Theory and Practice of Monotone Minimal Perfect Hashing

DJAMAL BELAZZOUGUI
Université Paris Diderot–Paris 7, France

PAOLO BOLDI
Università degli Studi di Milano, Italy

RASMUS PAGH
IT University of Copenhagen, Denmark

SEBASTIANO VIGNA
Università degli Studi di Milano, Italy

Abstract

Minimal perfect hash functions have been shown to be useful to compress data in several data management tasks. In particular, *order-preserving* minimal perfect hash functions [12] have been used to retrieve the position of a key in a given list of keys: however, the ability to preserve any given order leads to an unavoidable $\Omega(n \log n)$ lower bound on the number of bits required to store the function. Recently, it was observed [1] that very frequently the keys to be hashed are sorted in their intrinsic (i.e., lexicographical) order. This is typically the case of dictionaries of search engines, list of URLs of web graphs, etc. We refer to this restricted version of the problem as *monotone minimal perfect hashing*. We analyse experimentally the data structures proposed in [1], and along our way we propose some new methods that, albeit asymptotically equivalent or worse, perform very well in practice, and provide a balance between access speed, ease of construction, and space usage.

1 Introduction

A minimal perfect hash function maps bijectively a set S of n keys into the set $\{0, 1, \ldots, n-1\}$. The construction of such functions has been widely studied in the last years, leading to fundamental theoretical results such as [13, 14, 20].

From an application-oriented viewpoint, *order-preserving* minimal perfect hash functions have been used to retrieve the position of a key in a given list of keys [12, 27]. In [1] we note that all existing techniques for this task assume that keys can be provided in any order, incurring an unavoidable $\Omega(n \log n)$ lower bound on the number of bits required to store the function. However, very frequently the keys to be hashed are sorted in their intrinsic (i.e., lexicographical) order. This is typically the case of dictionaries of search engines, list of URLs of web graphs, etc. Thus, it is interesting to study *monotone minimal perfect hashing*—the problem of mapping each key of a lexicographically sorted set to its ordinal position.

In this paper our main goal is that of minimising the function description while maintaining quick (ideally, constant-time) access. In [1] we presented two solutions for the case where elements of S are taken from a universe of u elements. The first solution (based on longest common prefixes) provides $O((\log u)/w)$ access time, where w is the size of a machine word (so it is constant time if the string length is linear in the machine-word size), but requires $O(\log \log u)$ bits per element. The second solution (based on a *z-fast trie*) requires just $O(\log\log\log u)$ bits per element, but access requires $O((\log u)/w + \log\log u)$ steps.

In this paper, we present some new structures for this problem, and compare them experimentally with the ones mentioned above. The purpose is twofold: first of all, we want to understand the constants hidden in the asymptotic estimates of [1]; second, we want to establish whether in a real-world scenario the new solutions proposed here have some practical advantage over the theoretically better ones.

To this purpose, we provide precise, big-Oh-free estimates of the number of bits occupied by each structure, which turn out to match very closely the number of bits required in the actual implementations; such estimates are valuable in two ways: they make it possible to tune optimally the parameters (something that would happen modulo a multiplicative constant using big-Oh estimates); and they allow one to predict in advance the space needed by each data structure.

To do so, we must choose explicit implementations for a number of components (static functions, rank/select structures, succinct dictionaries, etc.). We have used in most cases classical structures, which are slightly less space efficient than more recent ones, but are in practice much faster.

We implement and engineer in detail all solutions in Java, and run them against large and real data. The choice of Java requires of course some elaboration. Recent benchmarks show that Java's performance is by now very close to that of C/C++, and in some cases even better (in particular when dynamic inlining of very short methods decreases the cost of method calls). Moreover, we are interested in comparing the speed of different data structures, and in this case the language choice is somewhat irrelevant.

The code we use is real-world code. By this we mean that we do not test highly specialised classes designed for the benchmarks of this paper. Our classes are publicly available and used in research and production systems. To be general, all our data structures are built around the idea that we should be able to hash *any* kind of object: thus, every constructor of a data structure accepts, besides a list of objects, a *strategy object* [15] providing a method that turns the given objects into bit strings. This makes it possible to use the same classes to hash UTF-8 strings, ISO-8859-1 strings, binary bit strings, or any kind of object whose intrinsic ordering can be reflected into the lexicographical ordering of a bit-string representation. This approach slows down, of course, construction and query timings with respect to hardwired classes that assume, for instance, to hash just ASCII strings. Nonetheless, we believe that studying the performance of realistic, well-engineered data structures is much more interesting (and here Java has a clear advantage over C/C++).

In Section 2 we define precisely our problem, and in Section 3 we set up the tools that will be used in the rest of the paper. Then, in Sections 4, 5, 6, 7 and 8 we present data structures that provide different tradeoffs between space and time. Throughout the paper, we use the example of Figure 1 to illustrate the algorithms. Finally, in Section 10 we present experiments based on Java implementations of our data structures.

The code used for our experiments is distributed as part of the Sux4J project[1] under GNU Lesser General Public License. The lists of URLs used in the experimental section are available as part of the data sets distributed by the Laboratory for Web Algorithmics (`http://law.dsi.unimi.it/`), so as to make our experiments fully reproducible.

2 Definitions and notation

Sets and integers. We use von Neumann's definition and notation for natural numbers: $n = \{0, 1, \dots, n-1\}$. We thus freely write $f : m \to n$ for a function from the first m natural numbers to the first n natural numbers. We do the same with real numbers, with a slight abuse of notation, understanding a ceiling operator.

[1]`http://sux4j.dsi.unimi.it/`.

s_0	0001001000000	s_6	0010011010100
s_1	0010010101100	s_7	0010011010101
$\boldsymbol{s_2}$	**0010010101110**	$\boldsymbol{s_8}$	**0010011010110**
s_3	0010011000000	s_9	0010011110110
s_4	0010011001000	s_{10}	0100100010000
$\boldsymbol{s_5}$	**0010011010010**		

Figure 1: A toy example: $S = \{s_0, \dots, s_{10}\}$ is divided into three buckets of size three (except for the last one that contains just two elements), whose delimiters $D = \{s_2, s_5, s_8\}$ appear in boldface.

In the following, we will always assume that a universe u of integers, called *keys*, is fixed; this set may in many applications be infinite, but unless otherwise specified we will suppose that it is finite. The set u has a *natural order* which corresponds to the string lexicographical order of the $\log u$-bit left-zero-padded binary representation. We assume, for sake of simplicity, that all strings have the same length $\log u$. At the end of the paper, in Section 9, we describe the few modifications that are needed to state our results in terms of the *average* string length of a prefix-free set of variable-length strings.

Given $S \subseteq u$ with $|S| = n$, and given m, an *m-bucket hash function for S* is any function $h : S \to m$. We say that:

- h is *perfect* iff it is injective;
- h is *minimal perfect* iff it is injective and $n = m$;
- h is *monotone* iff $x \leq y$ implies $h(x) \leq h(y)$ for all $x, y \in S$;
- h is *order-preserving* with respect to some total order $\preceq$ on U iff $x \preceq y$ implies $h(x) \leq h(y)$ for all $x, y \in S$.

We would like to stress the distinction between *monotone* and *order-preserving* functions, which we introduce because the structures proposed in the literature as "order-preserving" [12] actually make it possible to *impose* any order on the keys. On the contrary, we are interested in the *existing*, standard lexicographical order on keys viewed as strings. The distinction is not moot because the lower bound $\Omega(n \log n)$ for order-preserving hashing does not hold for monotone hashing.

Notice that since monotone hash functions are a special case of order-preserving hash functions (applied to the natural order), any structure for the latter can be used to implement the former, but not vice versa.

Tries. Let S be a prefix-free set of binary sequences. The *compacted trie* on S is a binary tree whose nodes contain binary sequences (the sequence contained in a node is called the *compacted path* at that node). It is defined recursively as follows:

- if $S = \varnothing$, it is the empty binary tree;
- otherwise, let p be the longest common prefix of the elements of S, and let $S_b = \{x \mid pbx \in S\}$ where $b \in \{0, 1\}$; then, the trie for S is made by a root with compacted path p whose left (right) subtree is the trie for S_0 (S_1, respectively).

The leaves of the trie are in one-to-one correspondence with the elements of S, with the k-th leaf from the left corresponding to the k-th element of S in lexicographical order.

Approximations. In this paper we purposely avoid asymptotic notation; our interest is in providing fairly precise estimates of the number of bits used by each structure. Nonetheless, we must allow some approximation if we want to control the size of our expressions. We will tacitly assume the following:

$$\log(\varepsilon + \log n) \approx \log\log n \quad \text{for small } \varepsilon$$
$$\log n - \log\log n \approx \log n \quad \text{when appearing as a subexpression.}$$

Moreover, $o(n)$ components will be tacitly omitted.

Model. We consider a standard RAM with word size w. We do not assume, as it often happens in theoretical literature, that the universe u satisfies $w \leq c \log u$: this assumption yields unrealistic results when manipulating long strings.

Hashing prefixes. Some care must be taken to get the best possible query time dependency on the key length $\log u$. The problem is that hashing a key of $\log u$ bits takes time $O((\log u)/w)$, and in many algorithms we need to hash several prefixes of a string. Thus, we precompute the internal state of the hashing algorithm for each word-aligned prefix of the key. This can be done in time $O((\log u)/w)$. Using the precomputed table, subsequent hash evaluations on any prefix can be done in constant time.

3 Tools

The data structures described in this paper are based on a combination of techniques from two different threads of research: minimal perfect hashing based on random hypergraphs, and succinct data structures.

3.1 Rank and select.

We will make extensive use of the two basic blocks of several succinct data structures—rank and select. Given a bit array (or bit string) $\boldsymbol{b} \in \{0,1\}^n$, whose positions are numbered starting from 0, $\text{rank}_{\boldsymbol{b}}(p)$ is the number of ones up to position p, exclusive ($0 \leq p \leq n$), whereas $\text{select}_{\boldsymbol{b}}(r)$ is the position of the r-th one in $\boldsymbol{b}$, with bits numbered starting from 0 ($0 \leq r < \text{rank}_{\boldsymbol{b}}(n)$). These operations can be performed in constant time on a string of n bits using additional $o(n)$ bits [23, 8]. When $\boldsymbol{b}$ is obvious from the context we shall omit the subscript.

