Storing a Compressed Function with Constant Time Access

Jóhannes B. Hreinsson, Morten Krøyer, and Rasmus Pagh*

IT University of Copenhagen, 2300 København S, Denmark
{johre06,kroyer,pagh}@itu.dk

Abstract. We consider the problem of representing, in a space-efficient way, a function $f : S \to \Sigma$ such that any function value can be computed in constant time on a RAM. Specifically, our aim is to achieve space usage close to the 0th order entropy of the sequence of function values. Our technique works for any set S of machine words, without storing S, which is crucial for applications.
Our contribution consists of two new techniques, of independent interest, that we use in combination with an existing result of Dietzfelbinger and Pagh (ICALP 2008). First of all, we introduce a way to support more space efficient approximate membership queries (Bloom filter functionality) with arbitrary false positive rate. Second, we present a variation of Huffman coding using approximate membership, providing an alternative that improves the classical bounds of Gallager (IEEE Trans. Information Theory, 1978) in some cases. The end result is an entropy-compressed function supporting constant time random access to values associated with a given set S. This improves both space and time compared to a recent result by Talbot and Talbot (ANALCO 2008).

1 Introduction

Compression is an important technique in modern computing systems. Often, some kind of random access is required, such that a given data item can be decompressed without decompressing all data. The standard way of dealing with this requirement is to split the data into blocks that are compressed and decompressed individually. The end result is a trade-off between compression and access time: Larger blocks lead to better compression, but slows down decompression. A recent theoretical breakthrough of Pătraşcu [17] shows how to combine optimal (in a certain sense) compression of an array with logarithmic decompression time. In this paper we are concerned with compression of *functions* that allows random access. Like most other compression methods we deal with the static case, and do not consider efficient updates.

The problem of storing a function f with certain specified values (referred to as the *retrieval* problem) has recently received renewed interest [7, 9, 18]. The

* This work was supported by the Danish National Research Foundation, as part of the project "Scalable Query Evaluation in Relational Database Systems."

main finding is that there exist simple methods that store f in space close to the space that would be needed to store the (uncompressed) function values, and provide constant access time. For example, if f maps city names to weather, and there are two kinds of weather, f can be stored in close to 1 bit per value, with fast random access to the value for a particular city name. This is of course much less than the space usage required for also storing the set of city names. The price paid compared to standard dictionary representations is that the data structure does not detect when it is used on an input where no function value has been specified—in this case it will simply return some arbitrary value.

The retrieval problem is relevant in situations where the amount of data associated with each key is small, and it is either known that queries will only be asked on keys in a fixed set S, or where the query answers for keys not in S are insignificant. For example, suppose that we have ranked a collection of web pages. Then a retrieval data structure would be able to return the ranking of a given URL, without storing the URL itself. This might allow the ranking information to be stored entirely in RAM (e.g., a browser could show highly ranked links more prominently). Several applications of retrieval structures as building blocks in other data structures are described in [1, 2].

Our starting point is that data values are often skewed (e.g., in some parts of the world it is sunny much more often than it is rainy), and we may like to extend the set of possible values (e.g., add more rare weather phenomena such as thunderstorms) without increasing the number of bits required to represent each function value. Therefore, it is desirable to have representations where the space usage is dependent on the *entropy* of the data values rather than on the number of possible values.

1.1 Our results

Let $S = \{x_1, \dots, x_n\}$ denote the domain of the function, and let Σ be the set of possible (or actual) function values. For simplicity we assume that $\Sigma = \{1, \dots, \sigma\}$ — the general case can be handled by a separate data structure implementing a bijection between Σ and $\{1, \dots, \sigma\}$ (e.g. a minimal perfect hash function [19, 12, 4] for Σ plus an array of size σ). We describe a new data structure that represents a function $f : S \to \Sigma$ in space that is close to the (empirical, 0th order) entropy H_0 of the sequence $f(x_1), \dots, f(x_n)$. If $p_1, p_2, \dots$ are the frequencies of different characters in the sequence, $H_0 = \sum_i p_i \log_2(1/p_i)$ is a lower bound on the number of bits needed per function value, assuming that the function values are independent of the corresponding input values.

