Engineering Compressed Static Functions

Marco Genuzio Sebastiano Vigna
Università degli Studi di Milano

Abstract

Recent advances in the compact representation of static functions (with constant access time) have made it possible to fully exploit constructions based on random linear system. Such constructions, albeit theoretically appealing, were previously too slow to be usable. In this paper, we extend such techniques to the problem of storing *compressed* static functions, in the sense that the space used per key should be close to the entropy of the list of values. From a theoretical viewpoint, we are inspired by the approach of Hreinsson, Krøyer and Pagh. Values are represented using a near-optimal instantaneous code. Then, a bit array is created so that by XOR'ing its content at a fixed number of positions depending on the key one obtains the value, represented by its associated codeword. In the construction phase, every bit of the array is associated with an equation on $\mathbf{Z}/2\mathbf{Z}$, and solving the associated system provides the desired representation. Thus, we pass from one equation per key (the non-compressed case) to one equation per *bit*: the size of the system is thus approximately multiplied by the empirical entropy of the values, making the problem much more challenging. We show that by carefully engineering the value representation we can obtain a practical data structure. For example, we can store a function with geometrically distributed output in just 2.28 bits per key, independently of the key set, with a construction time double with respect to that of a state-of-the-art non-compressed function, which requires $\approx \log\log n$ bits per key, where n is the number of keys, and slightly improved lookup time. We can also store a function with an output of 10^6 values distributed following a power law of exponent 2 in just 2.75 bits per key, whereas a non-compressed function would require more than 20, with a threefold increase in construction time and significantly faster lookups.

1 Introduction

Static functions are data structures designed to store arbitrary mappings $f : X \to \Sigma$ from a finite set of keys $X \subseteq U$ to an alphabet Σ, where U is a universe of possible keys. Given $x \in U$, a static function returns a value that is $f(x)$ when $x \in X$; on $U \setminus X$, the result is irrelevant. In general, one looks for representations returning their output in constant time.

Functions can be easily implemented using hash tables, but since they are allowed to return *any* value if the queried key is not in X, in the static case we can break the information-theoretical lower bound of storing the set X. In fact, constructions for static functions achieve just $O(|X| \log |\Sigma|)$ bits of space, regardless of the nature of X and U. This makes static functions a powerful techniques when handling, for instance, large sets of strings, and they are important building blocks of space-efficient data structures such as (compressed) full-text indexes [1], (monotone) MPHFs [2, 3], Bloom filter-like data structures [4], and prefix-search data structures [5].

Recently [6], it has been shown how to make the construction of very compact representations of static functions scalable. Such representations are based on the satisfiability of a random k-XORSAT instance (i.e., the solvability of a random system on $\mathbf{Z}/2\mathbf{Z}$) [7, 8]: in particular, the authors introduced *lazy Gaussian elimination* and a system representation based on *broadword programming* [9], applied on top of the by now classic HEM "chunking" construction [10]. The

purpose of all these techniques is to reduce drastically the size of the random systems that one needs to solve.

A further improvement is *compression*: for example, the representation of static function in a space per key *proportional to the empirical entropy of the list of output values*. The standard constructions assume that each output fits b bits, and use $O(|X|b)$ bits, with a very small constant factor. However, if the distribution of the values is skewed a structure using a space per key that is proportional to the entropy could provide major space gains.

As we discuss in Section 2, there are a few theoretical proposals for the construction of compressed static functions. We focus on the one introduced by Hreinsson, Krøyer and Pagh [11]: their construction achieve the goal above, at the price of multiplying the number of equations that must be solved approximately by the entropy itself. Thus, it is not obvious that previously known techniques are sufficient to make this construction scalable, and, indeed, to the best of our knowledge no one has ever engineered, implemented and published practical code for building compressed static function for a large number of keys, whereas several constructions for the non-compressed case are available. In this paper we show that it is actually possible to engineer the construction of [11] so to have a moderate impact on construction time, and very fast query time—in fact, equivalent to that of a non-compressed function, and improved in case of a significant space reduction. We use von Neumann's definition and notation for the set of natural numbers, so $n = \{0, 1, \ldots, n-1\}$. All implementations discussed in this paper are distributed as free software as part of the Sux4J project (`http://sux4j.di.unimi.it/`).