3.2 Storing functions.

In the rest of the paper we will frequently need to associate values to the key set S, that is, to store a function $f : S \to 2^r$ for some constant r. An obvious possibility is to store a minimal perfect hash function on S and use the resulting value to index a table of rn bits. Much better theoretical solutions were made available recently [6, 9]: essentially, it is possible to evaluate a function in constant time storing just $rn + o(n)$ bits. Since we are interested in practical applications, however, we will use an extension of a technique developed by Majewski, Wormald, Havas and Czech [27] that has a slightly larger space usage, but has the advantage of being extremely fast, as it requires just the evaluation of three hash functions[2] plus three accesses to memory.

The technique developed in [27] was used to compute order-preserving hash functions in γrn bits, where $\gamma = 1.23$. Actually, the very same approach allows one to assign *any value* to the keys—emitting a distinct value in n for each element of S is just one of the possibilities. Thus, we will extend (and improve) the technique to store arbitrary functions in just $\gamma n + rn$ bits.

[2]Actually, in our implementations we use Jenkins hashing [24], which provides three 64-bit hash values with a single evaluation.

We recall briefly the technique of [27]. We start by building a random 3-hypergraph with γn nodes and n hyperedges—one per element of S—defined by three random[3] hash functions $h_1, h_2, h_3 : S \to \gamma n$. The choice of $\gamma = 1.23$ makes the probability that the resulting graph is acyclic positive.[4]

The acyclicity check computes a (sort-of) topological order of the hyperedges with the property that by examining the hyperedges in that order, at least one vertex of each hyperedge, the *hinge*, will have never appeared previously. We now assign values a_i to vertices, with the aim of obtaining that

$$f(x) = (a_{h_1(x)} + a_{h_2(x)} + a_{h_3(x)}) \bmod 2^r.$$

This is always possible, because by examining the vertices in the order produced by the acyclicity check we can always choose the value for the hinge (if there are more unassigned values, we set them to zero).

Storing the function in this way would require $\gamma r n$ bits. We call such a structure an *MWHC function* (from Majewski, Wormald, Havas and Czech). We note, however, that when r is large we can use an additional bit array $\boldsymbol{s}$ to mark those vertices that have a non-zero value, and record in an array b only the (at most n) nonzero values. To compute a_i, we first look at s_i: if it is zero, $v_i = 0$; otherwise, we compute $\mathrm{rank}_{\boldsymbol{s}}(i)$ and use the resulting value to index the array b.

The resulting structure, which we call a *compacted MWHC function*, uses $\gamma n + rn$ bits: this is advantageous as long as $\gamma + r < \gamma r$, which happens when $r > 5$.[5]

Three remarks are in order:

- even the best imaginable solution obtained by coupling a minimal perfect hash function (requiring at least $n \log e \approx 1.44 n$ bits [13]) and an array of rn bits is never advantageous;
- for an order-preserving hash function, $\log(n!) = n \log n - O(n)$ bits is an obvious lower bound (as we can store the keys following any order), so a compacted MWHC function provides an optimal solution: thus, we will not discuss order-preserving functions further.

Another, complementary approach to the storage of static functions uses a minimal perfect hash function to index a *compressed* bit array (see the next section for some examples of suitable techniques). To obtain a minimal perfect hash function, we can adopt again the above hypergraph technique and use two bits per vertex to code the *index of the hash function outputting the hinge*. This effectively provides a perfect hash function $S \to \gamma n$ into the vertex space (by mapping each key to its hinge, a value in $\{1, 2, 3\}$). Thus, the perfect hash function can be turned into a *minimal* perfect hash function by ranking, as it is immediate to devise a space $o(n)$, constant-time ranking structure that counts nonzero pairs of bits, so we obtain minimal perfect hashing in $2\gamma n$ bits (the idea of coding the hinge position appeared for the first time in Bloomier filters [7]; using ranking to obtain a minimal perfect hash function was suggested in [4]). This approach is advantageous if the bit array can be compressed with a ratio better than $1 - \gamma/r$.

Two-step MWHC functions. To gain a few more bits when the distribution of output values is significantly skewed, we propose *two-step MWHC functions*. We fix an $s < r$: then, the $2^s - 1$ most frequent output values are stored in a (possibly compacted) s-bit MWHC function, whereas the value $2^s - 1$ is used as an escape to access a secondary MWHC function storing all remaining values. Of course, we need to store explicitly the mapping from $2^s - 1$ to the set of most frequent values. The value s is chosen so to optimise the space usage (either by brute-force search or by differentiation if the output-value distribution is known). In the frequent cases when the rank of the output values fit a geometric distribution with parameter p, the store space required is

$$\left(s + \gamma + (2^s - 1)\frac{r}{n} + (r + \gamma)(1 - p)^{2^s+1}\right) n$$

[3]In this paper we make the *full randomness* assumption—our hash functions are fully random. Albeit controversial, this is a common practical assumption that makes it possible to use the results about random hypergraphs.

[4]This value was just determined experimentally in [27], but was subsequently proved to be correct [28].

[5]This evaluation does not take into account that ranking structures are asymptotically $o(n)$, but on real data they occupy a significant fraction of the original data size. The actual threshold depends on that fraction.

bits: $(s + \gamma)n$ bits for the frequent-output MWHC function, $(2^s - 1)r$ bits for the array of most frequent values, and $(r + \gamma)$ bits for each of the $(1 - p)^{2^s+1}n$ inputs with less frequent output. Ignoring the array of frequent values, the store space above is minimised by

$$s = \log W\left(\frac{1}{(r+\gamma)(p-1)\ln 2}\right) - \log\ln(1-p),$$

where $W(-)$ is Lambert's W function, which in this case can be approximated by $W(x) \approx -\ln(-1/x) - \ln\ln(-1/x)$.

A large-scale approach. MWHC functions and minimal perfect hashes require a large amount of memory to be built, as they require random access to the 3-hypergraph to perform a visit. To make their construction suitable for large-size key sets we reuse some techniques from [5]: we divide keys into *chunks* using a hash function, and build a separate MWHC function for each chunk. We must now store for each chunk the offset in the array a where the data relative to the chunk is written, but using a chunk size $\omega(\log n)$ (say, $\log n \log\log n$) the space is negligible. The careful analysis in [5] shows that this approach can be made to work even at a theoretical level by carefully reusing the random bits when building the MWHC functions of each chunk.

3.3 Elias–Fano representation of monotone functions.

We will frequently need to store either a list of arbitrary integers, or a list of nondecreasing natural numbers. In both cases, we would like to consume as little space as possible. To this purpose, we will use the *Elias–Fano representation of monotone functions* [10, 11]. Such a data structure stores a monotone function $f : n \to 2^s$, that is, a list of nondecreasing natural numbers $0 \leq x_0 \leq x_1 \leq \cdots \leq x_{n-1} < 2^s$, provides constant-time access[6] to each x_i, and uses $2 + s - \log n$ bits per element when $n < 2^s$, and $1 + 2^s/n$ bits when $2^s \leq n$.

Here, for completeness we briefly recall from [10] the representation: we store explicitly the $s - \log n$ lower bits of each x_i in a bit array. The value v of the upper $\log n$ bits of x_i are written in a bit array b of $2n$ bits by setting the bit of position $i + v$. It is easy to see that now v can be recovered as $\mathrm{select}_b(i) - i$. Since the lower bits use $s - \log n$ bits per element, and the bit array requires $2n$ bits, we obtain the stated space bounds (in the case $2^s \leq n$, the first array is empty, and the second array requires just $n + 2^s$ bits). Selection in b can be performed with one of the many selection structures in the literature (see, e.g., [8, 17, 18, 25, 31]; we note that if $n \ll 2^s$, the impact of the selection structure on the space usage is irrelevant).

Finally, if we have to store a list of natural numbers $x_0, x_1, \ldots, x_{n-1}$, we can just juxtapose the binary representation of $x_0 + 1, x_1 + 1, \ldots, x_{n-1} + 1$ (without the most significant bit) and use an Elias–Fano monotone sequence to store the starting point of each binary representation. The resulting structure provides significant gains if the distribution is skewed towards small values, as the overall space usage is $2n + n\log(\sum_i \lfloor\log(x_i + 1)\rfloor/n) + \sum_i \lfloor\log(x_i + 1)\rfloor$. We will speak in this case of an *Elias–Fano compressed list*. Note that if a is average of $\lfloor\log(x_i + 1)\rfloor$ (i.e., the average length in bits of the x_i increased by one), then the space usage is bounded by $(2 + a + \log a)n$.

3.4 Bucketing.

We now discuss here briefly a general approach to minimal perfect monotone hashes that we will use in this paper and that will be referred to as *bucketing*. The same idea has been widely used for non-monotone perfect hashing, and its extension to the monotone case proves to be fruitful.

Suppose you want to build a minimal perfect monotone hash function for a set S; you start with:

[6] Actually, in the original Elias' paper access is not constant, as it relies on a selection structure that is not constant-time. Replacing the selection structure with a modern, constant-time structure provides constant-time access to the x_is; note, however, that our implementations use the original Elias' inventory-based linear scan, as it turns out to be faster.

- a monotone hash function $f : S \to m$ (called the *distributor*) mapping S to a space of m *buckets*;
- for each $i \in m$, a minimal perfect monotone hash function g_i on $f^{-1}(i)$;
- a function $\ell : m \to n$ such that, for each $i \in m$,

$$\ell(i) = \sum_{j<i} |f^{-1}(j)|.$$

Then, the function $h : S \to n$ defined by

$$h(x) = \ell(f(x)) + g_{f(x)}(x)$$

is a minimal perfect monotone hash function for S. The idea behind bucketing is that the distributor will consume little space (as we do not require minimality or perfection), and that the functions hashing each element in its bucket will consume little space if the bucket size is small enough. Note also that ℓ does not need to be stored if all buckets have the same size, except possibly for the last one (i.e., if $|f^{-1}(i)| = |f^{-1}(0)|$ for all $i < m - 1$).

4 Bucketing with longest common prefixes

The first solution we propose is taken from [1] and it is based on *longest common prefixes*. This solution has $(\log u)/w$ access time (thus, constant time if $w \leq c \log u$), since it requires just the evaluation of a fixed number of hash functions; on the negative side, it has in practice the highest memory usage among the algorithms we discuss.