We present our results in the Word RAM model of computation [11] with word size w. To simplify the presentation we assume that elements of S as well as values in Σ can be represented in a single word, and specifically that $w \geq 2 + \log \sigma$. We show the following:

Theorem 1. *Let n, w, and σ be positive integers, where $w^3 < n < 2^w$ and $\sigma \leq 2^w$, let $\delta > 0$ be a constant, and let $S = \{x_1, \dots, x_n\} \subseteq \{0, 1\}^w$ be a set of size n. Given a function $f : S \to \{1, \dots, \sigma\}$, let H_0 denote the empirical (0th*

order) entropy of the sequence $f(x_1), \ldots, f(x_n)$, and let p_1 denote the frequency of the most common function value. If n is larger than some constant (depending on δ) there exists a retrieval data structure for f using space

$$(1 + \delta)H_0 + \min(p_1 + 0.086, 1.82(1 - p_1)) \text{ bits}$$

per function value, plus $o(\sigma)$ bits to store information about the distribution, such that a function value can be computed in $O(1)$ time on a Word RAM with word size w.

Discussion. The term $o(\sigma)$ is negligible in most cases that are interesting from a compression point of view, i.e., in cases where the number σ of possible values is not much larger than the number of values n. Ignoring this term, the number of bits per character is at most H_0 (a lower bound) times $1 + \delta$, plus a small additive term. This is similar to the space that would be obtained by Huffman coding the sequence of function values (with no random access). If p_1 is close to 1 the space usage becomes much better than using Huffman coding — in fact, the space per value can get arbitrarily close to 0, while Huffman coding uses at least 1 bit per value. An example where this is important is storage of functions with many undefined/NULL values.

While the algorithm used to construct our data structure is somewhat complex, the algorithm for evaluating f is extremely simple. It consists of looking up $O(1)$ w-bit strings, performing a bitwise exclusive or, and applying a constant time decoding procedure similar to Huffman decoding. This is illustrated in Figure 1. Thus, we show how to extend the simplicity of existing retrieval data structures (with no compression) to the compressed case.

Techniques. Technically, we first show how to extend existing retrieval data structures to support variable length bit strings as values. The method works under the condition that the set of possible values is prefix-free, and involves a nontrivial load balancing idea using slightly correlated hash functions. Combining this with a variation of (length-limited) Huffman coding, and showing how the decoding can be done in constant time, yields a result that is close to Theorem 1, but missing the second part of the minimum in the additive term. To strengthen the result in the important case where the majority of function values are identical we show how to use an *approximate membership* data structure (i.e., a data structure with the same functionality as a Bloom filter [3]) to decrease the space usage.

To reduce the redundancy as much as possible, we describe a general reduction that can be used to obtain space-efficient approximate membership data structures for any false positive rate $\varepsilon > 0$. Previous space-efficient methods (see [9] for an overview) provided false positive rates that are negative powers of two. This means that one needs to "pay" (with extra space usage) for a false positive rate that is up to 50% smaller than desired – our method reduces this to less than 6%.

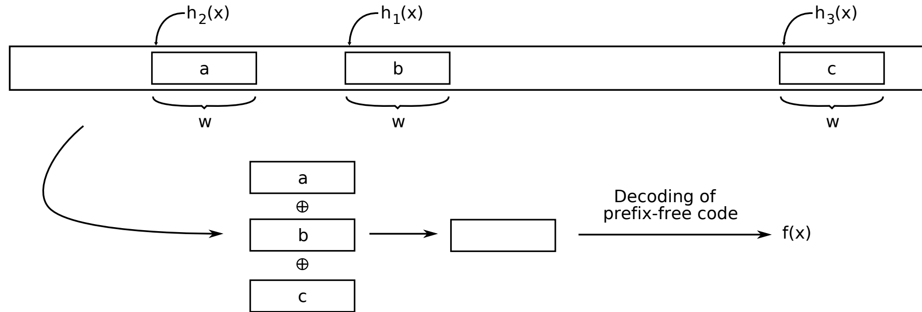

Fig. 1. Illustration of how a value $f(x)$ is computed from our data structure.