2 Background and related work

In their seminal paper [12], Majewski, Wormald, Havas and Czech (MWHC hereinafter) introduced the first static function construction with compact space. To store a function $f : X \to 2^b$ with b-bit values, they generate a random system with equations of the form

$$w_{h_0(x)} \oplus w_{h_1(x)} \oplus \cdots \oplus w_{h_{k-1}(x)} = f(x)$$

for each $x \in X$, where the h_i's are fully random hash functions. Due to bounds on the acyclicity of random graphs, if the ratio between the number of variables and the number of equations is above a certain threshold γ_k (e.g., $\gamma_3 \approx 1.23$), the system can be almost always triangulated in linear time, and thus solved. Storing the solution makes it clearly possible to compute $f(x)$ in constant time. The space usage is approximately $\gamma_k b$ bits per key.

HEM. Botelho, Pagh and Ziviani [10] introduced a practical external-memory algorithm called Heuristic External Memory (HEM) to construct *minimal perfect hash functions* for large sets. They replace each key with a *signature* of $\Theta(\log n)$ bits computed with a random hash function, and check that no collision occurs. The signatures are then sorted and divided into small chunks based on their most significant bits: a separate function is computed for each chunk. The bit representations of the chunk functions are then concatenated into a single bit array and their offsets (i.e., for each chunk, the position of the start of the chunk in the global array) are stored separately. The same strategy can be applied to static functions.

Cache-oblivious constructions. As an alternative to HEM, in [13] the authors propose *cache-oblivious* algorithms that use only scanning and sorting to peel hypergraphs and compute the corresponding structures. The main advantage is that of avoiding the cost of accessing the offset array of HEM without sacrificing scalability.

Beyond hypergraphs. The MWHC construction for static functions can be improved: Dietzfelbinger and Pagh [14] introduced a new construction that allows to make the constant of the space bound for static functions *arbitrarily small*; by known bounds on k-XORSAT instances,[1] the system generated by the MWHC technique is solvable when the ratio between variables and equations is above a threshold δ_k that goes to one as k increases. For example, if $r = 3$, $\delta_3 \approx 1.089 < \gamma_3 \approx 1.23$. Unlike MWHC's linear-time algorithm, general matrix inversion requires super*quadratic* time ($O(n^3)$ with Gaussian elimination). Two of the authors, in collaboration with Giuseppe Ottaviano, have devised techniques to make this construction scalable [6].

Compression. The next natural step is some sort of *output-dependent compression*. A trivial approach is that of building a minimal perfect hash function $p : X \to n$, a Huffman code for the output values, concatenate the codewords representing $f(x)$ in the order induced by $p(x)$ and store the codeword boundaries using a succinct data structure. In this case, the space used for the concatenated Huffman codes is close to the empirical entropy, but the ancillary data structures cause a constant overhead which becomes very significant when entropy is small (which is the interesting case). Moreover, it is necessary to store a mapping from the codewords to the associated elements of Σ. Subsequent theoretical works have worked around some of these limitations [15, 11, 4].

3 Engineering a compressed static function

First of all, we recall the formal statement of the problem. We have a universe U, a *key set* $X = \{x_0, x_1, \ldots, x_{n-1}\} \subseteq U$ and a function $f : X \to \Sigma$. Our goal is to build a data structure that given an element of U returns in constant time an element of Σ, with the condition that the element returned for every $x \in X$ is $f(x)$. The value returned on $U \setminus X$ is *irrelevant*.

3.1 Theoretical bounds We now recall the main ideas behind the construction described in [11], which we chose as a basis of our engineering effort because of its simplicity. The *empirical 0-th order entropy* $\mathscr{H}_0$ of a sequence $v_0, v_1, \ldots, v_{n-1} \in \Sigma$ is defined as

$$\mathscr{H}_0 = \sum_{\sigma \in \Sigma} -p_\sigma \log p_\sigma,$$

where p_σ is the frequency of σ in the sequence $v_0, v_1, \ldots, v_{n-1}$. In particular, in the following given a function f as above we will denote with $\mathscr{H}_0$ the empirical entropy of the sequence of the output values $f(x_0), f(x_1), \ldots, f(x_{n-1})$.