Let b be a positive integer, and divide the set S into buckets B_i of size b preserving order. We now apply bucketing as described in Section 3.4, but to store the function $f : S \to m$ (with $m = \lceil n/b \rceil$), we proceed as follows:

- we store a compacted MWHC function $f_0 : S \to \log(u/b)$ that assigns, to each $x \in S$, the length $\ell_{f(x)}$ of the longest common prefix of the bucket containing x (note that the length of the common prefix of a set of b strings of length $\log u$ cannot exceed $\log u - \log b$);
- we store a compacted (or a two-step) MWHC function $f_1 : \{ p_0, p_1, \dots, p_{m-1} \} \to m$ mapping p_i to i, where p_i is the longest common prefix of B_i.

To compute $f(x)$ for a given $x \in S$, one first applies f_0 obtaining the length $\ell_{f(x)}$; from this one can compute $p_{f(x)}$, whence, using f_1, one obtains $f(x)$. Figure 2 displays the functions f_0 and f_1 for the example of Figure 1.

The function f_0 requires $(\gamma + \log\log(u/b))n$ bits, whereas f_1 requires $(\gamma + \log(n/b))n/b$ bits [7]; the g_is would require $(\gamma + \log b)n$ bits, but we can pack the information returned by f_0 and the g_is into a single function. Altogether, we obtain:

$$\left(\gamma + \log\log\frac{u}{b} + \log b\right)n + \left(\gamma + \log\frac{n}{b}\right)\frac{n}{b}.$$

We note, however, that using a two-step MWHC function for f_0 could significantly decrease the term $\log\log\frac{u}{b}$, at the expense of additional γn bits (as f_0 and the g_is would not be grouped together). We will see in the experimental section that the two-step function provides the best results in terms of space.

[7] For the sake of simplicity, we are here assuming that f_1 is also stored as a compacted MWHC function, not as a two-step one.

s_0	2
s_1	2
s_2	2
s_3	8
s_4	8
s_5	8
s_6	11
s_7	11
s_8	11
s_9	1
s_{10}	1

(a)

00	0
00100110	1
00100110101	2
0	3

(b)

Figure 2: Bucketing with longest common prefix for the set S of Figure 1: (a) f_0 maps each element x of S to the length of the longest common prefix of the bucket to which x belongs; (b) f_1 maps each longest common prefix to the bucket index.

Approximating $\log\log(u/b)$ with $\log\log u$, the above function is minimised by $b = W(\xi n)$ where W is Lambert's W function and $\xi = e2^{\gamma} \approx 6.4$, so (using the fact that $W(x) \approx \ln x - \ln\ln x$ for large positive x)

$$b \approx \ln(\xi n) - \ln\ln(\xi n) \approx 1 + \gamma \ln 2 + \ln n - \ln\ln n,$$

giving about

$$\left(\gamma + \log e - \log\log e + \log\log n + \log\log \frac{u}{\log n}\right) n$$

bits. A good upper bound on the space usage is $(2.14 + \log\log n + \log\log u)n$ bits.

4.1 Variable-size slicing for large n.

In case n is close to u, we can still employ longest prefixes, but we can modify our structure so as to use $O(n \log\log(u/n))$ bits. To do so, we divide the elements of S into *slices* on the basis of their first $\log n$ bits. Clearly, the size of each slice will now be variable, but the position of the first element of each slice is a monotone function that can be recorded in $2n$ bits using the Elias–Fano representation. This construction provides a monotone hash function of each key into its slice.

We are now left to define the minimal perfect monotone hashing inside each slice. Let $b(x) = 1+\gamma \ln 2+\ln x-\ln\ln x$ be the bucket size used in the previous section. If a slice has size smaller than bu/n, we simply use a MWHC function (which requires at most $\gamma + \log(bu/n)$ bits per element). Otherwise, we exploit the fact that each slice can be seen as a subset of a smaller universe of size u/n by stripping the first $\log n$ bits of each string. Thus, the bound given in the previous section, when applied to a slice of size s, becomes

$$\left(2\gamma + \log e - \log\log e + \log\log s + \log\log \frac{u}{n \log s}\right) s,$$

where the additional γs bits come from the need of separating the storage for the f_0 and the g_is: in this way, both the $(\log(bu/n))$-sized data for large and small slices can be stored in the same bit array with no additional costs.

We conclude that the entire data structure requires at most

$$\left(2\gamma + \log e - \log\log e + 2 + \max\left\{\log\log n, \log\log \frac{u}{n}\right\} + \log\log \frac{u}{n}\right) n$$

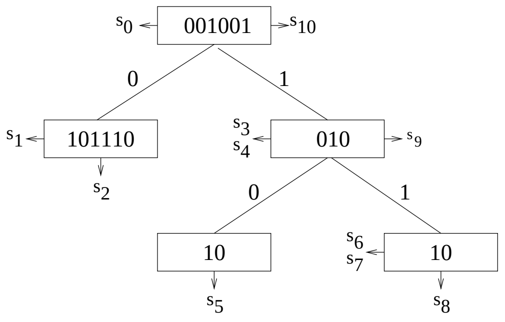


Figure 3: The standard compacted trie built from the set D of Figure 1. This data structure can be used to rank arbitrary elements of the universe with respect to D: when the trie is visited with an element not in D, the visit may terminate at some arbitrary node, determining that the given element is to the left (i.e., smaller than) or to the right (i.e., larger than) all the leaves that descend from that node. The picture shows, for each element of S, the node where the visit would end.

bits. Note that the latter formula only becomes advantageous when n is very large ($n \approx \sqrt{u}$ or larger).

5 Bucketing with partial compacted tries

We now turn to a data structure requiring access time linear in the length of the key (e.g., $O(\log u)$). As in the previous section, the idea is always that of dividing S into equally sized buckets, and then compute the bucket offset using a compacted MWHC function. However, this time for each bucket B_i we consider its *delimiter* k_i, which is the last string in B_i (with respect to the natural $\leq$-order). Let D denote the set of delimiters of all buckets, except for the last one: we will locate the bucket of a key $x \in S$ by counting the number of elements of D that are smaller than x. This is actually a standard operation, called rank:

$$\operatorname{rank}_T(x) = |\{t \in T \mid t < x\}| \text{ for } T \subseteq u \text{ and } x \in u.$$

There are many data structures in the literature that can be used to compute ranks. An obvious, naive possibility is using a compacted trie [26] containing the strings in D (see Figure 3). A much more sophisticated approach is described in [19], whose *fully indexable dictionaries* make it possible to store the distributor f in $(n/b)\log(bu/n)$ bits (plus lower-order terms); each g_i requires $(\gamma + \log b)b$ bits. So we need altogether

$$\frac{n}{b}\log\frac{bu}{n} + (\gamma + \log b)n \text{ bits.}$$

This quantity is minimised letting $b = -W(-ne/u)$, so when $n \ll u$ (using the approximation $W(x) \approx -\ln(-1/x) - \ln\ln(-1/x)$)

$$b \approx \ln\frac{u}{ne} + \ln\ln\frac{u}{ne} \approx \ln\frac{u}{n} + \ln\ln\frac{u}{n} - 1$$

and we need approximately

$$\Bigl(\gamma + \log e - \log\log e + \log\log\frac{u}{n}\Bigr)n \approx \Bigl(2.14 + \log\log\frac{u}{n}\Bigr)n$$

bits.

The time required to access the data structure is now dominated by the computation of the rank (see [19]; the cost should be actually multiplied by $\log u$ as we do not assume that $\log u$ is equal to the word size).

We propose, however, an interesting alternative based on the observation that both tries and the above dictionaries are able to rank *any element of* u on D. This is actually not necessary, as we need just ranking the elements of S.

5.1 Partial compacted tries.

When using a compacted trie for the computation of rank, one has to compare at each step the sequence contained in the currently visited node (say p) with the same-length prefix x of the element that is sought. If $x < p$ or $x > p$, the visit ends at this point (the sequence was, respectively, smaller than or larger than the elements of D corresponding to the leaves that descend from the current node). If $x = p$, we must continue our visit on the left or right subtrie, depending on the $|p|$-th bit of the string (this is what happens, in particular, for the elements of D).

In this section we introduce a new data structure, the *partial compacted trie*, that reduces the space usage of a compacted trie if you know in advance that you want to rank only strings out of a certain set $S \supseteq D$ (as opposed to ranking all strings in u).

To understand the idea behind this, suppose that you build the standard compacted trie of D, and that the root is labelled by p: when the trie is used to rank an element of S that has p as prefix, the visit continues on the left or right subtrie; this happens, in particular, for all elements of D (because p is the least common prefix of D) and, more generally, for all the elements of S that are between $\min D$ and $\max D$ (because they also have prefix p). On the other hand, some of the remaining elements of S (those that do not start with p and are smaller than $\min D$ or larger than $\max D$) cause the visit to end at the root. Now, in many cases it is not necessary to look at the *whole* prefix of length $|p|$ to understand that the visit must end; for example, suppose that $p = 01010$ but all the elements smaller than $\min D$ start with 00 and all elements larger than $\max D$ start with 11: then the first two bits of p are enough to determine if the visit must stop, or if we must proceed to some subtrie. We might store 01??? to mean that if the string starts with something *smaller* (larger, resp.) than 01, then it falls on the left (right, resp.) of the whole D, otherwise, we can ignore the following three bits (whose values are anyway fixed for the remaining elements of S), and proceed in the usual way looking at the sixth bit.

This intuitive idea is formalised as follows: a *partial compacted trie* (PaCo trie, for short) is a binary tree in which every node contains not a binary string but rather a pattern formed by a binary string followed by zero or more "don't know" symbols (?), for instance, "00101???". Given a PaCo trie, and an $x \in u$, the visit of the trie with x starts from the root and works as follows:

- if the node we are visiting is labelled by the pattern $w?^k$, we compare the first $|w|$ symbols of x (possibly right-padded with zeroes), say x', with w:
- if x' is smaller than w, or if $x' = w$ and the current node is a leaf, we end the visit and return the number of leaves to the left of the current node;
- if x' is larger than w, we end the visit and return the number of leaves to the left of the current node *plus* the number of leaves in the subtrie rooted at the current node;
- if $x' = w$ and the current node is not a leaf, let b be the $(|w| + k)$-th bit of x, and y be the suffix following it: we recursively visit the left or right subtrie (depending on whether $b = 0$ or $b = 1$) with y.