1.2 Related work

Several previous papers described data structures having two independent functionalities: They support retrieval queries (given $x \in S$, return $f(x)$), and approximate membership queries (given x, return "true" if $x \in S$, and return "false" with probability at least $1 - \varepsilon$ if $x \notin S$). A data structure with this interface is sometimes referred to as a *bloomier filter* [8]. In this paper we consider only the retrieval problem, but note that approximate membership queries can be added (without loss) by a separate approximate membership data structure.

A faster approach to space-efficient retrieval is through space-efficient minimal perfect hashing [19, 12, 4]. For $\Sigma = \{0, 1\}^r$, where $r \leq w$ is a positive integer, this gives a space usage of $nr + O(n)$ bits and $O(1)$ query time. However, this approach is rather complicated, and it seems especially complicated to generalize it to variable-length function values (which would be required for compression). Also, even without compression the use of minimal perfect hashing is known to yield a redundancy of at least $\log_2 e \approx 1.44$ bits per element in S, which is more than we achieve.

Rather than building on minimal perfect hashing, we use as a starting point the recent retrieval structure described by Dietzfelbinger and Pagh [9], which gives a very simple query algorithm for function values in $\Sigma = \{0, 1\}^r$ (with no compression). Similar techniques have recently been studied in two other papers [9, 18]. The result of Porat [18] is that a space usage of $nr + o(n)$ bits is possible with $O(1)$ query time. While this is asymptotically superior to [9], it relies heavily on tabulation and does not seem to admit a similarly efficient generalization to the compressed case. We refer to the discussion in [9] for a history of related data structures.

The first paper to consider compressed retrieval was recently published by Talbot and Talbot [20]. The authors consider a relaxation of the retrieval problem, where a fraction ε of the function values are allowed to be *incorrect* ("misassignments"). Arguably, this makes their data structure unusable for some applications, but we still find it instructive to compare their result to ours (even assuming ε is close to 1). Talbot and Talbot show how to obtain two different trade-offs between compression and query time: Space $1.44H_0 + 1 + \log\log\sigma$ per

value is possible with query time $O(\log^2 \sigma)$, or alternatively space $2.88H_0 + 2$ per value is possible with query time $O(\log \sigma)$. It is also stated (without proof) that the multiplicative constants can be improved to 1.23 and 2.46, respectively. The technique used in [20] is essentially adaptive decoding of Huffman encoded values (one bit at a time). Our data structure performs $O(1)$ non-adaptive memory accesses, does not have misassignments, and improves time as well as space of both trade-offs. In addition, the query algorithm is considerably simpler, and more likely to be of practical use.

Recently, another retrieval data structure that is able to take advantage of skew in the value distributions, "two-step MWHC", was described in [2]. While this method has $O(1)$ query time, its space usage is larger than ours, and cannot be bounded in terms of the 0th order entropy.

1.3 Preliminaries

Each element in $S = \{x_1, x_2, \ldots, x_n\} \subseteq D$ maps to the value $f(x_i)$ from an the alphabet $\Sigma = \{1, \ldots, \sigma\}$. Let a_i denote the ith most frequent value in Σ, breaking ties arbitrarily. For uniformly random $x \in S$ we let $p_i = \Pr\{f(x) = a_i\}$, for $i = 1, 2, \ldots, \sigma$.

Since we build on the data structure of Dietzfelbinger and Pagh [9], we now briefly describe it. Suppose that $k \geq 3$ hash functions $h_1, \ldots, h_k$ are given such that for any $x_i \in S$ the set $\{h_1(x_i), \ldots, h_k(x_i)\}$ is a random set of size k (no collisions), and the sets associated with different x_i are independent[1]. The idea is to set up a vector A of r-bits values, such that $f(x_i)$ can be calculated as a bitwise exclusive-or of k vector entries.

$$f(x_i) = \bigoplus_{j \in \{1, \ldots, k\}} A_{h_j(x_i)}. \tag{1}$$

On inputs $x \in D \setminus S$, the query algorithm returns an arbitrary value. The existence of a suitable vector A depends on the hash function values of the keys in S. It is shown in [9], using results from [5], that for any $\delta > 0$ there exists a constant $k = O(\log(1/\delta))$, such that array size $m = (1+\delta)n$ suffices to ensure the existence of A with high probability, given that n exceeds some sufficiently large constant. Specifically, the probability of failure for $k \geq 4$ is $O(n^{-5/7})$ [9]. We observe that a failure is not a serious problem, since we may simply choose k new hash functions and try again until we succeed. Even a small value of k allows a very space-efficient data structure, e.g., the redundancy δ for $k = 3, 4, 5$ is around 12%, 3%, and 1%, respectively.