Let $c(\cdot) : \Sigma \to \{0,1\}^*$ be a binary prefix-free encoding of Σ. We fix an integer k (in our implementations, $k = 3, 4$). The core of the construction is the creation of a bit array A and of k hash functions $h_{(\cdot)} : U \to |A|$ with the following property: given a key $x \in X$, if we XOR the content of A starting at positions $h_0(x), h_1(x), \ldots, h_{k-1}(x)$ (possibly wrapping up) we obtain a stream of bits which starts with $c(f(x))$. More precisely, the $|c(f(x))|$ bits given by

$$\bigoplus_{0 \le i < k} A_{h_i(x)+j}$$

for $0 \le j < |c(f(x))|$ are exactly $c(f(x))$.[2] Assuming that the longest word the in prefix-free code is a constant multiple of the machine word, the sequence of bits can be reconstructed and decoded

[1] The paper actually uses Calkin's bound for full rank matrices, which is not optimal.

[2] We assume that all indices of A are computed modulo $|A|$.

in constant time. This assumption can be made to hold at the expense of space occupation by limiting the depth of the tree associated with the prefix-free code.

Which constraints does the condition above put on the bit array A? If we call w_i a variable associated with the bit i of A, it is easy to see that for every $x \in X$ the i-th bit of $c(f(x))$ gives rise to the equation

$$w_{h_0(x)+i} \oplus w_{h_1(x)+i} \oplus \cdots \oplus w_{h_{k-1}(x)+i} = c(f(x))_i.$$

Thus, we obtain a system of $\sum_{x\in X} |c(f(x))|$ equations on $\mathbf{Z}/2\mathbf{Z}$, each with exactly k variables. Since the system is random, by the known bounds on the satisfiability of k-XORSAT instances [7, 8], which are equivalent to such linear system on $\mathbf{Z}/2\mathbf{Z}$, we know that we need A to be $\delta_k \sum_{x\in X} |c(f(x))|$ bits long, where δ_k is a constant that tends to 1 as k grows ($\delta_3 \approx 1.089$, $\delta_4 \approx 1.024$).

We know notice that if the distribution associated with the prefix-free code matches exactly the distribution of the values, $\sum_{x\in X} |c(f(x))|/n$ is exactly $\mathscr{H}_0$, which suggests that in general the space used per key will be very close to $\delta_k\mathscr{H}_0$, that is, very close to the entropy. In practice, as we will see, there will be some ancillary data which must be stored, too, and we will not be able to work with a truly optimal code, but both limitations have a relatively small impact on the whole data structure.

Note that in this process we shifted from n equations (for the non-compressed case) to $\sum_{x\in X} |c(f(x))|$ equations—an $\mathscr{H}_0$-fold increase in the size of the linear system, to which an even larger increase in construction time follows, as Gaussian elimination is cubic.

3.2 First phase: HEM and frequencies We now describe in sequence the various operations we perform on the data. We will combine several previous techniques, sometimes with overlaps so a step-by-step procedural description is easier to grasp (and to implement) than a compartmentalized one isolating each technique.

In the first pass, we scan the input data and perform two tasks: we build a frequency table for the output values, and we apply HEM [10]. HEM replaces the original pairs $\langle x, f(x)\rangle$, $x \in X$, with a simpler set of pairs $\langle s(x), f(x)\rangle$ where $s(x)$ is a digital fixed-length signature. The signature length should be chosen so to make unlikely a signature collision (we use 192 bits extracted from the 256-bit state of the "short" SpookyHash[3] of the keys). The pairs $\langle s(x), f(x)\rangle$ are then sorted by $s(x)$, and if collisions are found the construction is repeated with a different signature. Thus, at the end of the first pass we have on disk a sorted list of pairs $\langle s(x), f(x)\rangle$ and (in core memory) the frequencies p_σ, $\sigma \in \Sigma$.

3.3 Second phase: the prefix-free code Now, using the frequencies we can in principle generate an optimal prefix-free code for the distribution of the output values. Such a code, however, poses two problems: first, it is in general impossible to decode a codeword in constant time; second, for skewed distributions the longest generated codeword can be quite long.

The solution proposed in [11] is the following: a subset of most frequent elements of Σ is actually stored using an optimal length-limited code, together with an additional symbol $\perp$ accounting for the rest of the elements. Then, the non-frequent elements are stored explicitly after $\perp$ using a suitable table. Essentially, one trades the optimality for a small cost in space and faster decoding time.