Differently from a standard compacted trie, the construction of a PaCo trie depends both on S and D; it is similar to the construction of a standard trie, but instead of taking the whole least common prefix p of the elements of D, we just take the smallest part that is enough to disambiguate it from all the elements of S that do not start with p (and that are, necessarily, all smaller than $\min D$ or larger than $\max D$). In particular, it is enough to find the largest element smaller than $\min D$ that does not start with p (say s') and the smallest element larger than $\max D$ that does not start with p (say s''); these two elements alone are sufficient to determine how short we can take the prefix of p: if the prefix we take is enough to understand that s' falls on the left, then the same will *a fortiori* happen for smaller elements of S, and similarly for s''.

Formally, we define the PaCo trie associated with a set $S = \{s_0 < \cdots < s_{n-1}\} \subseteq u$ with respect to $D = \{s_{i_0} < \cdots < s_{i_{k-1}}\}$ as follows (the reader may find it easier to understand this construction by looking at Figure 4):

- let p be the least common prefix of D;
- let j' be the largest integer in $\{0, \ldots, i_1 - 1\}$ such that p is not prefix of $s_{j'}$, and ℓ' be the length of the longest common prefix between p and $s_{j'}$; in the case all strings in $s_0, \ldots, s_{i_1-1}$ start with p, we let $\ell' = |p| - 1$ and $j' = 0$;
- let j'' be the smallest integer in $\{i_k + 1, \ldots, n - 1\}$ such that p is not prefix of $s_{j''}$, and ℓ'' be the length of the longest common prefix between p and $s_{j''}$; in the case all strings in $s_{i_k+1}, \ldots, s_{n-1}$ start with p, we let $\ell'' = |p| - 1$ and $j'' = n$;
- let now $\ell = 1 + \max(\ell', \ell'')$, and q be the prefix of length ℓ of p;
- if $k = 1$, the PaCo trie of S w.r.t. D is the one-node PaCo trie labelled with $q?^{|p|-\ell}$;
- if $k > 1$, let j be the smallest index such that s_j starts with $p1$, and t be the smallest index such that $j \leq i_t$: the PaCo trie of S w.r.t. D has root labelled with $q?^{|p|-\ell}$, and the left and right subtries are defined as follows[8]:
  - the left subtrie is the PaCo trie of $\{s_{j'+1} - p, \ldots, s_{j-1} - p\}$ with respect to $\{s_{i_0} - p, \ldots, s_{i_{t-1}} - p\}$;
  - the right subtrie is the PaCo trie of $\{s_j - p, \ldots, s_{j''-1} - p\}$ with respect to $\{s_{i_t} - p, \ldots, s_{i_{k-1}} - p\}$.

The PaCo trie for the example of Figure 1 is shown in Figure 5.

The construction is justified by the following theorem:

Theorem 1 *Let $D \subseteq S \subseteq u$ and T be the PaCo trie of S with respect to D. Then, for every $x \in S$, the visit of T with x returns $|\{y \in D \mid y < x\}|$.*

Proof. We will use the same notation as above, and show that the visit of a PaCo trie with an $x \in S$ follows the same route as a visit to the full compacted trie built over D. Let us suppose that $x = s_m$. If $j' < m < j''$, then p is a prefix of x, hence also q is a prefix of x, and we proceed with the visit going down to the left or right subtrie, as we would do with the full compacted trie. If $m = j'$ (the case $m = j''$ is analogous) is treated as follows: q is strictly longer than the longest common prefix between p and $s_{j'}$ (because $|q| > \max\{\ell', \ell''\} \geq \ell'$); so the prefix of x of length $|q|$, say q', is different from q, and it must necessarily be lexicographically smaller than it. We therefore end the visit, as we would do in the full trie (x is smaller than all keys). If $m < j'$ (the case $m > j''$ is analogous) follows *a fortiori*, because the prefix of x of length $|q|$ will be even smaller than (or equal to) q'. ▮

In our general framework, we will use a PaCo trie as a distributor; more precisely, given $S = \{s_0, \ldots, s_{n-1}\}$ consider, for some positive integer b, the set $D = \{s_0, s_b, s_{2b}, \ldots\}$: the PaCo trie of S with respect to D is a distributor implementing the function $f : S \to \lceil n/b \rceil$ that maps s_m to $\lfloor m/b \rfloor$. In this application, all buckets have size b (except possibly for the last one, which may be smaller).

[8]Here, we are writing $s - p$ for the string obtained omitting the prefix p from s.

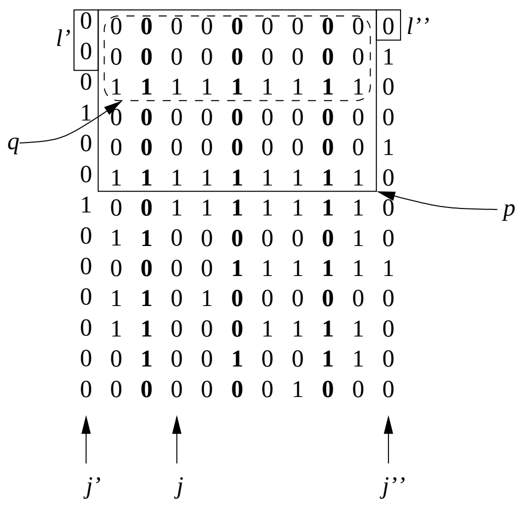


Figure 4: Constructing the first level of the PaCo trie for S with respect to D (see Figure 1): the central box corresponds to the longest common prefix $p = 001001$; here $\ell' = 2$ and $\ell'' = 1$, hence $\ell = 3$ giving rise to $q = 001$.

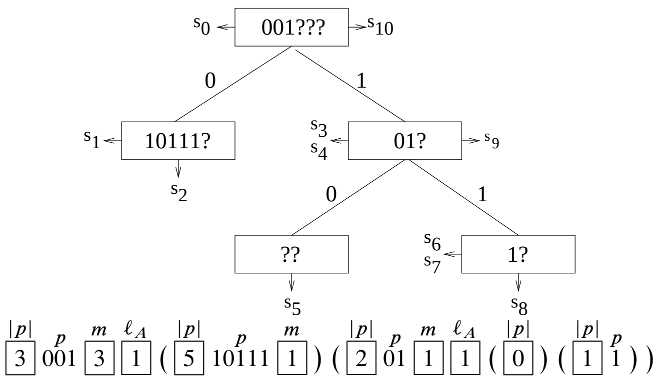


Figure 5: The PaCo trie for S with respect to D (see Figure 1): like for the trie in Figure 3, we can use this data structure to rank with respect to D, but *only* for elements of S, and not for arbitrary elements of the universe. At the bottom, we show the recursive bit stream representation: all framed numbers are written in δ coding. The skip component s is omitted but we use parentheses to isolate the parts of the stream corresponding to each subtree.

5.2 Implementation issues.

Experimental evidence suggests that PaCo tries usually save $30-50\%$ of the space occupied for paths compared to a standard trie. To get most benefits from this saving, we propose to store PaCo tries in a *recursive bit stream format*. More precisely, the representation $[\![T]\!]$ of a trie T whose root contains a bit string p followed by m don't-care symbols, left subtrie A and right subtrie B is given by the concatenation

$$s\ |p|\ p\ m\ \ell_A\ [\![A]\!]\ [\![B]\!],$$

where s is the length in bits of $[\![A]\!]$, ℓ_A is the number of leaves of A, and all numbers are represented in δ coding. Leaves have $s = 0$, and they do not record the information that follows p. Figure 5 shows the encoding of our example trie.

This representation (which is trivially decodable) makes it possible to navigate the PaCo trie in at most $\log u$ steps, as the left subtrie of a node is available as soon as the node data is decoded, and it is possible to jump to the right subtrie by skipping s bits. Moreover, at each navigation step we can compute in constant time the number of leaves at the left or under the current node using ℓ_A.[9]

By sizing buckets appropriately, the space usage of this representation is linear in n, and indeed we will see that using a PaCo trie as a distributor provides in practice the best space/time tradeoff for long keys.

We observe, however, that due to the difficulty of estimating the bit gain of a PaCo trie w.r.t. a dictionary or a standard trie, it is very likely that the computation of the optimal bucket size is far from the best value. Since trying exhaustively all bucket sizes is out of the question, we propose the following heuristic approach: we first estimate the bucket size using the formula above (the one that assumed the usage of a dictionary). Then, we compute the PaCo trie and assume that halving the bucket size (thus doubling the number of delimiters) will also approximately double the size of the PaCo trie. Using this simple model, we compute a new estimation of the bucket size and build the corresponding structure. In practice, this approach provides a very good approximation.

The algorithm given in the previous section requires accessing the strings of S and D randomly. However, there is a two-pass sequential algorithm that keeps in internal memory a representation of the trie built on D, only; this is feasible, since the optimal bucket size is rather large ($O(\ln u/n)$). The first pass builds the trie on D, whereas the second keeps track of the longest prefix of each compacted path that we actually need to be able to dispatch all elements of S to the leaf they belong.

6 Succinct hollow tries

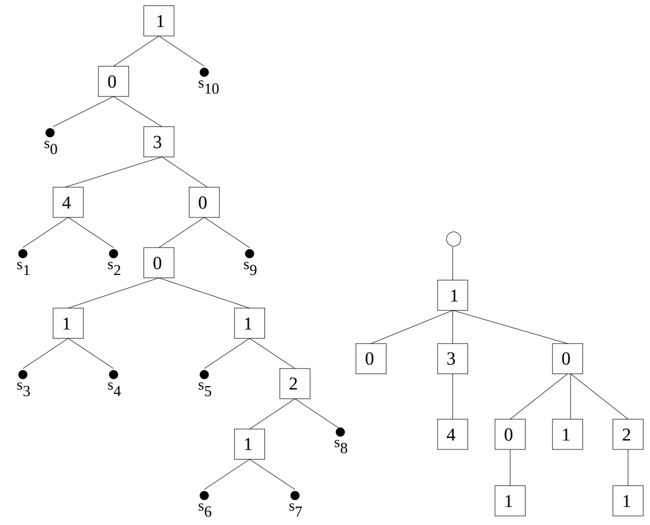
The *hollow trie* associated with S is a compacted trie [26] in which all paths of internal nodes have been replaced with their length, and all minimal paths between an internal node and a leaf have been discarded. In more detail, given S, we can define the hollow trie inductively as follows:

- if $|S| \leq 1$, the associated hollow trie is the empty binary tree;
- otherwise, if p is the longest common prefix of strings in S, the associated hollow trie is a binary tree whose root is labelled by $|p|$, and whose left and right subtrees are the hollow tries associated with the sets $\{\, x \in \{0,1\}^* \mid pix \in S \,\}$ for $i = 0, 1$, respectively.