Computing the entries of vector A requires solving a system of linear equations over GF(2). At first glance this seems to require Gaussian elimination in time $O(n^3)$, but it is shown in [9] how to set up equation systems that are easier to solve, and require time $O(n^{1+\epsilon})$, for any $\epsilon > 0$, or even linear time (increasing

[1] The assumption that fully independent hash functions are available can be justified, in the sense that there is a simulation of full randomness that makes everything work. See [9] for details.

k by $O(1)$). The first technique is based on splitting the set S into buckets of size $n^{\epsilon/2}$, and this also applies to our setting in a straightforward fashion. This means we can achieve construction time $O(n^{1+\epsilon})$, for any $\epsilon > 0$. Since the details are very similar to [9] we do not further describe this.

Overview of paper. In section 2 we present an efficient data structure for the retrieval problem with variable length values from a prefix-free set of bit strings. Section 3 describes how to combine this with a variation of Huffman coding that can be decoded in constant time. Finally, in section 5 we show how to improve the result in the case where some function value is much more frequent than others, using a result on approximate membership described in section 4.

2 Retrieval with variable-length values

We will use a prefix-free code for values in Σ, so that we get a function $f : S \to \{0,1\}^*$, where the keys are mapped to codewords of various length. We assume that the longest codeword has length at most w, which will be the case in our application. Our data structure will represent a similar function, $\hat{f} : S \to \{0,1\}^w$, so that for each key $x_i \in S$, $f(x_i)$ is a prefix of $\hat{f}(x_i)$. Since the values of f are prefixes of the values of $\hat{f}$, and the code is prefix-free, we can use a code tree to determine the value of $f(x_i)$ from $\hat{f}(x_i)$, so in effect this gives a representation of f. Section 3 will describe how to do the decoding efficiently.

The data structure and query algorithm. Our data structure is simply a *bit* array A of length m, where $m = (1 + \delta/3) \sum_i |f(x_i)| + O(w)$, i.e., m is essentially a factor $1 + \delta/3$ larger than the total length of all strings in $f(S)$. We round up m to the nearest multiple of w, and implement "wrap-around" reads by duplicating the initial word of A after the last word. This means that when we look up w consecutive bits, the data structure behaves like a cyclic array of m bits.

Let $k \geq 3$ be an integer constant, and let $h_1, \ldots, h_k$ be hash functions mapping each key to k bit positions in A (details below). We arrange the array so that for any key $x \in S$, we can compute $\hat{f}(x)$ as the bitwise exclusive-or of the k words starting at the bit positions $h_1(x), \ldots, h_k(x)$. (Note that these words are not necessarily aligned with the Word RAM machine words.)

Hash functions and construction algorithm. Let $h'_1, \ldots, h'_k : S \to [m/w]$ be hash functions such that for any x, the set $\{h'_1(x), \ldots, h'_k(x)\}$ contains k (distinct) values, and is uniformly distributed over all such sets. (See [9] for a discussion on how to construct such hash functions in an efficient way.) Also, let $q : S \to [w]$ be a fully random hash function. We use the k hash functions defined by:

$$h_i(x) = h'_i(x)w + q(x) \quad . \tag{2}$$

That is, for any x the $\log w$ least significant bits of the hash function values $h_1(x), \ldots, h_k(x)$ are identical, while the $\lceil \log(m/w) \rceil$ most significant bits are distinct.

In order to construct A we must solve the following system of linear equations

$$f(x_i)_d = \bigoplus_{j=1}^{k} A_{h'_j(x_i)w+q(x_i)+d}, \text{ for} \qquad (3)$$
$$i = 1, \ldots, n,\ d = 1, \ldots, |f(x_i)|$$

where $f(x_i)_d$ is the dth bit in the codeword associated with the key x_i. We observe that all equations involve only bit positions in A that have the same residue modulo w. This means that there are in effect w systems of equations that may be solved individually. In each such system, an equation for $f(x)_d$ involves the k variables at positions $\{h'_1(x), \ldots, h'_k(x)\}$ within the system. This is exactly the setting of [9], which means that the same construction algorithm and analysis applies to each system.

