Optimal Compression of Element Pairs in Fixed-Width Memories

Ori Rottenstreich[*] and **Yuval Cassuto**[†]
[*]Department of Computer Science, Princeton University, USA
[†]Viterbi Electrical Engineering Department, Technion – Israel Institute of Technology, Israel
orir@cs.princeton.edu, ycassuto@ee.technion.ac.il

***Abstract*—Data compression is a well-studied (and well-solved) problem in the setup of long coding blocks. But important emerging applications need to compress data to memory words of small fixed widths. This new setup is the subject of this paper. In the problem we consider we have a source with a known discrete distribution, and we wish to find a code that maximizes the success probability that two source instances can be represented together in L bits or less. A good practical use for this problem is a table with two-element entries that is stored in a memory of a fixed width L. Such tables of very large sizes are used in data-intensive computing applications. We solve the problem by efficiently finding an optimal code that uses a dictionary of linear size in the number of source elements.**

I. Introduction

In the best-known data-compression problem, a discrete distribution on source elements is given, and one wishes to find a representation for the source elements with minimum expected length. This problem was solved by the well-known Huffman coding [2]. Huffman coding reduces the problem to a very efficient recursive algorithm on a tree, which solves it optimally. Indeed, Huffman coding has found use in numerous communication and storage applications. Minimizing the expected length of the coded sequence emitted by the source translates to optimally low transmission or storage costs in those systems. However, there is an important setup in which minimizing the expected length does *not* translate to optimal improvement in system performance. This setup is *fixed-width memory*. Information is stored in a fixed-width memory in words of L bits each. A word is designated to store an entry of 2 fields, each emitted by the same n-element source. In this case the prime objective is to fit the representations of the 2 elements into L bits, and not to minimize the expected length of individual coded elements.

The fixed-width memory setup is extremely useful in real applications. In some data-intensive applications we want fast access to data entries composed of element pairs. Fast access is achieved by storing the entry in a memory word of fixed-width L. The massive number of data entries in these applications requires good compression for cost reduction. In this paper we consider the case where elements are drawn i.i.d. from some known distribution, and in particular the two elements of an entry are independent of each other. Our operational model for the compression assumes that success to fit an entry to L bits translates to fast data access, while failure results in a slower access because a secondary slower memory is accessed to fit the entry. Therefore, we want to maximize the number of entries we succeed to fit, but do allow some (small) fraction of failures. Correspondingly, the performance measure we consider throughout the paper is P_{success}: the probability that the encoding of the two-element entry fits in L bits. That is, we seek a code whose encoding function is allowed to declare encoding failure on some entries, but does so with the lowest possible probability for the source distribution. We emphasize that we do *not* allow decoding errors, and all entries that succeed encoding are recovered perfectly at read time. Our primary goal is to achieve optimal success probability while only allowing a single dictionary of size $O(n)$ to hold the code mapping. A trivial solution to the problem with $O(n^2)$ size dictionary is too costly in memory, hence the requirement for linear-size dictionary motivating this paper.

Finding codes that give optimal P_{success} cannot be done by known compression algorithms. Also, brute-force search for an optimal code has complexity that grows exponentially with n. This paper solves the problem of optimal fixed-width compression by presenting an efficient algorithm that takes any element distribution and an integer L, and outputs a code with optimal P_{success}. The key novelty toward the solution is formulating an efficient dynamic-programming algorithm building on a clever decomposition of the problem to smaller sub-problems.

In terms of prior work, the notion of fixed-width representations is captured by the work of Tunstall on variable-to-fixed coding [3], but without the feature of multi-element entries. The problem of fixed-width encoding is also related to the problem of compression with low probability of buffer overflow [4], [5]. However, the prior work only covers an asymptotic number of encoding instances (in our problem the number of encoding instances is fixed to 2). The most directly related to the results of this paper is our prior work [1], [6] considering other algorithmic problems related to fixed-width compression.