The hollow trie for the example of Figure 1 is shown in Figure 6.

Note that a hollow trie is very similar to the *blind trie* that underlies a Patricia trie [29]: however, in a blind trie we keep track of the lengths of all compacted paths, whereas in a hollow trie the lengths of paths to the leaves are discarded. Indeed, the blind trie of a single string is given by a

[9] We remark that in principle ℓ_A can be reconstructed by visiting the trie. However, adding it to the structure makes it possible to return the correct value immediately after locating the exit node.



$$\mathbf{(}\ \ (_1\ (_0\ )\ (_3\ (_4\ )\ )\ (_0\ (_0\ (_1\ )\ )\ (_1\ )\ (_2\ (_1\ )\ )\ )\ )\ \ \mathbf{)}$$

Figure 6: The hollow trie for the set S of Figure 1, and the associated forest (in this case, a tree); at a node labelled by i, look at the i-th bit (numbered from 0), follow the corresponding arc, and discard all bits up to the i-th (inclusive). At the bottom, we show the corresponding representation by balanced parentheses; the bold pair corresponds to the fictitious (round) node.

node containing the first character of the string and the number of remaining characters, whereas the hollow trie of a single string is the empty binary tree.

We store hollow tries by computing the corresponding forest (in the first-child/next-sibling representation), adding a node at the top, and using Jacobson's representation [23] to store the string of balanced parentheses associated with the tree. Thus, we use two bits per node plus $\log\log u$ bits for each label of an internal node (actually, in practice using a variable-length bit array for the labels provides significant savings). We recall that Jacobson's representation divides parentheses in blocks of $\log n$ elements and stores explicitly the closing parentheses corresponding to a subset of open parentheses called *pioneers*. In our case, we use 64-bit blocks; we store the list of positions of open pioneers using an Elias–Fano monotone sequence, and the distance of the corresponding closed parenthesis using an Elias–Fano compressed list. In practice, this brings down the space needed to less than one bit per node. Much more sophisticated schemes (e.g., [16, 30]) achieve in theory $o(n)$ bits usage, but they require either to store both open and closed pioneers (we need just open pioneers—see below) or to apply the standard trick of using several levels of blocking, which in our case leads to a very small space gain and a significant slowdown. Figure 6 shows our example trie, the corresponding forest, and the associated balanced parentheses.

All in all, just

$$(2 + \log\log u)\, n$$

bits are sufficient, plus the bits required to store the structure for balanced parentheses. Traversing such a trie on the succinct representation is an easy process: if we are examining the open parenthesis at position p, moving on the left is just moving on the open parenthesis at position $p + 1$ (if the parenthesis at $p + 1$ is closed, the left binary subtree is empty), whereas moving on the right is just matching the current parenthesis, getting position q, and moving to position $q + 1$ (if the parenthesis at $q + 1$ is closed, the right binary subtree is empty). For every element $x \in S$ the resulting leaf will be exactly the leaf associated with x; of course, since we have discarded all paths, the result for strings not in S will be unpredictable.

We note that while walking down the trie we can easily compute the number of strings in S that are lexicographically smaller then w and the number of open parentheses that precede the current one. When we move to the left subtree, the first number does not change and the second one is incremented by one; when we move to the right subtree, we must add to both numbers the number of leaves in the left subtree: this number is exactly $(q - p + 1)/2$ — the number of open parentheses between p and q, using the same notation as in the previous paragraph. This observation makes it possible to get the skip associated with each node, and also to return the correct value as soon as the right leaf has been found.

Note that the process above requires time $O(\log u)$, albeit it will be more like $O(\log n)$ for non-pathological sets. However, the constant factors associated with the navigation of the succinct representation are very high. In theory and in practice, a hollow trie occupies less space than the previous two structures, but we will see that its main usefulness lies in the possibility of using it as the base of a distributor.

6.1 Variable-size slicing for large n.

Similarly to the case of longest common prefixes, for dense sets we can get to a space usage $O(\log\log(u/n))$ by dividing the elements of S in slices on the basis of their first $\log n$ bits. We can record the start of each bucket using a $2n$-bits Elias–Fano representation, and concatenate the bit representations of the hollow tries of each slice. This gives a space usage of

$$\left(4 + \log\log\frac{u}{n}\right) n \text{ bits.}$$

This modification is not going to be practical unless n is very close to u ($n \geq u^{3/4}$).

6.2 Implementation issues.

The speed of balanced parentheses operations is of course crucial in implementing a hollow trie. We avoid tables by using new *broadword algorithms* in the spirit or [31], and use other standard constructions (such as Elias–Fano lists) to reduce the space usage.

Another important issue is the storage of the skips. As in the case of a compacted trie, it is difficult to estimate the actual size of the skips of a hollow trie; we store them in an Elias–Fano list, bringing space usage to

$$(4 + a + \log a)\, n$$

bits, where a is as in Section 3.3. It is possible to derive a bound based on the string length $\log u$. If s_i represents the value of the i-th skip we have (by Jensen's inequality):

$$a = \sum_i \lfloor \log(s_i + 1) \rfloor/(n - 1) \leq \log \sum_i \frac{s_i + 1}{n - 1} + 1 \leq \log(\log u + 1) + 1.$$

Finally, we describe the construction algorithm, which never keeps a non-succinct representation of the trie in main memory. To get this result, we note that a binary tree with subtrees A and B is represented as an open parenthesis, followed by the representation of A, followed by a closed parenthesis, followed by the representation of B. Thus, instead of building a standard hollow trie and then computing the succinct representation, we represent our trie using a stack of succinct representations during the construction. The stack contains the nodes on the rightmost path of the trie, and the succinct representations contain the left subtrees of each node on the stack. Each time a new string has to be added, we have to split one of the nodes on the stack: as a result, all the representations of the following nodes on the stack are combined into a single representation, and a new node is added at the end of the stack.

In this way, each node of the trie requires two bits, plus an integer for the skip[10] during construction (the memory used by the stack is of course negligible for large sets). Combining representations after a split requires adding a parenthesis pair around each representation, and concatenating them. It is easy to check, however, that the overall concatenation work is linear in the input size, as the data associated with a key (two bits plus skip) is first added at a distance in bits from the root that is equal to the string length. Each combination moves the data towards the root of at least one position. Thus, overall, the cost of copying is linear in the number of bits of the strings.

7 Bucketing with hollow tries

We now turn to the slowest data structure, which however makes it possible to use just $O(\log\log\log u)$ bits per element: in practice, unless the string length exceeds a billion, any data set can be monotonically hashed using less than one byte per element in time $O(\log u)$.

The basic idea is that of using a succinct hollow trie built on the delimiters as a distributor. Since skips require $\log\log u$ bits each, we can use the same size for our buckets; then, the occupation of the trie will be linear in the number of keys, and the bucket offset will require just $\log\log\log u$ bits.

It is however very easy to check that a hollow trie is not sufficient to map correctly each key into its bucket: in the example of Figure 1, if we built a hollow trie on the delimiters, the string s_{10} would be misclassified (more precisely, mapped to the first bucket), as the first test would be on the seventh bit, which is zero in s_{10}.

The idea we use to solve this problem is to mimic the behaviour of a trie-based distributor: all we need to know is, for each node and for each key, which behaviour (exit on the left, exit on the right, follow the trie) must be followed. This information, however, can be coded in very little space. To see how this can be done, consider what happens when visiting a trie-based distributor using a key x.

At each node, we know that x starts with a prefix that depends only on the node, and that there is a substring s of x, following the prefix, that must be compared with the compacted path stored at the node. The behaviour of x at the node depends *only* on the comparison between s and the compacted path. The problem with using a hollow trie as distributor is that we know just the *length* of the compacted path, and thus also s, but we do not know the compacted path.

The solution is to store explicitly the function (depending on the node) that maps s to its appropriate behaviour (exit or not). The fundamental observation is that the number of overall keys of such a map (upon which the space usage depends) is at most n (actually, in practice most keys exit on leaves, so this number is closer to n/b). Finally, we need a map with n keys that tells us whether we exit on the left or on the right, for the prefixes that exit. Figure 7 shows the distributor obtained for our example.

All in all, the space requirement is $n/b(2+\log\log u)$ for the trie, γn bits to store the exit/non-exit behaviour on internal nodes, and γn bits to store the left/right behaviour using MWHC functions[11]; finally, we need $\gamma n \log b$ bits to store the offset of each key into his bucket, resulting in

$$\frac{n}{b}(2 + \log\log u) + 2\gamma n + \gamma n \log b$$

bits. This is minimised when

$$b = \frac{\ln 2}{\gamma}(\log\log u + 2),$$

resulting in

$$\gamma\Bigl(\frac{1}{\ln 2} + 2 + \log\frac{\ln 2}{\gamma} + \log(2 + \log\log u)\Bigr)n$$

[10]In principle, we could use Elias–Fano lists to hold skips even during the construction process. This, however, would significantly slow down the combination phase.

[11]Actually, these are upper bounds, as it might happen that some strings exit on a leaf.

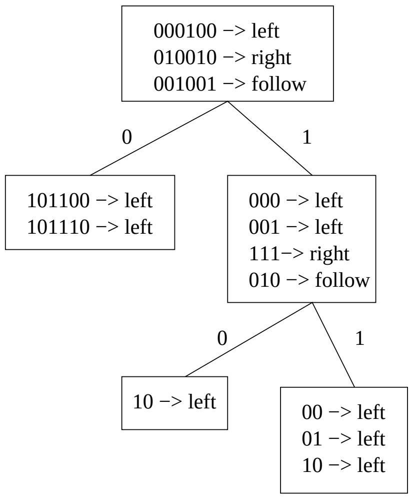


Figure 7: The distributor based on a hollow trie built from the set D of Figure 1. This data structure can be used to rank elements of S with respect to D (the reader is invited to compare with Figure 3). The picture shows, for each node of the hollow trie, the associated functions: for the sake of readability, we compacted the exit/follow map with the exit-right/exit-left map.

overall bits, that is, approximately $3.21 + 1.23 \log\log\log u$ bits per element. The time required to hash a string is $O(\log u)$, as in the case of the hollow trie; at each node we have to compute a MWHC function, but since the overall length of all the strings involved in such computation is bounded by $\log u$, this has no asymptotic effect on the computation time.