2.1 Analysis

Let n_d denote the number of equations in the linear system corresponding to bits in positions with residue d modulo w. Note that $n_d = \sum_i X_{d,i}$, where $X_{d,i}$ is an indicator random variable that is 1 if and only if some equation involving x_i involves bits in position with residue d modulo w. Note that $X_{d,i}$ depends entirely on $q(x_i)$. For every d, the random variables $X_{d,i}$, $i = 1, \ldots, n$, are independent, so the sum n_d is tightly concentrated around the expectation of $\sum_i |f(x_i)|/w$. By Chernoff bounds (e.g. [16, Theorem 4.1]), and using that $n > w^3$, we have that $\Pr[n_d > (1 + \delta/2)n/w] < 1/(2w)$ when n is sufficiently large. This means that with probability at least $1/2$ (over the choice of q) the w equation systems are "well balanced" in the sense that $n_d \leq (1 + \delta/2)n/w$ for all d.

Conditioned on $n_d \leq (1 + \delta/2)n/w$ we can use Calkin's bounds [5, 9], which imply that each equation system is solvable with probability $1 - O((n/w)^{-5/7}) \geq 1 - o(1/w)$, where the inequality uses that $n \geq w^3$. By the union bound this means that with probability $1 - o(1)$ all equation systems are simultaneously solvable, and hence a suitable bit array A exists. In conclusion, the probability that the randomly chosen hash functions are suitable is bounded away from 0, so a constant number of trials suffices in expectation to find good hash functions for a given set S.

3 Constant time Huffman-like decoding

Without loss of generality assume that $\delta < 1$ in the statement of Theorem 1. We wish to apply the data structure of section 2 with a near-optimal prefix-free code. One possibility would be a Huffman code [13], but since we are willing to sacrifice a factor $1 + \delta/3$ in space usage it is possible to do better, both in terms of decoding time and in terms of the size of the representation of the code tree.

The first step is to identify the set of values Σ' for which the number of occurrences is above $n/\sigma^{1/(1+\delta/3)}$. Observe that $|\Sigma'| \leq \sigma^{1/(1+\delta/3)}$. Conceptually, we substitute all function values in $\Sigma \backslash \Sigma'$ with a new value $\perp$, and then consider

a code tree for the values of the resulting function f^*. Length-limited Huffman codes [14, 15] provide codewords of maximum length $\ell = \log|\Sigma'| + O(1)$, whose redundancy is within an additive constant of Huffman codes. In fact, the additive constant can be made arbitrarily small by increasing the $O(1)$ term. Decoding of length-limited Huffman codes can be done in constant time using a table indexed by all bit strings of length ℓ, where an entry contains the value in Σ corresponding to its prefix. The size of the table is $2^\ell \log\sigma = O(|\Sigma'|\log\sigma) = o(\sigma)$ bits.

Whenever the value $\perp$ is observed in the lookup table, we fall back on a trivial encoding of the appropriate symbol. We store the codeword for $\perp$, and use the next $\lceil\log\sigma\rceil$ bits to encode the value. Since the frequency of each symbol encoded in this way is at most $\sigma^{-1/(1+\delta/3)}$, the total length of the resulting n codewords is at most a factor $1/(1+\delta/3)$ from the length if an optimal code was used.

In conclusion, we have described a variation of Huffman coding that can be decoded in constant time, using a table of $o(\sigma)$ bits, at the expense of increasing the length of codewords by a factor arbitrarily close to (but bounded away from) 1. Using this with section 2 we get a result that is very close to Theorem 1, but where the additive term in the space usage does not decrease with p_1.