II. Model and Problem Formulation

We start by providing some definitions, upon which we next cast the formal problem formulation. Throughout the paper we assume that data placed in the memory come from a known distribution, as now defined.

Definition 1 (Element Distribution). *An* element distribution $(S, P) = ((s_1, \ldots, s_n), (p_1, \ldots, p_n))$ *is characterized by an (ordered) set of elements with their corresponding positive appearance probabilities. An element of* S *is drawn randomly and independently according to the distribution* P*, i.e.,* $\Pr(a = s_i) = p_i$*, with* $p_i > 0$ *and* $\Sigma_{i=1}^{n} p_i = 1$.

Throughout the paper we assume that the elements s_i are ordered in *non-increasing* order of their corresponding probabilities p_i. Given an element distribution, a *fixed-to-variable code* σ, which we call *code* for short, is a set of binary codewords B (of different lengths in general), with a mapping from S to B called an *encoder*, and a mapping from B to S called a *decoder*. We denote the encoder function of the code σ by $\sigma(\cdot)$, and the decoding function by $\sigma^{-1}(\cdot)$; we assume that the encoder and decoder are implemented through a simple dictionary, listing the mapping between the elements $a \in S$ and their codewords $\sigma(a)$.

In this paper we are interested in coding data entries composed of *element pairs*. Each coded pair needs to be stored in a memory word of fixed width L. This setup calls for the following definitions.

Definition 2 (Entry Encoding Function). *An* entry encoding function Σ *is a mapping* $\Sigma : (S, S) \to \{0, 1\}^L \cup \{\perp\}$*, where* $\{0, 1\}^L$ *is the set of binary vectors of length* L*, and* $\perp$ *is a special symbol denoting encoding failure.*

The input to the encoding function are two elements taken from S. In successful encoding, the encoder outputs a binary vector of length L to be stored in memory, and this vector is obtained uniquely for this input entry. The encoder is allowed to fail, in which case the output is the special symbol $\perp$. When the encoder fails, we assume that an alternative representation is found for the entry, and stored in a different memory not bound to the width-L constraint. This alternative representation is outside the scope of this work, but because we know it results in higher cost (in space and/or in time), our objective here is to minimize the failure probability of the encoding function. Before we formally define the failure probability, we give a definition of the decoding function matching the encoding function of Definition 2.

Definition 3 (Entry Decoding Function). *An* entry decoding function Π *is a mapping* $\Pi : \{0, 1\}^L \to (S, S)$*, such that if* $\Sigma(a_1, a_2) = b \neq \perp$*, then* $\Pi(b) = (a_1, a_2)$.

The definition of the decoding function is straightforward: it is the inverse mapping of the encoding function when encoding succeeds. With encoding and decoding in place, we turn to define the important measure of encoding success probability.

Definition 4 (Encoding Success Probability). *Given an element distribution* (S, P) *and an entry encoding function* Σ*, define the* encoding success probability *as*

$$P_{\text{success}}(P, \Sigma) = \Pr(\Sigma(a_1, a_2) \neq \perp), \tag{1}$$

where the probability is calculated over all pairs of (a_1, a_2) *drawn independently from* (S, S) *according to the distribution* P.

A code σ for (S, P) induces a **direct entry encoding function** by concatenating two encoding instances. That is, the direct entry encoding function in successful instances is $\Sigma(a_1, a_2) = [\sigma(a_1), \sigma(a_2), 0, \ldots, 0]$, where , represents concatenation and the zeros are padding to get an L-bit vector. When the representation lengths of $\sigma(a_1), \sigma(a_2)$ sum to more than L bits, we get $\Sigma(a_1, a_2) = \perp$, i.e., encoding failure. If the code σ is a *prefix code*, then the direct entry encoding function has a straightforward entry decoding function by parsing the L-bit vector b using the prefix property to two codewords of σ (plus potentially some trailing zeros). Recall the definition of a prefix code [7]:

