Algorithm 1.

We consider a family of perfect hash functions for a set $\mathbb{S}$ which are given by the output of a hash function hash that approximates a *random oracle* applied to the input $x \in \mathbb{S}$ concatenated with a bit string b. We describe the generative algorithm for the perfect hash function in Algorithm 1. Note that in Algorithm 1, the concatenation of two bit strings x and y is denoted by $x \# y$.

The perfect hash function generated by Algorithm 1 has the statistical property that the output is uniformly distributed as given by the following theorem.

Theorem 3.2. *The perfect hash function* $\text{hash}[\mathbb{A}][r] : \mathbb{X} \mapsto \{0, 1, \ldots, k-1\}$*, where* $k = \frac{|\mathbb{A}|}{r}$*, is a random oracle over* $\mathbb{X} - \mathbb{A}$ *and the restriction* $\text{hash}^r_{\mathbb{A}}|_{\mathbb{A}} : \mathbb{A} \mapsto \{0, 1, \ldots, k-1\}$ *is a random* k*-permutation oracle of* $\mathbb{A}$*.*

Proof. First, in Algorithm 1 on Line 5, b_n is a bit string such that each $x \in \mathbb{S}$ concatenated with b_n hashes to a unique integer in $\{1, \ldots, N\}$ by the hash function hash, thus the hash function found perfectly hashes the elements of $\mathbb{S}$.

Second, by Definition 3.2, hash approximates a random oracle whose output is uniformly distributed over the elements of $\{0,1\}^n$. Viewing the elements of $\{0,1\}^n$ as integers, where $|\{0,1\}^n| = 2^n$, hash is uniformly distributed over $\{0, 1, \ldots, 2^n - 1\}$.

If $N = k2^n$ for some integer k, the remainder of the output of hash when dividing by N, given by the modulo operator, and adding 1 is uniformly distributed over $\{1, \ldots, N\}$ since each $j \in \{1, \ldots, N\}$ has an equal number of hashes assigned to it by hash. If $2^n \gg N$ and $N \neq k2^n$ for some integer k, then it is approximately uniformly distributed and converges to the uniform distribution as $n \to \infty$. □

The probability that no collisions occur for a particular bit string in Algorithm 1 is given by the following theorem.

Theorem 3.3. *The probability that a bit string* $b \in \{0,1\}^*$ *results in a perfect hash function is given by*

$$p(m, r) = N^{-m} P^N_m \tag{11}$$

where m *is the cardinality of the set being perfectly hashed,* r *is the load factor, and* $N = \frac{m}{r}$*.*

Proof. Suppose we have a set $\mathbb{A} \subset \{0,1\}^*$ of cardinality m. The set of perfect hash functions $\text{hash}^r_{\mathbb{A}} : \{0,1\}^* \mapsto \{0, 1, \ldots, N-1\}$ restricted to $\mathbb{A}$ where $N = \frac{m}{r}$ has a cardinality given by P^N_m since any choice of m out of N elements in the codomain and any ordering of the m elements in $\mathbb{A}$ to the chosen m elements in $\{0, 1, \ldots, N-1\}$ satisfy the definition of perfect hashing.

The set of hash functions $\{0,1\}^* \mapsto \{0, 1, \ldots, N-1\}$ restricted to $\mathbb{A}$ has a cardinality of N^m. Therefore, the ratio of perfect hash functions restricted to $\mathbb{A}$ to the total hash functions restricted to $\mathbb{A}$ is just

$$p = \frac{P^N_m}{N^m}. \tag{a}$$

By the property of the hash function hash^* being a random oracle, the algorithm randomly samples one of the hash functions, which is a perfect hash function with probabilty p.

□

Equation (11) may be reparameterized with respect to the load factor. By (2), $r = m/N$. Solving this equation with respect to N yields the solution

$$N = \frac{m}{r}, \tag{12}$$

and thus plugging in this value of N yields the result

$$p(m,r) = \frac{\left(\frac{m}{r}\right)!}{\left(\frac{m}{r}\right)^m \left(\frac{m}{r} - m\right)!}. \tag{13}$$

The expected *in-place* coding size is given by the following theorem.