4 Approximate Membership with Arbitrary Error

As a building block to be used in the next section, we now consider the approximate membership problem. Using traditional theory on approximate membership (AM), the theoretical lower limit for the space usage of AM structures is $n\log_2(1/\varepsilon)$ bits [6]. In known optimal constructions, the limit can only be reached (or reached within a factor $1+\delta$) when ε is a negative power of 2. We now describe a way to circumvent this limitation that is more efficient than simply choosing a lower false positive probability.

We describe an AM structure with near optimal space complexity and with an arbitrary false positive error rate, $\varepsilon = c\,2^{-i}$, where $c \in [1/2;1]$, and i is a positive integer. Let g be a random hash function, $g : D \to [0;1]$ mapping the keys in S uniformly and independently to $[0;1]$ (a discrete approximation would be needed in an implementation, but the analysis would be essentially the same as in the idealized scenario). Let $\gamma = 2(1-c) \in [0;1]$, and divide S into two subsets, S_1 and S_2, mapping to values smaller and larger than γ respectively. Formally

$$S_1 = S \cap g^{-1}([0;\gamma])$$
$$S_2 = S \cap g^{-1}((\gamma;1]).$$

Note that the expected number of keys in S_1 is a fraction $2(1-c)$ of all the keys in S.

The AM structure is made up of two ordinary AM structures, each with a false positive rate that is a negative power of 2: one for S, with false positive rate

$\varepsilon_0 = 2^{-i}$, and one for S_1, with $\varepsilon_1 = 1/2$. A query on a key x is performed by first consulting the larger structure. In case of a negative answer, we know the key is not present. In case of a positive answer, we calculate $g(x)$. If $g(x)$ indicates that the key is in S_2, we return a positive answer. Otherwise, we consult the smaller AM structure, and return the answer obtained.

It is easy to analyze the expected false positive rate of the AM structure described above. Consider the false positives of the structure for S. For a fraction $2(1-c)$ of those, the hash function g will point to S_1 and we therefore consult the AM structure for S_1. This will in turn result in a negative result for half of the queries, since $\varepsilon_1 = 1/2$. That is, a fraction $1-c$ of the false positives of the structure for S has been eliminated. Since the structure for S has an error rate $\varepsilon_0 = 2^{-i}$, the error rate of the whole AM structure is $\varepsilon = c\,2^{-i}$.

The expected space usage of the AM structure is $i + 2(1-c)$ bits per key, which is close to the optimal $\log(1/\varepsilon) = i - \log c$ bits per key. In fact, the expected space usage is at most $1 - \log_2 e + \log_2 \log_2 e \approx 0.086$ bits per key from the optimal value.

5 Improvement for skewed distributions

We now provide the last part needed to show Theorem 1. In particular, we focus on the setting where the probability p_1 of the most frequent value is large. The idea is to use one or more approximate membership data structures as "filters" that determine a large fraction of values at little cost.

The redundancy of a prefix-free code is the difference between the expected cost of coding and the theoretical lower limit cost. The lower limit for coding a sequence of symbols related to n keys (assuming that values and corresponding keys are independent) is nH_0, where H_0 is the 0th order entropy of the value distribution. The redundancy per key is therefore $r = E(|u|) - H_0$.

Gallager [10] established that the redundancy of a Huffman code could be bounded by

$$r \leq p_1 + \rho \tag{4}$$

where p_1 is the probability of the most frequent symbol a_1 and $\rho = 1 - \log_2 e + \log_2 \log_2 e \approx 0.086$. For $p_1 \geq 0.5$ the bound further reduces to $r \leq p_1$. Together with the analysis in section 3 this immediately implies Theorem 1 whenever $p_1 < 1/2$.

In the following we examine the case where $p_1 \geq 0.5$, and determine a constant bound lower than 1, by adding a filter for handling the most frequent symbol. More specifically, we create an approximate membership structure, using the approach from the previous section. The structure supports $O(1)$ time membership queries on n keys, in space

$$(1+\delta)n(\log_2(1/\varepsilon) + \rho) \text{ bits,}$$

where $\rho \approx 0.086$ and ε is the false positive rate. Recall that the data structure returns "true" for any key in the filter, and "false" with probability at least $1-\varepsilon$ for other keys.