Definition 5 (Prefix Code). *For a set of elements* S*, a code* σ *is called a* prefix code *if in its codeword set* B *no codeword is a prefix (start) of any other codeword.*

The advantage of the direct entry encoding function and its corresponding decoding function is that it is sufficient to hold a dictionary for σ of size $O(n)$, despite having n^2 possible combinations of (a_1, a_2). Hence our main objective in this paper is to design the (prefix) code σ to maximize the success probability of its direct entry encoding function. We note that the problem of maximizing the success probability with $O(n^2)$ dictionary size is trivial, because we can order all pairs (a_1, a_2) in non-increasing joint probabilities and assign fixed-length codewords to the 2^L pairs with the highest probabilities.

III. Algorithm for Optimal-Success Codes

In this section we present an algorithm that finds the optimal code σ in the sense of maximizing the success probability of its direct entry encoding function. To construct the code σ we assign codewords to n' elements of S, where $n' \leq n$. Clearly, all such codewords have lengths of at most L bits. For a (single) binary string x, let $\ell(x)$ denote the *length* in bits of x. Since for every element $a \in S$ that is assigned a codeword the code σ satisfies $\ell(\sigma(a)) \leq L$, it holds that $2^{-\ell(\sigma(a))}$ must be an integer multiple of 2^{-L}. We define the *weight* of a codeword of length ℓ_0 as the number of units of 2^{-L} in $2^{-\ell_0}$, denoted by $N_{\ell_0} = 2^{-\ell_0}/2^{-L} = 2^{L-\ell_0}$. The $n - n'$ elements of S not represented by σ are said to have length $\ell_0 = \infty$ and codeword weight zero. A prefix code exists with prescribed codeword lengths if they satisfy Kraft's inequality [7]. In our terminology, this means that the sum of weights of the codewords of σ needs to be at most 2^L.

It can be proved that without loss of generality the optimal code σ is *monotone*, that is, $i < i'$ implies that $\ell(\sigma(s_i)) \leq \ell(\sigma(s_{i'}))$ for two elements $s_i, s_{i'} \in S$ (recall that by the assumed ordering $p_i \geq p_{i'}$). In a monotone code it is sufficient to list the n' codeword lengths to specify the code.

A. Naïve attempts at optimal σ

An encoding success occurs when the codeword lengths assigned to S maximize the probability that a pair (a_1, a_2) fits in L bits. A conceptually simple algorithm exhaustively tries all monotone length assignments whose weights sum to 2^L, and evaluate their success probabilities. However, even with the monotone restriction there is an exponential (in n) number of codes to try. A less naïve approach is divide-and-conquer, whereby we allocate weight N' to the first k elements in S, and the remaining weight $2^L - N'$ to the last $n-k$ elements. It appears that this would allow finding the optimal length assignment by trying a small number of weight partitions. However, this approach does not work because the optimal assignment of the first part depends on the assignment of the second part, and the two cannot be decoupled this way to smaller sub-problems.

B. *Efficient algorithm for optimal σ*

In the remainder of the section we show an algorithm that offers an efficient way around the above-mentioned difficulty to assign codeword lengths to elements. We present this efficient algorithm formally, but first note its main idea.

The main idea: For the dependence issue noted in the previous sub-section, it is not possible to maximize the success probability for $k+1$ elements given the optimal codeword lengths for k elements. So it does not work to successively add elements to the solution while maintaining optimality as an invariant. But fortunately, it turns out that it does work to successively add *codeword lengths* to the solution while maintaining optimality as an invariant. The subtle part is that for this idea to work the lengths need to be added in a carefully thought-of sequence, which in particular, is *not* the linear sequence $(1, 2, \dots, L-1)$ or its reverse-ordered counterpart. We show that if the codeword lengths are added in the order of the sequence