[3] `http://burtleburtle.net/bob/hash/spooky.html`

After noticing that the decoding process had some effect on the lookup time, we decided to use a different practical approach based on canonical Huffman codes [16]. Canonical Huffman codes are based on the observation that among many equivalent Huffman decoding trees, there is one that is canonical: the one in which the depth of leaves from left to right is nondecreasing. The structure of such a tree (and of the associated code) can be described by two lists of integers of the same length: the first list enumerates the codeword lengths, in increasing order. The second list determines, for each length, the cardinality of the codewords with that length. Codewords of the same length are consecutive integers.

It is possible to decode a canonical code in time that is proportional to the length of the lists, rather then to the length of the keywords [17]. More interestingly, it is trivial to modify the structure of a canonical Huffman code so to obtain a limited-length code. Once we fix a length L, it is sufficient to reassign equal frequencies p_σ to all values σ whose current codeword has length greater than L, and rebuild the canonical Huffman code. For a fixed L, decoding now happens in constant time, as the new lists have length at most $L + 2$. The code is no longer optimal, but for the distributions we are interested in the loss in compression is very limited, even for fairly low values of L. Finally, we remark that since we are able from read from the bit array whole codewords, it is possible to arrange the decoding table so that the decoding loop for the canonical Huffman code reduces to a linear search in an array.

3.4 Third phase: generating systems From now on, we fix a *chunk log size* ℓ. Our target is now to work with sets of keys of size approximately 2^ℓ. To do so, we will scan the sorted pairs $\langle s(x), f(x)\rangle$ and divide them in *chunks* using the highest ℓ bits of the signature $s(x)$ (as it is standard when applying HEM). Clearly, the average size of a chunk will be 2^ℓ. We will apply the construction of [11] *independently for each chunk*, but we will be using for all chunks the prefix-free code generated in the second phase. We will use a *local seed* for each chunk, so to be able to generate different systems until we find a solvable one (which must happen soon, as after the k-XORSAT threshold systems are asymptotically solvable with high probability). Note that here the fact that we are using HEM pairs $\langle s(x), f(x)\rangle$ provides a major speed improvement, since we have simply to combine $s(x)$ with the local seed and shuffle the resulting fixed-length bit vector. There is no need to rehash the original keys (in practice, we set the 256-bit internal state of the short SpookHash algorithm with the 192-bit signature and the local seed and perform a round of mixing).

If we focus on a single chunk the code we are using is no longer optimal, but this does not change the probability of solving systems, as the system size is dependent on the sum of the codewords representing values actually in the chunk, and does not depend on the optimality of the code. Concatenating the bit arrays of all chunks we get exactly the same space we would have used without HEM. We have however to store the cumulative distribution of chunk lengths, so to be able to retrieve the part of the global bit array associated with a chunk, and the local seed. Note that the approach of having a local prefix-free optimal code per chunk would never work, as the space used to store the local codes would be preposterous.

We remark that the authors of [11] advocate a special form for the hash functions $h_{(\cdot)}$:

$$h_i(x) = h'_i(x)w + q(x), \tag{3.1}$$

where w is the length of the longest codeword, and $h_{(\cdot)} : U \to \lceil m/w \rceil$ and $q : X \to w$ are fully

random hash functions (with some additional mild conditions—see [11]). By structuring the functions in this way, the systems associated with the locations of A that are not congruent modulo w are independent, creating smaller systems and making it possible a parallel solution.

We found that in practice this approach is detrimental. Moreover, HEM can be easily parallelized without effort (each chunk can be analyzed independently, as indeed happens in Sux4J), so it is not really necessary to exploit parallelism. The problem, which is hard to detect from the theoretical side, is that the k-XORSAT bounds we are using have a different asymptotic behavior than, say, acyclicity thresholds for random graphs. The asymptotic behavior starts to become typical at much larger values (in the acyclicity case, just a few dozens vertices are sufficient). Since we are solving systems chunk by chunk, but we are using a global code, some chunks happen to have a quite large maximum codeword w, in particular in the case of skewed distributions. For these chunks, using (3.1) scales the size of the systems and the number of variables down by a factor of w, bringing them out of the "good" region.