We build an AM structure for the subset S^* that contains all keys in S except those mapping to a_1. All queries start by consulting this AM structure. A negative result for a key $x \in S$ means that the key with certainty maps to a_1. A positive result, however, means that the key is either in S^* or is a false positive (in which case it maps to a_1). The remaining task is therefore to store the restriction of f to the set S_{AM} of keys for which the AM structure returned "true". This can be done using the method previously described (base case), or recursively using the same method — we analyze the former case, where only one filter is used. The expected number of keys in S_{AM} is $(1 - p_1 + \varepsilon p_1)n$.

In terms of our original code tree, this approach corresponds to adding a top level node, with the old root and a new a_1 leaf as children. The new code is one that allows two codewords for one symbol; one represented by the new leaf, and another in a modified version of the old tree. The advantage lies in the the low cost of coding the first bit using an AM structure, analyzed in the following.

5.1 Space Analysis

We now examine the space cost incurred by adding a filter as described. To simplify the calculations we pretend that Huffman coding is used rather than the method described in section 3. This means that all space bounds should be multiplied by a factor $1 + \delta/3$. The size of the approximate membership data structure with false positive rate ε is

$$n(1 - p_1)(\log_2(1/\varepsilon) + \rho) \text{ bits.} \tag{5}$$

If we let $\alpha = (1 - p_1 + \varepsilon p_1)$ and assume that a_2 is the second most frequent symbol, with probability p_2, Gallager [10] tells us that the space for encoding the remaining values using a Huffman code can be bounded by

$$\alpha n(H_0' + \frac{p_2}{\alpha} + \rho), \tag{6}$$

bits per value, where H_0' is the 0th order entropy of the distribution of values over keys in S_{AM}. Note that a_2 may be identical to a_1 if the same value remains the most frequent.

In order to determine the redundancy of the combined structure, we express H_0' in terms of the original entropy H_0, p_1, and ε,

$$H_0' = \sum_{i=2}^{\sigma} \left(\frac{n_i}{\alpha n} \log \frac{\alpha n}{n_i} \right) + \frac{\varepsilon n_1}{\alpha n} \log \frac{\alpha n}{\varepsilon n_1}, \tag{7}$$

By using

$$H_0 = \sum_{i=2}^{\sigma} \left(\frac{n_i}{n} \log \frac{n}{n_i} \right) + \frac{n_1}{n} \log \frac{n}{n_1}, \tag{8}$$

we may conclude from (7) that

$$\alpha H_0' = H_0 + p_1 \log p_1 + (1 - p_1) \log \alpha + \varepsilon p_1 \log \left(\frac{\alpha}{\varepsilon p_1} \right). \tag{9}$$

Summing the space usage from (5) and (6) and subtracting the lower bound H_0 we can bound the redundancy per value r by a function of p_1, p_2, and ε:

$$r < \log p_1 + \alpha \log\left(\frac{\alpha}{\varepsilon p_1}\right) + p_2 + (1 - p_1 + \alpha)\rho. \tag{10}$$

If we choose $\varepsilon = 1 - p_1$, which is a near-optimal choice, we get $\alpha = 1 - p_1^2$, and using $p_2 \leq 1 - p_1$ the redundancy is bounded by

$$\hat{r}(p_1) = \log p_1 + (1 - p_1^2)\log\left(\frac{1 + p_1}{p_1}\right) + (1 - p_1) + (2 - p_1 - p_1^2)\rho. \tag{11}$$

The function $\hat{r}$ is convex, $\hat{r}(1) = 0$, and $\frac{du}{dt}\hat{r}(1) > -1.82$. Therefore $\hat{r}(p_1) \leq 1.82(1 - p_1)$. At the same time, Gallager's bound means that we should not use filtering whenever the resulting redundancy is above p_1 bits per element (assuming $p_1 > 0.5$). The crossover happens around $p_1 = 0.63$, meaning that this is the maximum redundancy.

We have shown the following upper bound for the redundancy of our prefix-free code:

$$r < \min(p_1 + 0.086, 0.63, 1.82(1 - p_1)) \tag{12}$$

providing an alternative to Gallager's [10] bound for Huffman codes. Our bound is an improvement when there is a significant imbalance in the distribution of symbols, in the way that one symbol dominates with $p_1 > 0.63$. In the statement of Theorem 1 we have omitted the middle term, which is only a small improvement for a narrow range of p_1 values.