A more precise estimation of the actual space usage should take into account the average length of a skip and the related Elias–Fano list bound to compute a more accurate value (see Section 3.3).

7.1 Implementation issues.

Storing n/b pairs of functions explicitly would significantly increase space usage. Rather, we store just two functions mapping pairs node/string to the associated action.

The distributor we discussed can be easily built using a two-pass linear procedure. In the first pass, a hollow trie on the delimiters is built. In the second pass, we consider all the keys in order, and for each key x we keep track of the two subsequent delimiters d and d' such that $d \leq x \leq d'$. Let p be the length of the longest common prefix of d and d': the key x will exit on the right/left at some node along the branch to d/d', depending on the p-th bit of x. The actual exit node can be determined computing the longest common prefix between x and d (or d'). In this way, we compute for each key the action at the exit node, and store it in a temporary file; moreover, we record all paths leading to a “follow” action and all paths leading to an “exit” action, taking care of never saving twice the “follow” action at an internal node (“follow” actions are produced when the hollow trie is visited with the delimiters). At the end of the second pass, we read the temporary files and build the necessary MWHC functions. Finally, we compute the MWHC function mapping each key to its offset.

8 Bucketing with z-fast tries

In [1] we have proposed a data structure for the monotone minimal perfect hashing problem which has space usage $O(\log\log\log u)$ bits per element and access time $O((\log u)/w + \log\log u)$ (thus, access time is logarithmic in the string length $\log u$ if $w \leq c \log u$). The structure used as distributor, called a *z-fast trie*, is sketched in the following. We invite the interested reader to consult [1] for the details.

Consider the compacted trie of delimiters and a query key $x \in S$. The basic idea is that the query algorithm should perform a binary search for the (depth of) the edge of the trie that contains the longest prefix of x, i.e., where a leaf corresponding to x would branch from the trie. Once this branching edge a is found, there are two possibilities for the rank of x depending on whether x branches to the left or to the right: this information is stored as a function $f : S \to \{0, 1\}$. A generalised rank data structure, called a *leaf ranker*, is used to compute the rank, which for the subtree below a is either the maximum rank or minimum rank minus 1, depending on the value of $f(x)$.

The main technical contribution of [1] lies in devising a small-space data structure that supports the binary search. The data structure is a function T that maps selected prefixes of the delimiters into signatures of $O(\log\log u)$ bits. If a prefix x_p of x maps to the signature of x_p, with high probability x_p is one of the selected prefixes (and hence the longest common prefix between x and a delimiter is at least $|x_p|$ bits long). Conversely, we make sure that for any prefix x_p considered in the binary search, if $T(x_p)$ is different from the signature of x_p this implies that x_p is not a prefix of a delimiter, i.e., the branching edge is at a lower depth. This idea leads to a data structure using space $O(n(\log\log u)^2)$. To further reduce the space usage we make use of an alternative to standard binary search, where each step determines one bit of the end position. This leads to larger overlaps in the search paths for different elements in S, and the claimed space usage. To achieve the stated query time we make use of fast algorithms for computing hash values of all prefixes of a string of words in linear time.

We have implemented the z-fast trie using the techniques described in this paper: the formula giving the space usage, however, is rather knotty. We thus sketch the space required by various part of the data structure, and provide the optimal bucket size b. For details about the various pieces, we refer the reader to [1].

First of all, the z-fast trie is represented by an MWHC function mapping a set of n/b *handles*, which represent nodes of the compacted trie built on the n/b delimiters, to the length of the path compacted at the respective node ($\log\log u$ bits) and to a signature of $\log b + \log\log\log u$ bits, giving overall

$$\frac{n}{b}(\gamma + \log\log u + \log\log\log u + \log b)$$

bits.

Then, we have the leaf ranker. We must hash monotonically each of three strings associated with each extent ($3n/b$ strings overall). We provide the space bound for the standard LCP-based solution, even if our implementation uses a two-step map, as we have no information on the distribution of the keys; there is also an additional bit per key as the monotone hash function maps each string into a suitable bit vector:

$$\frac{3n}{b}\left(1 + \gamma + \log e - \log\log e + \log\log\frac{3n}{b} + \log\log\frac{u}{\log(3n/b)}\right)$$

Then, we must store $\log b$ signatures of the expected n/b elements which fill fail to walk correctly through the z-fast trie. In turn, these signatures will generate at most n/b false positives, so we also need $2\gamma b$ bits for a one-bit function that tells false from true positives, and finally $\log\log u$ bits for each failing element to store the exact result. The number of bits is:

$$\frac{n}{b}(\log b + 2\gamma + \log\log u)$$

Finally, we need a one-bit function (γn bits) for storing the exit direction of each key, and the usual $\gamma n \log b$ bits to store the offset of each key into its bucket (buckets will be very small, so a non-compacted MWHC function is usually better).

Adding up all the cost factors, and using the approximations $\log\log(3n/b) \approx \log\log(3n)$ and $\log\log(u/\log(3n/b)) \approx \log\log u$, we can solve exactly the minimisation problem of the cost function, obtaining a rather ugly expression involving Lambert's W, again. Some careful estimation and elimination of lower-order terms leads to the following expression for the optimal bucket size:

$$b = 10.5 + 4.05 \ln\ln u + 2.43 \ln\ln n + 2.43 \ln\ln\ln u.$$

9 Average length

In this section we note the few modifications that are necessary to make the space usage of our structures proportional to $\log \ell$, where ℓ is the average length of a set of prefix-free, variable-length strings S. The main ingredient is again the Elias–Fano representation. Note, however, that on the files used for our experiments (see Section 10) the (small) additional constant costs in bits per key due to the additional structures makes the final hashing structure actually more expensive than the original one in terms of bits per key. However, generally speaking a bound in terms of the average length makes the method robust against the situation where a few keys in S are much longer than the average.

In the case of bucketing based on longest common prefixes, we simply observe that since the longest common prefix of a bucket is shorter than the shortest string in the bucket, the sum of lengths of all prefixes does not exceed $(n/b)\ell$, so we can replace the function storing the prefixes length with a minimal perfect hash function (2γ bits per prefix) followed by an Elias-Fano list ($\log((n/b)\ell)/(n/b)) = \log \ell$ bits per prefix). The analysis of Section 4 can be carried out using ℓ in place of $\log u$, obtaining

$$(2\gamma + 2 + \log \ell + \log\log \ell + \log b)n + \Bigl(\gamma + \log \frac{n}{b}\Bigr)\frac{n}{b}.$$

The value of b minimising the expression is always the same, so we get to

$$(2\gamma + 2 + \log \ell + \log\log \ell + \log e - \log\log e + \log\log n)n \\ \approx (5.37 + \log \ell + \log\log \ell + \log\log n)n$$

overall bits.

In the case of bucketing based on PaCo tries, we choose as delimiters the *shortest string* of each bucket, and then again the overall number of bits in the trie paths cannot exceed $(n/b)\ell$ (*in lieu* of the obvious bound $(n/b)\log u$). In this case, however, bucket offsets can be twice larger than the chosen bucket size (because of the unpredictable location of the delimiters), so we need an additional bit per key.

Finally, in a hollow trie the sum of skips cannot exceed the overall string length ℓn, so using the notation of Section 6.2 in the variable-length case we have $a \leq \log \ell + 1$.

Similar results can be obtained for structures using $O(\log\log\log u)$ bits per element: however, even more in that case the triple-log makes any advantage unnoticeable.

10 Experiments

In this section we discuss a set of experiments carried out using the data structures we introduced, and, for comparison, an order-preserving hash function computed by an MWHC compacted function and a standard minimal perfect hash function (built using the technique described in Section 3.2). We used Java for all implementations; the tests were run with a Sun JVM 1.6 on a 64-bit Opteron processor running at 2814 MHz with 1 MiB of first-level cache.

For our tests, we used a number of files that we describe in detail:

File	`trec-title` 137 b/key longest: 4576 b			`trec-text` 197 b/key longest: 360016 b			`webbase-2001` 481 b/key longest: 81696 b			`uk-2007-05` 898 b/key longest: 16384 b			`uk` 1044 b/key longest: 25680 b			64-bit random			32-bit random		
	b/key	μs/key	c.t.	b/key	μs/key	c.t.	b/key	μs/key	c.t.	b/key	μs/key	c.t.	b/key	μs/key	c.t.	b/key	μs/key	c.t.	b/key	μs/key	c.t.
Plain string encoding																					
MWHC	22.55	0.48	6.6	27.55	0.73	3.1	28.51	1.10	4.3	28.52	1.27	5.4	31.55	1.48	6.1	28.52	0.76	3.5	28.52	0.69	2.3
MPH	2.71	0.35	9.5	2.71	0.57	3.5	2.67	0.93	5.4	2.68	1.20	10.5	2.70	1.25	6.0	2.68	0.50	2.9	2.68	0.45	2.1
LCP	15.71	0.77	3.0	21.26	1.14	4.4	24.25	1.65	6.7	21.26	1.91	7.7	21.38	2.16	9.4	12.25	1.28	4.2	12.25	1.06	2.8
VLLCP	18.12	0.98	5.2	18.88	1.46	14.0	20.70	2.22	8.8	21.88	2.66	9.4	22.18	3.07	9.8	16.68	1.45	5.0	16.68	1.52	3.9
LCP (2-step)	13.02	0.94	6.7	14.64	1.47	14.8	17.85	2.29	9.4	18.94	2.71	10.0	19.07	3.06	12.1	10.80	1.28	6.0	10.80	1.09	4.9
PaCo	8.49	1.72	5.9	8.71	2.60	13.8	9.90	4.04	13.0	10.59	4.40	13.9	10.67	5.66	16.1	7.09	1.64	5.9	6.84	1.58	3.7
VLPaCo	9.03	1.93	5.3	9.02	2.92	11.2	10.15	4.43	10.3	10.72	4.92	10.0	10.99	5.81	12.2	8.34	2.06	4.0	8.08	1.89	3.2
Hollow	6.74	5.62	1.9	6.91	8.08	3.1	7.21	15.29	2.5	7.57	15.78	2.5	7.87	19.09	3.0	4.41	6.84	.9	4.37	6.78	.7
HTDist	5.99	10.64	15.4	5.95	17.70	15.7	6.17	26.70	19.5	6.15	29.06	22.4	6.23	35.15	20.4	5.15	13.17	13.0	5.15	12.35	11.2
ZFast	9.24	4.54	13.1	9.78	6.25	15.9	10.14	9.09	19.5	9.64	11.27	24.9	9.72	12.78	27.5	8.47	5.74	9.2	8.46	5.99	9.0
Hu–Tucker encoding																					
LCP	13.71	1.28	4.6	21.27	1.63	5.5	23.25	4.26	12.0	21.26	6.93	16.3	21.38	7.92	17.4	12.25	1.59	5.2	12.25	1.47	3.1
VLLCP	16.02	1.96	8.2	16.92	2.49	6.3	19.91	7.53	12.5	21.23	12.74	17.0	21.57	14.83	18.4	16.68	2.29	5.8	16.68	1.76	3.9
LCP (2-step)	12.69	1.89	8.7	13.94	2.34	7.7	17.68	7.53	14.3	18.68	12.73	18.8	18.95	14.93	20.2	10.80	2.08	8.0	10.80	1.47	5.4
PaCo	7.37	1.76	7.3	7.61	2.47	15.9	9.52	6.18	37.1	9.95	9.10	59.4	10.12	10.98	69.0	7.05	2.06	9.9	7.37	1.90	5.5
VLPaCo	8.10	1.88	5.7	8.20	2.56	13.5	9.75	6.43	29.2	10.46	9.35	46.3	10.64	11.33	69.2	8.30	2.34	6.7	8.63	2.38	5.7
Hollow	5.09	5.43	2.6	5.23	7.48	2.7	6.70	17.03	7.9	7.11	21.54	12.9	7.42	26.98	15.2	4.41	7.66	1.5	4.37	6.86	1.1
HTDist	5.54	8.88	17.4	5.52	13.47	16.9	6.04	29.64	34.1	6.04	33.64	49.0	6.08	43.25	59.4	5.15	13.07	15.8	5.15	12.87	12.7
ZFast	9.10	4.73	14.6	9.64	6.87	15.7	10.07	10.58	34.5	9.57	15.01	53.6	9.63	17.35	82.1	8.47	6.11	10.9	8.48	6.14	9.0