3.5 Fourth phase: solving systems We represent our systems in a compact form using bit arrays in which a bit set represents a variable in the equation. All operations can be then performed using word-by-word XORs. First, we try to triangulate greedily the system by removing equations containing a variable appearing exactly once (this is equivalent to the *peeling* procedure of the MWHC construction). Then, we apply *lazy Gaussian elimination* [6], a refined, parameterless version of *structured Gaussian elimination* [18, 19]. In our case, lazy Gaussian elimination reduces the size of the system to be solved by standard elimination to around 4% of the original one.

If the system is not solvable we generate a new one after incrementing the local seed.

3.6 Fifth phase: storing the data structure Finally, we store the following data:

- the decoding table of the length-limited canonical code (a few dozen integers);
- the mapping from codewords to values in Σ;
- the global bit array ($\approx \delta \mathscr{H}_0 n$ bits, $\delta \geq \delta_k$);
- for each chunk, a 64-bit integer containing in compacted form the local seed of the chunk and the number of keys in previous chunks (i.e., the cumulative distribution of chunk sizes).

4 Parameters

The construction we just described requires fixing a number of parameters, such as the chunk log size,etc. The interplay between these parameters is quite contrived, due to their multiple effects. More in detail:

k Increasing the number of hash functions decreases the space usage, due to a lower δ_k, but increases lookup time, due to additional memory accesses, and increases construction time, due to denser linear systems.

Chunk log size The chunk log size ℓ has several effects. Larger chunks means a lower impact of the ancillary data (cumulative chunk size distribution and local seeds) and a larger fraction of solvable systems (systems gets larger, and the asymptotic "kicks in"), but also a much slower computation time (same reason). Practically, the possible range of value for ℓ is $[9 \ldots 13]$.

Dataset	Size	Geometric	Zipfian	Uniform	Indegree
Hosts	11 264 052 (233 MiB)	[0 . . 23], 2.0	[1 . . 774599], 2.36	[0 . . 63], 6.0	[1 . . 174433], 4.22
URLs	1 070 557 254 (79 GiB)	[0 . . 31], 2.0	[1 . . 997570], 2.36	[0 . . 63], 6.0	[1 . . 20252239], 5.10

Table 1: Actual ranges and empirical entropy of our synthetic and real-world datasets.

δ One can use a value δ slightly larger than δ_k as a multiplier for $\sum_{x\in X} |c(f(x))|$. This has the effect of increasing the fraction of solvable systems, and thus speeding up construction time, at the expense of a few percent of additional space.

Maximum Huffman code length This is a somewhat easier choice: by setting, say, the maximum length of an optimal codeword to 20 we have the guarantee that the loss of space with respect to an optimal code will be negligible.

5 Experiments

In this section we present the results of our experiments, which were performed on an Intel® Core™ i7-7770 CPU @3.60GHz (Kaby Lake), with 64 GiB of RAM, Linux 4.10.12 and Java 8. For all the timings we report, the relative standard deviation is very low (below 5%) and due mostly to I/O unpredictability and Java garbage collection.

Besides establishing whether the constructions proposed are practical, we performed a grid analysis to find the best combination of parameters. Only k has an effect on the lookup speed, so our interest is to find a good compromise between space usage and construction time. The experiments we report use $\ell = 10$ (i.e., chunks of size ≈ 1024), $\delta = 1.10$ for $k = 3$ and $\delta = 1.03$ for $k = 4$. Due to the considerations of Section 3.4, these turn out to be the best choices. By choosing a few percent larger δ values (and obtaining correspondingly larger space) it is possible to reduce by about 50% the construction time. Increasing the chunk size reduces slightly the cost of ancillary data per key, but the necessary increase in δ to maintain reasonable a construction time turns out to increase the overall space usage.

5.1 Datasets We used two key sets derived from the public `eu-2015` crawl[4] gathered by the LAW (Laboratory for Web Algorithmic) of the Università degli Studi di Milano using the crawler BUbiNG [20]. In particular, we consider a list of 11 264 052 host names, and a list of 1 070 557 254 URLs. For each key set we generate three synthetic lists of values using different distributions: a geometric distribution with probability $p = 1/2$; a Zipfian (i.e., finite power-law) distribution with 10^6 values and exponent 2; a discrete uniform distribution with 64 values. Moreover, we consider the mapping from each URL (host) to its indegree in the web (host, respectively) graph. The geometric and uniform distributions are the most skewed and the less skewed distribution with exactly matching optimal codes, and the Zipfian distribution sits somewhere in the middle. Note that in principle the geometric distribution has unlimited range, but in practice for n keys the largest generated value is $O(\log n)$ with high probability. Table 1 reports the empirical value range and empirical entropy of the eight resulting combinations.