Theorem 3.4. *The expected coding size is given approximately by*

$$\log_2 \mathrm{e} - \left(\frac{1}{r} - 1\right) \log_2\left(\frac{1}{1-r}\right) \quad \textit{bits/element.} \tag{14}$$

Proof.

$$\frac{n}{r}(1-r)\log_2(1-r) - \log_2 n - (u-n)\log_2(1-n/u) \tag{a}$$

$$\left(\frac{1}{r} - 1\right)\log_2(1-r) - \frac{1}{n}\log_2 n - \frac{u-n}{n}\log_2\left(1 - \frac{n}{u}\right) \tag{b}$$

The space required for the perfect hash function found by Algorithm 1 is of the order of the length n of the bit string b_n in the returned tuple. Therefore, for space efficiency, the algorithm exhaustively searches for a bit string in the order of increasing size n.

We are interested in the first case when no collision occurs, which is a geometric distribution with probability of success $p(m,r)$ as given by the discrete random variable

$$Q \sim \text{Geometric}\big(p(m,r)\big), \tag{c}$$

where $p(m,r)$ is given by (13. The expected number of trials for the geometric distribution is given by

$$\mathbb{E}[Q] = \frac{1}{p(m,r)} = \frac{\left(\frac{m}{r}\right)^m \left(\frac{m}{r} - m\right)!}{\left(\frac{m}{r}\right)!}. \tag{d}$$

By Definition 3.1, the n-th trial uniquely maps to a bit string of length $m = \lfloor \log_2 n \rfloor$. Thus, the expected bit length is given approximately by

$$\mathbb{E}[\log_2 Q] = \log_2\left(\frac{\left(\frac{m}{r}\right)^m \left(\frac{m}{r} - m\right)!}{\left(\frac{m}{r}\right)!}\right) \tag{e}$$

$$= m\log_2\left(\frac{m}{r}\right) + \log_2\left(\frac{m}{r} - m\right)! - \log_2\left(\frac{m}{r}\right)! \text{ bits.} \tag{f}$$

By Stirling's approximation,

$$\log_2 n! \approx n\left(\log_2 n - \log_2 \mathrm{e}\right). \tag{g}$$

and so the expectation may be rewritten approximately as

$$\begin{aligned} \mathbb{E}[Q] \approx m\log_2\left(\frac{m}{r}\right) - \frac{m}{r}\left(\log_2\left(\frac{m}{r}\right) - \log_2 \mathrm{e}\right) + \\ \left(\frac{m}{r} - m\right)\left(\log_2\left(\frac{m}{r} - m\right) - \log_2 \mathrm{e}\right) \quad \text{bits.} \end{aligned} \tag{h}$$

Since we are interested in the expected *bits per element*, we divide the expectation by m and after further simplification arrive at

$$\begin{aligned} \log_2\left(\frac{m}{r}\right) - \frac{1}{r}\log_2\left(\frac{m}{r}\right) + \\ \left(\frac{1}{r} - 1\right)\log_2\left(\frac{m}{r} - m\right) + \log_2 \mathrm{e} \text{ bits/element.} \end{aligned} \tag{i}$$

After further simplification, we arrive at

$$\log_2 \mathrm{e} - \left(\frac{1}{r} - 1\right) \log_2 \left(\frac{1}{1-r}\right) \text{ bits/element.} \tag{j}$$

□

The entropy of Q is given by

$$\frac{-(1-p)}{p} \log_2(1-p) - \log_2(p) \tag{15}$$

$$\left(\frac{1}{p} - 1\right) \log_2 \left(\frac{1}{1-p}\right) + \log_2 \frac{1}{p} \tag{16}$$

Note that Algorithm 1 has an exponential time complexity with respect to the cardinality of the input set $\mathbb{S}$. As a result, Algorithm 1 is not a practical algorithm for any reasonably large m. However, it is intended to illustrate theoretical properties useful to data structures that implement *oblivious* sets and maps[4, 3] based on the perfect hash function. Simple and efficient algorithms exist [1, 2].

The theoretical lower-bound of a *minimal* perfect hash function is given by the following postulate.