Table 1: For each of the test files, we show the number of bits per key and the length in bits of the longest key. Then, for each combination of test file and hash function we show the number of bits per key, the number of microseconds per key of a successful probe (unsuccessful probes are actually faster), and the time spent constructing the data structure ("c.t.") expressed again in microseconds per key. The first table uses the encoding of each file as specified in Section 10, whereas the second table shows structures built using optimal lexicographical Hu–Tucker encoding.

File	`trec-title` 137 b/key			`trec-text` 197 b/key			`webbase-2001` 481 b/key			`uk-2007-05` 898 b/key			`uk` 1044 b/key			64-bit random			32-bit random		
	pred	actual	Δ	pred	actual	Δ	pred	actual	Δ	pred	actual	Δ	pred	actual	Δ	pred	actual	Δ	pred	actual	Δ
Plain string encoding																					
LCP	18.70	15.71	-16%	25.30	21.26	-16%	23.26	24.25	+4%	20.93	21.26	+2%	21.71	21.38	-2%	13.00	12.25	-6%	12.07	12.25	+1%
VLLCP	20.36	18.12	-11%	21.01	18.88	-10%	22.68	20.70	-9%	23.49	21.88	-7%	23.83	22.18	-7%	19.97	16.68	-16%	18.74	16.68	-11%
LCP (2-step)	13.60	13.02	-4%	15.04	14.64	-3%	18.37	17.85	-3%	19.14	18.94	-1%	19.20	19.07	-1%	11.07	10.80	-2%	11.07	10.80	-2%
PaCo	14.30	8.49	-41%	20.60	8.71	-58%	18.46	9.90	-46%	16.14	10.59	-34%	16.79	10.67	-36%	7.65	7.09	-7%	5.89	6.84	+16%
VLPaCo	14.30	9.03	-37%	20.60	9.02	-56%	18.46	10.15	-45%	16.14	10.72	-34%	16.79	10.99	-35%	7.65	8.34	+9%	5.89	8.08	+37%
Hollow	15.12	6.74	-55%	15.73	6.91	-56%	17.22	7.21	-58%	18.24	7.57	-59%	18.49	7.87	-57%	14.02	4.41	-69%	12.99	4.37	-66%
HTDist	7.14	5.99	-16%	7.19	5.95	-17%	7.32	6.17	-16%	7.41	6.15	-17%	7.43	6.23	-16%	7.03	5.15	-27%	6.92	5.15	-26%
ZFast	9.72	9.24	-5%	9.83	9.78	-1%	10.05	10.14	+1%	10.19	9.64	-5%	10.24	9.72	-5%	9.60	8.47	-12%	9.46	8.46	-11%
Hu–Tucker encoding																					
LCP	16.69	13.71	-18%	24.17	21.27	-12%	22.96	23.25	+1%	20.75	21.26	+2%	21.58	21.38	-1%	12.89	12.25	-5%	11.85	12.25	+3%
VLLCP	18.25	16.02	-12%	19.00	16.92	-11%	21.96	19.91	-9%	22.90	21.23	-7%	23.30	21.57	-7%	19.97	16.68	-16%	18.74	16.68	-11%
LCP (2-step)	13.20	12.69	-4%	14.61	13.94	-5%	18.17	17.68	-3%	19.07	18.68	-2%	19.13	18.95	-1%	11.07	10.80	-2%	11.07	10.80	-2%
PaCo	12.28	7.37	-40%	19.47	7.61	-61%	18.16	9.52	-48%	15.96	9.95	-38%	16.66	10.12	-39%	7.48	7.05	-6%	5.22	7.37	+41%
VLPaCo	12.28	8.10	-34%	19.47	8.20	-58%	18.16	9.75	-46%	15.96	10.46	-34%	16.66	10.64	-36%	7.48	8.30	+11%	5.22	8.63	+65%
Hollow	13.10	5.09	-61%	13.82	5.23	-62%	16.51	6.70	-59%	17.68	7.11	-60%	17.96	7.42	-59%	13.89	4.41	-68%	12.75	4.37	-66%
HTDist	6.93	5.54	-20%	7.00	5.52	-21%	7.26	6.04	-17%	7.36	6.04	-18%	7.38	6.08	-18%	7.01	5.15	-27%	6.89	5.15	-25%
ZFast	9.44	9.10	-4%	9.57	9.64	+1%	9.95	10.07	+1%	10.11	9.57	-5%	10.16	9.63	-5%	9.58	8.47	-12%	9.43	8.48	-10%

Table 2: A comparison of predicted and actual number of bits per key.

File	trec-title 9.5 MB 137 b/key	trec-text 440 MB 197 b/key	webbase-2001 7.1 GB 481 b/key	uk-2007-05 11.9 GB 898 b/key	uk 77.8 GB 1044 b/key	64-bit random 900 MB	32-bit random 500 MB
Plain string encoding							
MWHC	11.99 MB	187.36 MB	574.67 MB	543.98 MB	3.29 GB	501.04 MB	609.94 MB
MPH	3.80 MB	66.64 MB	228.34 MB	232.10 MB	628.73 MB	74.98 MB	349.20 MB
LCP	23.43 MB	243.10 MB	833.06 MB	631.46 MB	3.82 GB	368.29 MB	443.71 MB
LCP (2-step)	53.90 MB	225.31 MB	545.55 MB	501.49 MB	3.86 GB	347.31 MB	501.95 MB
PaCo	62.72 MB	766.13 MB	2.90 GB	1.58 GB	6.30 GB	2.80 GB	2.16 GB
Hollow	57.15 MB	753.65 MB	2.11 GB	1.19 GB	7.63 GB	1.68 GB	1.59 GB
HTDist	111.52 MB	1.04 GB	2.74 GB	501.16 MB	4.06 GB	2.30 GB	2.17 GB
ZFast	110.28 MB	1.02 GB	4.51 GB	4.15 GB	13.74 GB	2.67 GB	2.78 GB
Hu–Tucker encoding							
LCP	22.57 MB	227.54 MB	1.21 GB	1.81 GB	4.05 GB	1.49 GB	1.54 GB
LCP (2-step)	72.95 MB	475.15 MB	1.45 GB	2.10 GB	3.81 GB	1.40 GB	1.51 GB
PaCo	62.89 MB	646.37 MB	644.70 MB	593.87 MB	4.02 GB	2.21 GB	2.69 GB
Hollow	59.76 MB	551.95 MB	1.37 GB	1.66 GB	9.38 GB	1.86 GB	1.93 GB
HTDist	97.41 MB	2.39 GB	1.47 GB	1.21 GB	6.46 GB	752.37 MB	3.89 GB
ZFast	54.37 MB	847.24 MB	3.46 GB	3.31 GB	10.90 GB	2.66 GB	2.06 GB

Table 3: For each of the test files, we show the maximum amount of main memory used during the construction. Note that the amount shown is the space used by the Java heap, which is just an upper bound on the actual memory usage; the latter is rather difficult to estimate with a language based on garbage collection. From a theoretical viewpoint, all algorithms use $O(n(w + \ell))$ bits of memory, where ℓ is the average string length.

- `trec-title.terms` (9.5 MB, $\approx$ 1 million strings): the terms appearing in titles of the TREC GOV2 collection (UTF-8);
- `trec-text.terms` (440 MB, $\approx$ 35 million strings): the terms appearing in the text of the TREC GOV2 collection (UTF-8);
- `webbase-2001.urls` (7.1 GB, $\approx$ 118 million strings): the URLs of a general crawl performed by the WebBase crawler [21] in 2001 (ISO-8859-1);
- `uk-2007-05.urls` (11.9 GB, $\approx$ 106 million strings): the URLs of a 101 Mpages crawl of `.uk` performed by UbiCrawler [2] in May 2007 (ISO-8859-1);
- `uk.urls` (77.8 GB, $\approx$ 600 million strings): the joint URLs of thirteen snapshots of `.uk` gathered by UbiCrawler [3] between May 2006 and May 2007 (ISO-8859-1);
- `largerandom.bin` (900 MB, 100 million strings): random 64-bit strings;
- `smallrandom.bin` (500 MB, 100 million strings): random 32-bit strings.

The first two files represent typical text dictionaries. We provide two different URL lists because they are very different in nature (the most recent one has significantly longer URLs on average, whereas the first one contains a few very long URLs). The dataset with 600 million strings is useful to test scalability at construction time. Finally, the random string files are useful to test the behaviour of our structures in the presence of random strings. The latter, in particular, is extremely dense (n is close to u). All variable-length data sets were made into prefix-free sets by adding a NUL (code 0) character at the end of each string.