[4] `http://law.di.unimi.it/webdata/eu-2015/`

We compared our construction with the state-of-the-art (non-compressed) implementation provided by the classes `GOV3Function` and `GOV4Function` of Sux4J, which are based on the same k-XORSAT bounds [6].

5.2 Discussion In Table 2 and 3 we report the results on our eight combinations of datasets and output values, coupled with similar results for the non-compressed data structures distributed with Sux4J. The number of bits per key is measured on the serialized data structure, and contains some Java serialization overhead (below 1%). Time is measured by wall clock. Note that the URL dataset is much bigger (per key): additional ≈ 600 ns per key are necessary just to read the input. If the distribution is skewed and the entropy is small, we obtain a significant compression, very close to $\delta_k \mathscr{H}_0$ bits per key, as expected. The compression is actually more impressive in the case of a distribution with a long tail, as in that case for large datasets a few very large values occur, which forces the standard implementation to choose a large number of bits per value. In the case of the geometric distribution this might happen, too, but with very small probability. In the uniform case, construction time increases significantly, and of course we have no space gain. Nonetheless, since we are able to use a value of δ very close to the best possible, we do not increase the space usage. Lookup time is slightly slower due to decoding, but by using a flat binary code it would be equivalent to the standard case. Lookup times for Zipfian distributions are (sometimes significantly) smaller. This is due to the reduced stress on the cache and on the Translation Lookahead Buffer, which limit the speed at which the CPU can access memory. With respect to the case $k = 3$, the case $k = 4$ sports slightly increased access time, a few percent reduction in space (as expected) and much longer construction time, due to the increased density of the linear systems. Decoding the canonical Huffman code has a very marginal impact, as it requires ≈ 10 ns or less (≈ 2 ns in the uniform case, as the decoding table contains just one entry).
Switching to triangulation. Since we try to triangulate the system before solving it by lazy Gaussian elimination, for $k = 3$ when $\delta = 1.23 \approx \gamma_3$ we can always triangulate the system in linear time (as in the MWHC case). In practice, with a 12% increase in space we can *guarantee* that the increase in construction time will be at most multiplied by the entropy. In fact, as we show in Table 4, the increase is usually much less, as the triangulation process is responsible for a small fraction of the overall construction time. Maybe surprisingly, because of the reduced memory footprint the construction time for the geometric case is *faster* than the standard construction. Note that γ_3 is *minimum* among the γ_k, so there is no sense in trying this approach for $k \neq 3$.

6 Conclusions and future work

We presented a detailed, engineered implementation of the theoretical construction proposed in [11] for compressed static functions. By carefully tuning each part of the construction process, we have been able in most cases to contain the increase in construction time. We obtain this result by a combination of HEM-based key split, lazy Gaussian elimination, parameter optimization and limited-length canonical codes. Note that to circumvent the high-entropy case one can measure at construction time the entropy of the output values and just switch to the construction based on linear-time system triangulation. In any case, the implementations we distribute within Sux4J uses multicore parallelization to further decrease the construction time (the single-thread

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

Table 2: Comparison between the non-compressed implementations from Sux4J (Std) and our compressed implementation (Comp) for the host dataset.

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

Table 3: Comparison between the non-compressed implementations from Sux4J (Std) and our compressed implementation (Comp) for the URL dataset.

evaluation we used in this paper provides more consistent results across different techniques).

If the distribution of values is known in advance, Huffman codes can in principle be replaced by a standard instantaneous code such as unary or Elias's γ.

We remark that one of the great advantages of the theoretical approach we have chosen for our algorithmic engineering efforts [11] is that the number of memory accesses to compute a value is identical to that of the non-compressed case we are comparing with: indeed, in our tests we found a minor lookup slowdown in the uniform case, and a speedup in some skewed cases, in spite of the time that is necessary for decoding, as the smaller memory footprint reduces the out-of-cache accesses and the stress on the Translation Lookaside Buffer.