$$(L/2, L/2+1, L/2-1, L/2+2, L/2-2, \dots, L-1, 1) \quad (2)$$

(for even[1] L), then for any sub-sequence we can maximize the success probability given the optimal codeword lengths taken from the sub-sequence that is one shorter. For example, when $L = 8$ our algorithm will first find an optimal code only using codeword length $L/2 = 4$; based on this optimum it will find an optimal code with lengths 4 and 5, and then continue to add the codeword lengths $3, 6, 2, 7, 1$ in that order. This approach does work, because when following that length sequence the success probability of the assignment of lengths $L/2, \dots, l$ does not depend on the assignments of subsequent lengths (shorter lengths will succeed encoding regardless of the length assignment, and longer lengths will fail regardless of the length assignment).

We now turn to a formal definition of the algorithm. We first define the function holding the optimal success probabilities for sub-problems of the problem instance.

Definition 6. *Consider assignments of finite codeword lengths to the consecutive elements $\{s_{k_1}, \dots, s_{k_2}\}$ from S, where the lengths are assigned from the values $\{L/2, L/2+1, L/2-1, \dots, l\}$ taken from the sub-sequence of (2) that ends with l. For $N \in [0, 2^L]$ we denote by $G(l, [k_1, k_2], N)$ the maximal success probability for such an assignment whose sum of weights for these $k_2 - k_1 + 1$ codewords is at most N. G is defined formally in (3), where $I[\cdot]$ is the indicator function.*

[1]For convenience we assume that L is even, but all the results extend to odd L.

The following two theorems are the key drivers of the efficient dynamic-programming algorithm finding the optimal code.

Theorem 1. *Let $l = L/2 - d + 1$ for some integer $1 \le d \le L/2 - 1$. For the length following l in the sequence (2) $l' = L/2 + d$, we have*

$$G(l', [k_1, k_2], N) = \max_{j \in [0, k_2 - k_1 + 1]} G(l, [k_1, k_2 - j], N - j \cdot N_{l'}), \quad (4)$$

where we define

$$G(l, [k_1, k_1 - 1], N) \triangleq 0, \text{ for } N \ge 0. \quad (5)$$

Proof. Given maximal values G for all values of N and with lengths up to l in the sequence, the maximal value G when l' is also allowed is obtained by assigning length l' to between 0 and $k_2 - k_1 + 1$ elements in the range $[k_1, k_2]$. By the monotonicity of p_i, l' which is *higher* than all previous lengths must be assigned to the highest j indices in the range $[k_1, k_2]$. Thus for each j the success probability is the value of G for the corresponding range of elements $[k_1, k_2 - j]$ with the residual weight $N - j \cdot N_{l'}$. In particular, the elements assigned length l' do not add to the success probability, because l' plus any length in the sub-sequence up to l exceeds L. In the extreme case when $j = k_2 - k_1 + 1$ (all elements assigned length l'), the definition (5) when appearing in the right-hand side of (4) gives a valid assignment with success probability 0 if N in the left-hand side is sufficiently large. □

An important property provided by the length sequence (2) is that the success probability up to l' depends only on the success probability up to l, no matter what the assignments are among the lengths up to l. With this property Theorem 1 allows to efficiently extend the optimality from l of type $l = L/2 - d + 1$ (with $d \ge 1$) to the next length in the sequence. To complete what is required for an efficient algorithm, we need the same extension of optimality from l of type $l = L/2 + d$ to the next length in the sequence. We do this in the next theorem.

Theorem 2. *Let $l = L/2 + d$ for some integer $1 \le d \le L/2 - 1$. For the length following l in the sequence (2) $l' = L/2 - d$, we have $G(l', [k_1, k_2], N)$ given in (6).*