6 Conclusion

We have described a data structure for space efficiently representing a function with skewed function values. The representation uses space close to the entropy of the function values, and is independent of the size of the domain of the function.

Our adaptation of [9] to handle variable-length strings is nontrivial in the sense that the most straightforward generalizations do not seem to work, while our method of choosing hash functions that are slightly correlated does. In addition, we have introduced several techniques that may be of independent interest: A general reduction that gives approximate membership data structures with arbitrary error probability, the use of filtering for efficient compression, and constant time decoding of a Huffman-like code.

Acknowledgement. We thank the anonymous reviewers for their thorough comments.

References

1. D. Belazzougui, P. Boldi, R. Pagh, and S. Vigna. Monotone minimal perfect hashing: Searching a sorted table with $O(1)$ accesses. In *Proceedings of the 20th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '09)*. ACM Press, 2009.

2. D. Belazzougui, P. Boldi, R. Pagh, and S. Vigna. Theory and practise of monotone minimal perfect hashing. In I. Finocchi and J. Hershberger, editors, *ALENEX*, pages 132–144. SIAM, 2009.
3. B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.
4. F. C. Botelho, R. Pagh, and N. Ziviani. Simple and space-efficient minimal perfect hash functions. In *Proceedings of the 10th International Workshop on Algorithms and Data Structures (WADS '07)*, volume 4619 of *Lecture Notes in Computer Science*, pages 139–150. Springer, 2007.
5. N. J. Calkin. Dependent sets of constant weight binary vectors. *Combinatorics, Probability & Computing*, 6(3):263–271, 1997.
6. L. Carter, R. Floyd, J. Gill, G. Markowsky, and M. Wegman. Exact and approximate membership testers. In *Proceedings of the 10th Annual ACM Symposium on Theory of Computing (STOC '78)*, pages 59–65. ACM Press, 1978.
7. D. Charles and K. Chellapilla. Bloomier filters: A second look. In *Proceedings of the 16th European Symposium on Algorithms (ESA '08)*, volume 5193 of *Lecture Notes in Computer Science*, pages 259–270. Springer, 2008.
8. B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal. The Bloomier filter: An efficient data structure for static support lookup tables. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '04)*, pages 30–39. ACM Press, 2004.
9. M. Dietzfelbinger and R. Pagh. Succinct data structures for retrieval and approximate membership. In *Proceedings of the 35th International Colloquium on Automata, Languages and Programming (ICALP '08)*, Lecture Notes in Computer Science. Springer, 2008.
10. R. Gallager. Variations on a theme by Huffman. *IEEE Transactions on Information Theory*, 24(6):668–674, 1978.
11. T. Hagerup. Sorting and searching on the word RAM. In *Proceedings of the 15th Symposium on Theoretical Aspects of Computer Science (STACS '98)*, volume 1373 of *Lecture Notes in Computer Science*, pages 366–398. Springer-Verlag, 1998.
12. T. Hagerup and T. 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.
13. D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers*, 40(9):1098–1101, 1952.
14. L. L. Larmore and D. S. Hirschberg. A fast algorithm for optimal length-limited Huffman codes. *Journal of the ACM*, 37(3):464–473, July 1990.
15. R. L. Milidiú and E. S. Laber. Bounding the inefficiency of length-restricted prefix codes. *Algorithmica*, 31(4):513–529, 2001.
16. R. Motwani and P. Raghavan. *Randomized algorithms.* Cambridge University Press, 1995.
17. M. Pătraşcu. Succincter. In *Proceedings of the 49th Annual Symposium on Foundations of Computer Science (FOCS '08)*, pages 305–313, 2008.
18. E. Porat. An optimal Bloom filter replacement based on matrix solving. *CoRR*, abs/0804.1845, 2008.
19. J. P. Schmidt and A. Siegel. The spatial complexity of oblivious k-probe hash functions. *SIAM J. Comput.*, 19(5):775–786, 1990.
20. D. Talbot and J. M. Talbot. Bloom maps. In *Proceedings of the Fourth Workshop on Analytic Algorithmics and Combinatorics (ANALCO)*. IEEE, 2008.