References

[1] Djamal Belazzougui and Gonzalo Navarro, "Alphabet-independent compressed text indexing," *TALG*, vol. 10, no. 4, 2014.

[2] D. Belazzougui, P. Boldi Pagh, and S. Vigna, "Monotone minimal perfect hashing: Searching a sorted table with $O(1)$ accesses," in *SODA*. 2009, pp. 785–794, ACM Press.

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

Table 4: Data for $\delta = 1.23$, using triangulation to solve linear systems.

[3] D. Belazzougui, P. Boldi, R. Pagh, and S. Vigna, "Theory and practice of monotone minimal perfect hashing," *ACM Journal of Experimental Algorithmics*, vol. 16, no. 3, pp. 3.2:1–3.2:26, 2011.

[4] D. Belazzougui and R. Venturini, "Compressed static functions with applications," in *SODA*. 2013, pp. 229–240, ACM Press.

[5] D. Belazzougui, P. Boldi, R. Pagh, and S. Vigna, "Fast prefix search in little space, with applications," in *ESA 2010*. 2010, vol. 6346 of *LNCS*, pp. 427–438, Springer.

[6] M. Genuzio, G. Ottaviano, and S. Vigna, "Fast scalable construction of (minimal perfect hash) functions," in *SEA 2016*, 2016, pp. 339–352.

[7] O. Dubois and J. Mandler, "The 3-xorsat threshold," *Comptes Rendus Mathematique*, vol. 335, no. 11, pp. 963–966, 2002.

[8] M. Dietzfelbinger, A. Goerdt, M. Mitzenmacher, A. Montanari, R. Pagh, and M. Rink, "Tight thresholds for cuckoo hashing via XORSAT," in *Automata, Languages and Programming*, vol. 6198 of *LNCS*, pp. 213–225. Springer Berlin Heidelberg, 2010.

[9] D. E. Knuth, "The Art of Computer Programming. Pre-Fascicle 1A. Draft of Section 7.1.3: Bitwise Tricks and Techniques," 2007.

[10] F. C. Botelho, R. Pagh, and N. Ziviani, "Practical perfect hashing in nearly optimal space," *Inf. Syst.*, vol. 38, no. 1, pp. 108–131, 2013.

[11] J. B. Hreinsson, M. Krøyer, and R. Pagh, "Storing a compressed function with constant time access," in *ESA 2009*, 2009, pp. 730–741.

[12] B. S. Majewski, G. Havas N. C. Wormald, and Z. J. Czech, "A family of perfect hashing methods," *Comput. J.*, vol. 39, no. 6, pp. 547–554, 1996.

[13] D. Belazzougui, P. Boldi, R. Venturini G. Ottaviano, and S. Vigna, "Cache-oblivious peeling of random hypergraphs," in *DCC 2014*. 2014, pp. 352–361, IEEE.

[14] M. Dietzfelbinger and R. Pagh, "Succinct data structures for retrieval and approximate membership (extended abstract)," in *ICALP*. 2008, vol. 5125 of *LNCS*, pp. 385–396, Springer.

[15] E. Porat, "An optimal bloom filter replacement based on matrix solving," in *CSR*. 2009, vol. 5675 of *LNCS*, pp. 263–273, Springer.

[16] E. S. Schwartz and B. Kallick, "Generating a canonical prefix encoding," *Commun. ACM*, vol. 7, no. 3, pp. 166–169, Mar. 1964.

[17] D. S. Hirschberg and D. A. Lelewer, "Efficient decoding of prefix codes," *Comm. ACM*, vol. 33, no. 4, pp. 449–459, Apr. 1990.

[18] A. M. Odlyzko, "Discrete logarithms in finite fields and their cryptographic significance," in *Advances in cryptology*. 1985, pp. 224–314, Springer.

[19] B. A. LaMacchia and A. M. Odlyzko, "Solving large sparse linear systems over finite fields," in *Advances in Cryptology: CRYPT0'90*, pp. 109–133. Springer, 1991.

[20] P. Boldi, A. Marino, M. Santini, and S. Vigna, "BUbiNG: Massive crawling for the masses," in *WWW 2014 Companion*, 2014, pp. 227–228.