Postulate 3.1. *The theoretical lower-bound for* minimal *perfect hash functions has an expected coding size given approximately by*

$$1.44 \textit{ bits/element.} \tag{17}$$

Theorem 3.5. *The cryptographic perfect hash generator given by Algorithm 1 results in a* minimal *perfect hash function that obtains the theoretical lower-bound of* 1.44 *bits/element by invoking* `make_perfect_hash(`$\cdot$`,` $r = 1$`)`.

Proof. By (14), letting $r \to 1^+$ for the *minimal* perfect hash results in an expected coding size given by

$$\log_2 \mathrm{e} - \lim_{a \to 0^-} a \log_2 \frac{1}{a} = 1.44 \text{ bits/element,} \tag{a}$$

which is the expected lower-bound given by Postulate 3.1. □

This is as expected, since Algorithm 1 finds the *smallest* perfect hash function that is a function of a random oracle for any load factor $0 < r \leq 1$ in which the distribution of the sets are uniformly distributed. The following corollary follows as a result.

Corollary 3.5.1. *The lower-bound for perfect hash functions with a load factor* $0 < r \leq 1$ *is given by (14.*

As r goes to 0, the expected bits per element goes to 0. t Note that the *probability mass function* of the random bit string found for b is known exactly. Thus, if the desire is to *serialize* the perfect hash function for transmission or storage, shorter bit strings may be assigned to more probable bit strings. This representation is not usable *in-place*, i.e., the serialization must be decoded, but it can reduce transmission or storage cost.

Algorithm 1: Cryptographic perfect hash function constructor (single-level)

Input: $\mathbb{X} \subseteq \{0,1\}^*$ is the set to be perfectly hashed, and $r \in (0,1]$ is the load factor.

Output: A perfect hash function of $\mathbb{X}$ with load factor r, represented as (b_n, N) where $N = \lceil |\mathbb{X}|/r \rceil$.

function `make_perfect_hash(`$\mathbb{X}$, r`)`
- $m \leftarrow |\mathbb{X}|$;
- $N \leftarrow \lceil m/r \rceil$;
- $k \leftarrow \lceil \log_2 N \rceil$;
- **for** $n \leftarrow 1$ **to** ∞ **do**
  - $\mathbb{Y} \leftarrow \emptyset$;
  - $\alpha \leftarrow$ **true**;
  - **for** $x \in \mathbb{X}$ **do**
    - $h \leftarrow \text{trunc}(\text{hash}(x' \# n'), k)' \mod N$;
    - **if** $h \in \mathbb{Y}$ **then**
      - $\alpha \leftarrow$ **false**;
      - **break**;
    - $\mathbb{Y} \leftarrow \mathbb{Y} \cup \{h\}$;
  - **if** α **then**
    - **return** (n', N);

Algorithm 2: Two-level perfect hash function constructor

Input: $\mathbb{X} \subseteq \mathbb{U}$ is the objective set, $r \in \{q \in \mathbb{Q} | q \in 2^{-j} \wedge j \in \mathbb{N}\}$ is the load factor, and $k \in \mathbb{N}$ is the number of entries in the intermediate hash level.

Output: A two-level perfect hash function of $\mathbb{X} \subseteq \mathbb{U}$ with a load factor r and an intermediate level of k indices, denoted by $\text{hash}^r_{\mathbb{X}} : \mathbb{U} \mapsto \{0, \ldots, N\}$ where $N = \frac{|\mathbb{X}|}{r}$.

function `make_perfect_hash(`$\mathbb{X}$, r, k`)`
- $N = \frac{|\mathbb{X}|}{r}$;
- $t = \lceil \log_2 k \rceil$;
- `//` $\mathbb{X}[1], \ldots, \mathbb{X}[k]$ `is a total partition of` $\mathbb{X}$ `and` $\mathbb{X}[j_1], \ldots, \mathbb{X}[j_k]$ `is in decreasing order of cardinality.`
- $\mathbb{X}[\ell] = \{x \in \mathbb{X} \mid \text{trunc}(\text{hash}(x' \# 0'), t)' \mod k = \ell\}$;
- $\mathbb{Y} \leftarrow \emptyset$;
- $\mathbb{A} \leftarrow \emptyset$;
- **for** $\ell \leftarrow 1$ **to** k **do**
  - **for** $n \leftarrow 1$ **to** ∞ **do**
    - $\beta(x) = \text{trunc}(\text{hash}(x' \# n'), N)$;
    - $\mathbb{Y}_\ell = \{\beta(x) | x \in \mathbb{X}[\ell]\}$;
    - **if** $|\mathbb{Y}_\ell| = |\mathbb{X}_\ell| \wedge \mathbb{Y}_\ell \cap \mathbb{Y} = \emptyset$ **then**
      - $\mathbb{Y} \leftarrow \mathbb{Y} \cup \mathbb{Y}_\ell$;
      - **break**;
  - $\mathbb{A} \leftarrow \mathbb{A} \cup \{(\ell, n)\}$;
- **return** $(\mathbb{A}, k, N)$