Proof. Given maximal values G for all values of N and with lengths up to l in the sequence, the maximal value G when l' is also allowed is obtained by assigning length l' to between 0 and $k_2 - k_1 + 1$ elements in the range $[k_1, k_2]$. By the monotonicity of p_i, l' which is *lower* than all previous lengths must be assigned to the lowest j indices in the range $[k_1, k_2]$. Now the success probability has two components: first is the success between pairs of elements assigned lengths up to l in the sequence, and second is the success between element pairs

$$G(l,[k_1,k_2],N) = \max_{\sigma:\left(\forall i\in[k_1,k_2]:\ell(\sigma(s_i))\in\{L/2,\ldots,l\}\ ,\ \sum_{i=k_1}^{k_2} N_{\ell(\sigma(s_i))}\le N\right)} \left(\sum_{i=k_1}^{k_2}\sum_{i'=k_1}^{k_2} p_i\cdot p_{i'}\cdot I\Big[\ell(\sigma(s_i))+\ell(\sigma(s_{i'}))\le L\Big]\right). \quad (3)$$

$$G(l',[k_1,k_2],N) = \max_{j\in[0,k_2-k_1+1]}\left[G(l,[k_1+j,k_2],N-j\cdot N_{l'}) + \left(\sum_{i=k_1}^{k_1+j-1} p_i\right)\left(\sum_{i=k_1}^{k_2} p_i\right) + \left(\sum_{i=k_1+j}^{k_2} p_i\right)\left(\sum_{i=k_1}^{k_1+j-1} p_i\right)\right]. \quad (6)$$

that involve the new length l' (recall that in Theorem 1 the second component did not exist, but here it does because l' sums to at most L with any previous length in the sequence.) The first of the three terms in the summation of (6) gives the first component, and the latter two terms give the second component. Maximization is done as before by considering all possible j with the residual weight $N - j\cdot N_{l'}$. $\square$

As in Theorem 1, the length sequence provides the property that maximization in Theorem 2 only depends on the success probabilities of the previous length sub-sequence, without care to the specific assignment of lengths among the lengths in that sub-sequence. In the following Algorithm 1 we formally present the algorithm for finding an optimal prefix code σ, building on Theorems 1 and 2. For terseness we only track the optimal success probabilities G, omitting the more technical task of tracking the optimal assigned lengths, which is required to find the optimal code in a real implementation. The noteworthy parts of Algorithm 1 are:

- The initialization of G to $-\infty$ for negative N, and to 0 for non-negative N and empty ranges of elements (according to (5)).
- Starting the length sequence at $l = L/2$ and calculating the success probability when all elements in the range are assigned that length.
- The main iteration following the progressions in the length sequence using Theorems 1 and 2.
- Outputing the optimal success probability for the code parameters as a maximization over all n possible numbers of elements assigned codewords, starting from the first element.

Time Complexity: There are $L-1$ half iterations (go right or go left in Algorithm 1) and in each $O(2^L)$ values are calculated for each of $O(n^2)$ ranges of elements, each by considering $O(n)$ possibilities for the number of elements with the new codeword length. It follows that the time complexity of the algorithm is $O(n^3\cdot 2^L\cdot L)$, which is polynomial in the size of the input, or $O(n^5\cdot L)$ (2^L is at most quadratic in n, because otherwise we can fit all n elements in $L/2$ bits with a fixed-length code). We note that the n^5 complexity term is a loose bound, because many of the counted iterations are not exercised in a given run.

Example 1. *Consider the element distribution* $(S,P) = ((s_1,\ldots,s_n),(p_1,\ldots,p_n))$ *satisfying* $(p_1,\ldots,p_n) = (0.4, 0.4, 0.08, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01)$ *with* $n = 15$. *The width parameter is* $L = 6$. *A simple possible coding scheme can assign codewords of length* $L/2 = 3$ *to the* $2^3 = 8$ *most probable elements* $s_1,\ldots,s_8$, *while codewords are not assigned for the remaining elements* $s_9,\ldots,s_{15}$. *This scheme successfully encodes all pairs of elements from* $s_1,\ldots,s_8$ *achieving success probability of* $P'_{\text{success}} = \left(\sum_{i=1}^{8} p_i\right)^2 = 0.93^2 = 0.8649$. *Alternatively, the above optimal algorithm finds a code that achieves higher success probability. In this code elements* s_1, s_2 *are assigned codewords of length 2, while elements* $s_3,\ldots,s_{10}$ *are assigned codewords of length 4. Later elements are not assigned codewords. This optimal scheme achieves an improved success probability of* $P_{\text{success}} = \left(\sum_{i=1}^{2} p_i\right)^2 + 2\cdot\sum_{i=1}^{2} p_i\cdot\sum_{i=3}^{10} p_i = 0.8^2 + 2\cdot 0.8\cdot 0.15 = 0.88$.