We remark that none of our constructors require loading the entire set of data to be processed into internal memory. Moreover, we do not assume any prior knowledge about the data, which implies that in some cases we have to make one pass just to gather statistics that are relevant to the computation of the optimal bucket size.

In all our implementations, bucket sizes are limited to powers of two. This guarantees that we make full use of the space occupied by offset functions, and accelerates division and modulo operations, which can be carried out using shifts and bit masks. We have verified experimentally that perturbing the bucket size increases the space usage, except for a very small number of cases, in which the gain is below 0.1%. This empirical consideration shows that our space computations are precise enough to produce actually optimal bucket sizes.

Table 1 reports the results of our experiments. Beside the plain encoding of each file, we also tried a more sophisticated approach based on Hu–Tucker codes [22]. Hu–Tucker codes are *optimal lexicographical codes*—they compress optimally a source reflecting, however, the order between the symbols of the source in the lexicographical order of the codewords (this entails a loss of space w.r.t. entropy, which is however bounded by 2 bits). It is interesting to experiment with Hu–Tucker codes because the increase of compression (or, better, a lack thereof) can be used as a measure of the effectiveness of our data structures (in case of binary numbers, Hu–Tucker codes were computed on bytes).

We accessed one million randomly selected strings from each set. The tests were repeated thirteen times, the first three results discarded (to let the Java Virtual Machine warm-up and optimise dynamically the code) and the remaining ones averaged.

In Table 3 we show the maximum amount of memory used at construction time; in Table 2 we show the difference between predicted and actual number of bits per key used by each structure.

- From the results of our experiments, PaCo-based monotone hash function has the best trade-off between space and time, but in case a high speed is required, LCP-based monotone hash functions have a much better performance.

- Variable-length variants of distributors based on LCPs and PaCo tries are not useful: in the case of LCPs, they are always beaten by the two-step version; in the case of PaCo tries, they actually increase space usage.
- There is no significant advantage in using Hu–Tucker coding. The only relevant effect is on LCP-based monotone hashing, as the size of the hash function is very dependent on the string length, and Hu–Tucker coding does reduce the string length; however, a more significant gain can be obtained by a two-step LCP-based function.
- The smallest structures come from succinct hollow tries—either used directly, or as a distributor, but accessing them is usually very expensive. The case of short strings, however, is different: here hollow tries provide a very significant compression gain, with a small loss in access time.
- Structures based on z-fast tries, albeit the best in theory, are beaten by PaCo-based structures in all our experiments. The gap, however, is small, and might suggest that some additional engineering might make z-fast tries the best also in practice.
- Our prediction are very good, except for trie-based. The problem with tries is that the actual measure depends on the *trie size* of the set, that is, the number of bits required to code the compacted paths of a trie representing the key set, or even of the *hollow trie size* of the set, that is, the number of bits required to code the *lengths* of the compacted paths of that hollow trie. Our estimates depends instead on the average length of a string and on the number of strings. Elsewhere, our estimates are almost always good upper bounds: in a few cases, however, the fact that we round the bucket size to a power of two causes the data structure to be slightly larger than expected (as our prediction use the theoretically best bucket size).
- Conforming to theoretical predictions, the space used by $O(\log\log\log u)$ structures fluctuates in a much narrower interval than the space for those using $O(\log\log u)$ bits per key. For instance, z-fast tries are not competitive in terms of space with PaCo tries on small and medium dataset, but they become competitive on the large `uk` dataset.

11 Conclusions

We have presented experimental data about some old and new data structures that provide monotone minimal perfect hashing in a very low number of bits. The conclusion is that our data structures are practical and cover a wide range of space/time tradeoffs.

Improvements to function storage (e.g., using $rn + o(n)$ methods) would yield immediately improvements to the data structures presented here. However, current methods appear to be too slow for practical implementations.

The speed of our structures is presently mainly constrained by memory latency and unaligned access. Future work will concentrate on reducing this burden without increasing space usage.

References

[1] Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. Monotone minimal perfect hashing: Searching a sorted table with $O(1)$ accesses. In *Proceedings of the 20th Annual ACM-SIAM Symposium On Discrete Mathematics (SODA)*, pages 785–794, New York, 2009. ACM Press.

[2] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. UbiCrawler: A scalable fully distributed web crawler. *Software: Practice & Experience*, 34(8):711–726, 2004.

[3] Paolo Boldi, Massimo Santini, and Sebastiano Vigna. A large time-aware graph. *SIGIR Forum*, 42(2):33–38, 2008.

[4] Fabiano C. Botelho, Rasmus Pagh, and Nivio Ziviani. Simple and space-efficient minimal perfect hash functions. In Frank K. H. A. Dehne, Jörg-Rüdiger Sack, and Norbert Zeh, editors, *Algorithms and Data Structures, 10th International Workshop, WADS 2007, Halifax, Canada, August 15-17, 2007, Proceedings*, volume 4619 of *Lecture Notes in Computer Science*, pages 139–150. Springer, 2007.

[5] Fabiano C. Botelho and Nivio Ziviani. External perfect hashing for very large key sets. In Mário J. Silva, Alberto H. F. Laender, Ricardo A. Baeza-Yates, Deborah L. McGuinness, Bjørn Olstad, Øystein Haug Olsen, and André O. Falcão, editors, *Proceedings of the Sixteenth ACM Conference on Information and Knowledge Management, CIKM 2007, Lisbon, Portugal, November 6-10, 2007*, pages 653–662. ACM, 2007.

[6] Denis Xavier Charles and Kumar Chellapilla. Bloomier filters: A second look. In Dan Halperin and Kurt Mehlhorn, editors, *Algorithms - ESA 2008, 16th Annual European Symposium, Karlsruhe, Germany, September 15-17, 2008. Proceedings*, volume 5193 of *Lecture Notes in Computer Science*, pages 259–270. Springer, 2008.

[7] Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, and Ayellet Tal. The Bloomier filter: an efficient data structure for static support lookup tables. In J. Ian Munro, editor, *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2004*, pages 30–39. SIAM, 2004.

[8] David R. Clark and J. Ian Munro. Efficient suffix trees on secondary storage. In *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 383–391. ACM/SIAM, January 28–30 1996.

[9] Martin Dietzfelbinger and Rasmus Pagh. Succinct data structures for retrieval and approximate membership (extended abstract). In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part I: Track A: Algorithms, Automata, Complexity, and Games*, volume 5125 of *Lecture Notes in Computer Science*, pages 385–396. Springer, 2008.

[10] Peter Elias. Efficient storage and retrieval by content and address of static files. *J. Assoc. Comput. Mach.*, 21(2):246–260, 1974.

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

[12] Edward A. Fox, Qi Fan Chen, Amjad M. Daoud, and Lenwood S. Heath. Order-preserving minimal perfect hash functions and information retrieval. *ACM Trans. Inf. Sys.*, 9(3):281–308, 1991.

[13] Michael L. Fredman and János Komlós. On the size of separating systems and families of perfect hash functions. *SIAM J. Algebraic Discrete Methods*, 5(1):61–68, 1984.

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

[15] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of reusable object-oriented software*. Addison-Wesley, 1995.

[16] R.F. Geary, N. Rahman, R. Raman, and V. Raman. A simple optimal representation for balanced parentheses. volume 368, pages 231–246. Elsevier, 2006.

[17] Alexander Golynski. Optimal lower bounds for rank and select indexes. In Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener, editors, *Automata, Languages and Programming, 33rd International Colloquium, ICALP 2006, Venice, Italy, July 10-14, 2006, Proceedings, Part I*, volume 4051 of *Lecture Notes in Computer Science*, pages 370–381. Springer, 2006.

[18] R. Gonzàlez, S. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In *Poster Proceedings Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA'05)*, pages 27–38. CTI Press and Ellinika Grammata, 2005.

[19] Ankur Gupta, Wing-Kai Hon, Rahul Shah, and Jeffrey Scott Vitter. Compressed data structures: Dictionaries and data-aware measures. *Theoretical Computer Science*, 387(3):313–331, 2007.

[20] Torben Hagerup and Torsten Tholey. Efficient minimal perfect hashing in nearly minimal space. In *Proceedings of the 18th Symposium on Theoretical Aspects of Computer Science (STACS '01)*, volume 2010 of *Lecture Notes in Computer Science*, pages 317–326. Springer–Verlag, 2001.

[21] Jun Hirai, Sriram Raghavan, Hector Garcia-Molina, and Andreas Paepcke. WebBase: a repository of Web pages. *Computer Networks*, 33(1–6):277–293, 2000.

[22] T. C. Hu and A. C. Tucker. Optimal Computer Search Trees and Variable-Length Alphabetical Codes. *SIAM J. Applied Math.*, 21(4):514–532, 1971.

[23] Guy Jacobson. Space-efficient static trees and graphs. In *Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science, FOCS'89 (Research Triangle Park, NC, October 30 - November 1, 1989)*, pages 549–554. IEEE, IEEE Computer Society Press, 1989.

[24] Bob Jenkins. Algorithm alley: Hash functions. *Dr. Dobb's Journal of Software Tools*, 22(9):107–109, 115–116, September 1997.

[25] Dong Kyue Kim, Joong Chae Na, Ji Eun Kim, and Kunsoo Park. Efficient implementation of rank and select functions for succinct representation. In Sotiris E. Nikoletseas, editor, *Proc. of the Experimental and Efficient Algorithms, 4th InternationalWorkshop*, volume 3503 of *Lecture Notes in Computer Science*, pages 315–327. Springer, 2005.

[26] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, second edition, 1997.

[27] Bohdan S. Majewski, Nicholas C. Wormald, George Havas, and Zbigniew J. Czech. A family of perfect hashing methods. *Comput. J.*, 39(6):547–554, 1996.

[28] Michael Molloy. Cores in random hypergraphs and Boolean formulas. *Random Struct. Algorithms*, 27(1):124–135, 2005.

[29] Donald R. Morrison. PATRICIA—practical algorithm to retrieve information coded in alphanumeric. *J. Assoc. Comput. Mach.*, 15(4):514–534, 1968.

[30] J.I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001.

[31] Sebastiano Vigna. Broadword implementation of rank/select queries. In Catherine C. McGeoch, editor, *Experimental Algorithms. 7th International Workshop, WEA 2008*, number 5038 in Lecture Notes in Computer Science, pages 154–168. Springer–Verlag, 2008.