4 A practical two-level perfect hash function

4.1 Analysis

Suppose we have a set $\mathbb{X}$ of m elements with some total order $x_{(1)}, \ldots, x_{(m)}$ for which we seek a perfect hash function $\text{hash}_{\mathbb{X}} : \mathbb{U} \mapsto mathbbmZ$.

We denote the *serialization* of some value x by x'. If x is already understood to be a serialization, then x' denotes deserialization instead.

By the assumption that the hash function $\text{hash} : \mathbb{U} \mapsto \{0,1\}^k$ is a random oracle restricted to the codomain $\{0,1\}^k$, the n-th attempt (trial) $\text{hash}\left(x'_{(j)} \# n'\right)$ to find a non-colliding hash for $x_{(j)}$ has a probability of success $p_j = \frac{m-j+1}{m}$ since we have already found hashes for the previous $j-1$ elements and therefore there are only $m-(j-1)$ hashes remaining.

This is an example of the Coupon collector's problem. Let T_j denote the uncertain number of trials needed to find a non-colliding hash for $x_{(j)}$. The total number of trials needed is an uncertain random variable given by

$$T = \sum_{j=1}^{m} T_j. \tag{18}$$

By the linearity of the expectation operator,

$$E(T) = \sum_{j=1}^{m} E(T_j) = \sum_{j=1}^{m} \frac{1}{p_j} = mH_{m-1}, \tag{19}$$

which asymptotically converges to $E(T) = m\gamma + m\ln(m-1)$ or on average $\gamma + \ln(m-1)$ trials per element of $\mathbb{X}$.

If we use the minimum number of bits

The variance of T is given by

$$VT = \sum_{j=2}^{m} VT_j. \tag{20}$$

5 Algebra of function composition

Perfect hash functions can be composed with other functions to create new perfect hash functions with different properties. This algebraic structure provides flexibility in designing hash function families.

5.1 Composition preserves injectivity

While a perfect hash function $h_{\mathbb{A}} : \mathbb{X} \mapsto \mathbb{Y}$ is not globally injective, its restriction to $\mathbb{A}$ is injective:

$$h_{\mathbb{A}}|_{\mathbb{A}} : \mathbb{A} \mapsto \mathbb{Y}. \tag{21}$$

This restriction property enables composition with injective functions while preserving the perfect hash property.

Theorem 5.1 (Post-composition with injection)**.** *Let $g : \mathbb{Y} \mapsto \mathbb{Z}$ be an injective function, $h^r_{\mathbb{A}} : \mathbb{X} \mapsto \mathbb{Y}$ be a perfect hash function, and $|\mathbb{Z}| = (1+\alpha)|\mathbb{Y}|$ for $\alpha \geq 0$. The composition $g \circ h^r_{\mathbb{A}} : \mathbb{X} \mapsto \mathbb{Z}$ is a perfect hash function $h^{r'}_{\mathbb{A}}$ where $r' = \frac{r}{1+\alpha}$.*

Proof. From the load factor definition, $r = \frac{|\mathbb{A}|}{|\mathbb{Y}|}$, so $|\mathbb{Y}| = \frac{|\mathbb{A}|}{r}$. Since g is injective, $g \circ h_{\mathbb{A}}^r$ is injective on $\mathbb{A}$, making it a perfect hash function with load factor

$$r' = \frac{|\mathbb{A}|}{|\mathbb{Z}|} = \frac{|\mathbb{A}|}{(1+\alpha)|\mathbb{Y}|} = \frac{|\mathbb{A}|}{(1+\alpha)\frac{|\mathbb{A}|}{r}} = \frac{r}{1+\alpha}. \tag{a}$$