Algorithm 1 can be extended to the case where the two elements of the data entry have *different distributions* on the same elements, and a single code with optimal success probability is found. We do not explore this generalization in the paper, but it is as simple as replacing some instances of p_i in the algorithm by q_i corresponding to the distribution of the second distribution.

IV. Simulation Results

We examine the performance of the proposed code construction. In the experiments, the probabilities in the element distribution follow the Zipf distribution. A low positive Zipf parameter μ results in a distribution that is close to the uniform distribution, while for a larger parameter the distribution is more biased. We compare the proposed optimal codes to Huffman coding [2]. The results are shown in Fig. 1. The number of elements is $n = 128$ with a Zipf $\mu = 1.6$ distribution. We compute the optimal code σ using Algorithm 1, and plot its success probability. In comparison, we also plot the success probability of the Huffman code computed from the same element distribution. Huffman coding gives inferior success to the optimal code, with a maximal probability difference of 0.194 at $L = 4$. In Fig 2, we plot the success probability as a function of the source parameter n, with Zipf parameter $\mu = 0.5$. The minimal width required to obtain a success probability of 1 for n elements is given by $L = 2 \cdot \log_2 n$, i.e. $L \geq 6$ for $n = 8$ and $L \geq 8$ for $n = 16$. For a given L, the optimal success probability decreases when the number of elements increases.

V. Conclusion

We solved the problem of optimally and efficiently compressing element pairs in a fixed-width memory. We note two important problems left open by this paper's results. One is whether there is a simpler (and more elegant) "Huffman-like" algorithm that gives an optimal solution. Another is how to extend the optimal algorithm to more than 2 elements in an entry.

Algorithm 1: Optimal Fixed-Width Code

```
input : Element distribution $(S, P)$, memory width $L$
output: Prefix code $\sigma$ with optimal direct entry-encoding function
initialization:
foreach $N < 0$ do $G(l, [k_1, k_2], N) = -\infty$ for all indices $l$ and $k_1 \leq k_2$ ;
foreach $N \geq 0$ do $G(l, [k, k-1], N) = 0$ for all indices $l$ and $k$ ;
codewords of length $L/2$:
foreach $[k_1, k_2] \subseteq [1, n]$, $k_1 \leq k_2$ do
    for $N = (k_2 - k_1 + 1) \cdot N_{L/2} : 2^L$ do
        $G(L/2, [k_1, k_2], N) = \left(\sum_{i=k_1}^{k_2} p_i\right)^2$
    end
end
main iteration:
for $d = 1 : L/2 - 1$ do
    go right to length $L/2 + d$:
    for $N = 0 : 2^L$ do
        foreach $[k_1, k_2] \subseteq [1, n]$, $k_1 \leq k_2$ do
            $G(L/2 + d, [k_1, k_2], N) = \max_{j \in [0, k_2 - k_1 + 1]} G(L/2 - d + 1, [k_1, k_2 - j], N - j \cdot N_{L/2+d})$
        end
    end
    go left to length $L/2 - d$:
    for $N = 0 : 2^L$ do
        foreach $[k_1, k_2] \subseteq [1, n]$, $k_1 \leq k_2$ do
            $G(L/2 - d, [k_1, k_2], N) = \max_{j \in [0, k_2 - k_1 + 1]} \Big[ G(L/2 + d, [k_1 + j, k_2], N - j \cdot N_{L/2-d}) + \left(\sum_{i=k_1}^{k_1+j-1} p_i\right)\left(\sum_{i=k_1}^{k_2} p_i\right) + \left(\sum_{i=k_1+j}^{k_2} p_i\right)\left(\sum_{i=k_1}^{k_1+j-1} p_i\right) \Big]$
        end
    end
end
output results:
Encoding success probability
$P_{\text{success}} = \max_{k \in [1,n]} G(1, [1, k], 2^L)$
```