□

5.2 Permutation equivalence classes

Given a perfect hash function $h_{\mathbb{A}} : \mathbb{X} \mapsto \mathbb{Y}$, composing with any permutation $\pi : \mathbb{Y} \mapsto \mathbb{Y}$ yields another perfect hash function $\pi \circ h_{\mathbb{A}}$ over $\mathbb{A}$ with the same load factor. Since there are $|\mathbb{Y}|!$ permutations of $\mathbb{Y}$, a single perfect hash function generates a family of $|\mathbb{Y}|!$ related perfect hash functions.

These functions form an *equivalence class* where all members have identical collision structure outside of $\mathbb{A}$. This equivalence class can be denoted $[\pi \circ h_{\mathbb{A}}]$ where π ranges over all permutations of $\mathbb{Y}$.

Corollary 5.1.1. *For a perfect hash function with codomain $\mathbb{Y}$, the ratio of permutation-equivalent perfect hash functions to all possible functions is*

$$\frac{|\mathbb{Y}|!}{|\mathbb{Y}|^{|\mathbb{X}|}}. \tag{22}$$

Appendices

A Probability mass of random bit length

Algorithm 3: Bit length sampler of the cryptographic perfect hash function

Input: The number m is the cardinality of the perfect hash set and the number r is the load factor.

Output: The *minimum bit length* n conditioned on a random set of cardinality m and a load factor r.

```
function sample_bit_length(m, r)
    for i ← 1 to m do
        x ← randomly draw a bit string from {0,1}* without replacement;
        𝕊 ← 𝕊 ∪ {x};
    (b_n, N) ← make_perfect_hash(𝕊, r);
    // The integer N must also be coded, which we assume has a constant bit
       length.
    return n + O(1);
```

The bit length of the perfect hash function found by Algorithm 1 has an uncertain value with respect to (random) sets and therefore we may model it as a random variable as given by the following definition.

Definition A.1. *The perfect hash function generated by Algorithm 1 has a random bit length given by*

$$N = \texttt{sample\_bit\_length}(m, r) \tag{23}$$

where m is the cardinality of the random set and r is the load factor.

Note that if the distribution of sets is not uniformly distributed as given by `sample_bit_length`, then better algorithms than Algorithm 1 are possible, i.e., the lower-bounds are with respect to *uniformly distributed* random sets and smaller lower-bounds are possible if the random sets are non-uniformly distributed.

The probability mass of the random bit length N is given by the following theorem.

Theorem A.1. *The random bit length N has a probability mass function given by*

$$p_N(n|m, r) = q^{2^n-1}\left(1 - q^{2^n}\right). \tag{24}$$

where m is the cardinality of the random set, r is the load factor, and $q = 1 - p(m, r)$,

Proof. Each iteration of the loop in Algorithm 1 has a collision test which is Bernoulli distributed with a probability of success $p(m, r)$, where success denotes no collision occurred. We are interested in the random length N of the bit string when this outcome occurs.

For the random variable N to realize a value n, every bit string smaller than length n must fail and a bit string of length n must succeed. There are $2^n - 1$ bit strings smaller than length n and each one fails with probability q, and so by the product rule the probability that they all fail is given by

$$q^{2^n-1}. \tag{a}$$

Given that every bit string smaller than length n fails, what is the probability that every bit string of length n fails? There are 2^n bit strings of length n, each of which fails with probability q as before and thus by the product rule the probability that they all fail is q^{2^n}, whose complement, the probability that not all bit strings of length n fail, is given by

$$1 - q^{2^n}. \tag{b}$$

By the product rule, the probability that every bit string smaller than length n fails and a bit string of length n succeeds is given by the product of (a) and (b),

$$q^{2^n-1}\left(1 - q^{2^n}\right). \tag{c}$$

For (c to be a probability mass function, two conditions must be met. First, its range must be a subset of $[0, 1]$. Second, the summation over its domain must be 1.

The first case is trivially shown by the observation that q is a positive number between 0 and 1 and therefore any non-negative power of q is positive number between 0 and 1.