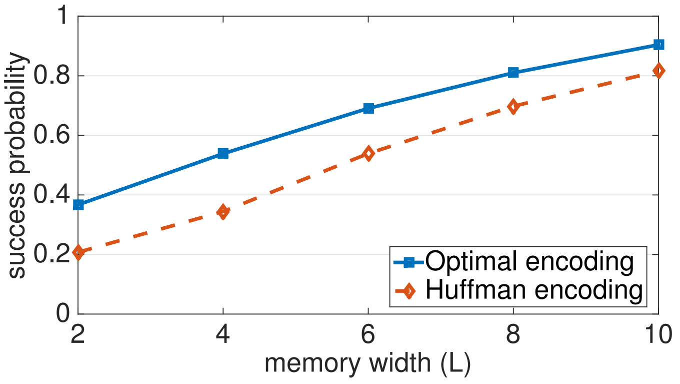

Fig. 1. Comparison of the optimal code vs. the Huffman code. Probabilities follow Zipf distribution with $n = 128$ and $\mu = 1.6$.

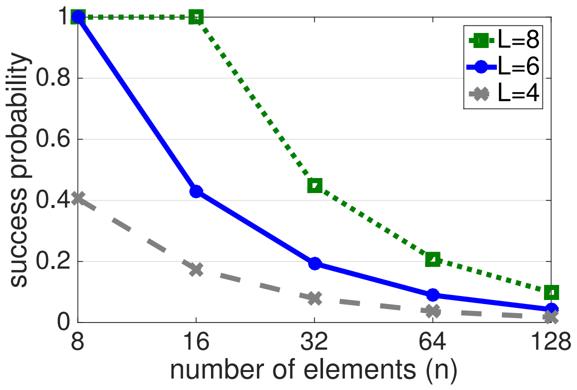

Fig. 2. The success probability of the optimal code as a function of the number of elements in the distribution for different values of the memory width L. Probabilities follow the Zipf distribution with parameter $\mu = 0.5$.

VI. Acknowledgment

This work was partly supported by the US-Israel Binational Science Foundation, by the Israel Science Foundation, and by the Intel ICRI-CI Center.

References

[1] O. Rottenstreich, A. Berman, Y. Cassuto, and I. Keslassy, "Compression for fixed-width memories," in *IEEE ISIT*, 2013.
[2] D. Huffman, "A method for the construction of minimum redundancy codes," *Proc. IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.
[3] B. P. Tunstall, *Synthesis of Noiseless Compression Codes.* Ph.D. dissertation, Georgia Inst. Tech., Atlanta, GA, 1967.
[4] F. Jelinek, "Buffer overflow in variable length coding of fixed rate sources," *IEEE Trans. on Information Theory*, vol. 14, no. 3, pp. 490–501, 1968.
[5] P. A. Humblet, "Generalization of Huffman coding to minimize the probability of buffer overflow," *IEEE Trans. on Information Theory*, vol. 27, no. 2, pp. 230–232, 1981.
[6] O. Rottenstreich, M. Radan, Y. Cassuto, I. Keslassy, C. Arad, T. Mizrahi, Y. Revah, and A. Hassidim, "Compressing forwarding tables for datacenter scalability," *IEEE Journal on Selected Areas in Communications*, vol. 32, no. 1, pp. 138–151, 2014.
[7] T. M. Cover and J. A. Thomas, *Elements of information theory (2. ed.).* Wiley, 2006.