The second case is shown by calculating the infinite series

$$S = \sum_{n=0}^{\infty} q^{2^n-1}\left(1 - q^{2^n}\right) \tag{d}$$

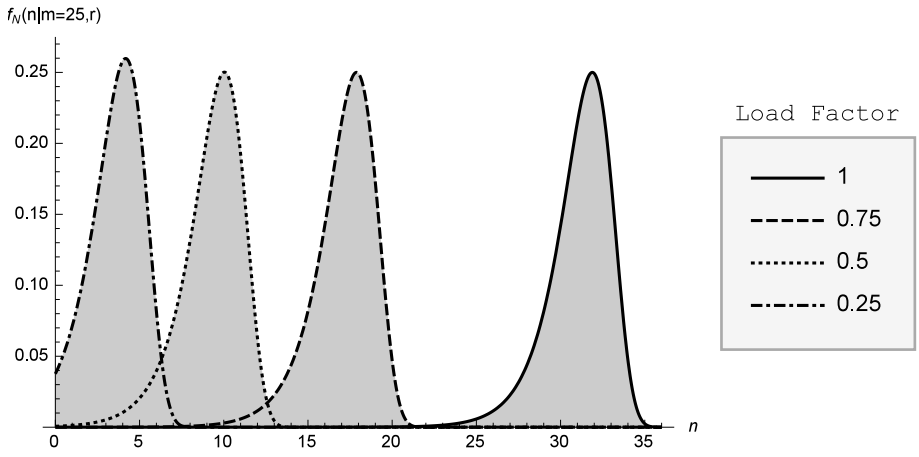
$$= \sum_{n=0}^{\infty} q^{2^n-1} - q^{2^{n+1}-1}. \tag{e}$$

Explicitly evaluating this series for the first 4 terms reveals a telescoping sum given by

$$S = (1 - q) + (q - q^3) + (q^3 - q^7) + (q^7 - q^{15}) + \cdots, \tag{f}$$

where everything cancels except 1. □

Figure 2: Probability mass function of
random bit length N



In Figure 2, we plot the probability mass function of N conditioned on $m = 25$ and several different load factors. We see that the probability mass is peaked and unimodal and tends to nearly zero everywhere except over a concentrated interval around its expected value.

The expected size was already computed, but the probability mass function contains everything there is to know about the distribution of N, not just the *expected* value. However, for illustration, we show how the expected value may be computed.

Theorem A.2. *The expected bit length of N conditioned on random sets of cardinality m and a load factor r is given by*

$$\mathbb{E}[N] = \sum_{j=1}^{\infty} q^{2^j - 1}, \tag{25}$$

where $q = 1 - p(m, r)$.

Proof. The expectation of N is given by

$$\sum_{j=0}^{\infty} j q^{2^j - 1} \left(1 - q^{2^j}\right) \tag{a}$$

$$= \sum_{j=0}^{\infty} j \left(q^{2^j - 1} - q^{2^{j+1} - 1}\right). \tag{b}$$

Explicitly evaluating this series for the first 4 terms reveals a converging sum given by

$$0 + (q - q^3) + 2(q^3 - q^7) + 3(q^7 - q^{15}) + \cdots \tag{c}$$

$$= q + q^3 + q^7 + q^{15} + \cdots. \tag{d}$$

□

When we numerically evaluate (25 for various m and r, the results are in agreement with (14.

References

[1] Djamal Belazzougui, Fabiano C Botelho, and Martin Dietzfelbinger. Compress, hash, and displace algorithm. *ACM Journal of Experimental Algorithmics (JEA)*, 14:1–26, 2009.

[2] Fabiano C Botelho, Rasmus Pagh, and Nivio Ziviani. Simple and space-efficient minimal perfect hash functions. In *International Workshop on Algorithms and Data Structures*, pages 139–150. Springer, 2007.

[3] Zbigniew J Czech, George Havas, and Bohdan S Majewski. A family of perfect hashing methods. *The Computer Journal*, 35(6):547–554, 1992.

[4] Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert E Tarjan. Space-efficient hash tables with worst-case constant access time. In *STACS 90: 7th Annual Symposium on Theoretical Aspects of Computer Science Rouen, France, February 22–24, 1990 Proceedings 7*, pages 271–282. Springer, 1990.