Dipartimento di Informatica
Università del Piemonte Orientale "A. Avogadro"
Via Bellini 25/G, 15100 Alessandria
http://www.di.unipmn.it

Space-Conscious Compression

Author: T. Gagie (travis@mfn.unipmn.it) and
G. Manzini (manzini@mfn.unipmn.it)

TECHNICAL REPORT TR-INF-2007-06-02-UNIPMN
(June 2007)

The University of Piemonte Orientale Department of Computer Science Research Technical Reports are available via WWW at URL `http://www.di.unipmn.it/`. Plain-text abstracts organized by year are available in the directory

Recent Titles from the TR-INF-UNIPMN Technical Report Series

2007-01 *Markov Decision Petri Net and Markov Decision Well-formed Net Formalisms*, Beccuti, M., Franceschinis, G., Haddad, S., February 2007.

2006-04 *New challenges in network reliability analysis*, Bobbio, A., Ferraris, C., Terruggia, R., November 2006.

2006-03 *The Engineering of a Compression Boosting Library: Theory vs Practice in BWT compression*, Ferragina, P., Giancarlo, R., Manzini, G., June 2006.

2006-02 *A Case-Based Architecture for Temporal Abstraction Configuration and Processing*, Portinale, L., Montani, S., Bottrighi, A., Leonardi, G., Juarez, J., May 2006.

2006-01 *The Draw-Net Modeling System: a framework for the design and the solution of single-formalism and multi-formalism models*, Gribaudo, M., Codetta-Raiteri, D., Franceschinis, G., January 2006.

2005-06 *Compressing and Searching XML Data Via Two Zips*, Ferragina, P., Luccio, F., Manzini, G., Muthukrishnan, S., December 2005.

2005-05 *Policy Based Anonymous Channel*, Egidi, L., Porcelli, G., November 2005.

2005-04 *An Audio-Video Summarization Scheme Based on Audio and Video Analysis*, Furini, M., Ghini, V., October 2005.

2005-03 *Achieving Self-Healing in Autonomic Software Systems: a case-based reasoning approach*, Anglano, C., Montani, S., October 2005.

2005-02 *DBNet, a tool to convert Dynamic Fault Trees to Dynamic Bayesian Networks*, Montani, S., Portinale, L., Bobbio, A., Varesio, M., Codetta-Raiteri, D., August 2005.

2005-01 *Bayesan Networks in Reliability*, Langseth, H., Portinale, L., April 2005.

2004-08 *Modelling a Secure Agent with Team Automata*, Egidi, L., Petrocchi, M., July 2004.

2004-07 *Making CORBA fault-tolerant*, Codetta Raiteri D., April 2004.

2004-06 *Orthogonal operators for user-defined symbolic periodicities*, Egidi, L., Terenziani, P., April 2004.

2004-05 *RHENE: A Case Retrieval System for Hemodialysis Cases with Dynamically Monitored Parameters*, Montani, S., Portinale, L., Bellazzi, R., Leonardi, G., March 2004.

2004-04 *Dynamic Bayesian Networks for Modeling Advanced Fault Tree Features in Dependability Analysis*, Montani, S., Portinale, L., Bobbio, A., March 2004.

2004-03 *Two space saving tricks for linear time LCP computation*, Manzini, G., February 2004.

2004-01 *Grid Scheduling and Economic Models*, Canonico, M., January 2004.

Space-conscious compression

Travis Gagie and Giovanni Manzini *

Dipartimento di Informatica
Università del Piemonte Orientale
{travis,manzini}@mfn.unipmn.it

Abstract. Compression is most important when space is in short supply, so compression algorithms are often implemented in limited memory. Most analyses ignore memory constraints as an implementation detail, however, creating a gap between theory and practice. In this paper we consider the effect of memory limitations on compression algorithms. In the first part we assume the memory available is fixed and prove nearly tight upper and lower bounds on how much memory is needed to compress a string close to its k-th order entropy. In the second part we assume the memory available grows (slowly) as more and more characters are read. In this setting we show that the rate of growth of the available memory determines the speed at which the compression ratio approaches the entropy. In particular, we establish a relationship between the rate of growth of the sliding window in the LZ77 algorithm and its convergence rate.

1 Introduction

Data compression has come of age in recent years and compression algorithms are now vital in situations unforeseen by their designers. This has led to a discrepancy between the theory of data compression algorithms and their use in practice: compression algorithms are often designed and analysed assuming the compression and decompression operations can use a "sufficiently large" amount of working memory; however, in some situations, particularly in mobile or embedded computing environments, the memory available is very small compared to the amount of data we need to compress or decompress.

Even when compression algorithms are implemented to run on powerful desktop computers, some care is taken to be sure that the compression/decompression of large files do not take over all the RAM of the host machine. This is usually accomplished by splitting the input in blocks (bzip2), using heuristics to determine when to discard the old data (compress, ppmd), or by maintaining a "sliding window" over the more recently seen data and forgetting the oldest data (gzip). With the exception of the use of a sliding window (see Sect. 4), the validity of these techniques has not been established in a satisfying theoretical way.

In this paper we initiate the theoretical study of space-conscious compression algorithms. Although data compression algorithms have their own peculiarities,

* Both authors partly supported by Italian MUIR Italy-Israel FIRB Project "Pattern Discovery Algorithms in Discrete Structures, with Applications to Bioinformatics".

this study belongs to the general field of algorithmics in the streaming model (see, e.g., [1, 10]), in which we are allowed only one pass over the input and memory sublinear (possibly polylogarithmic or even constant) in its size.

Our results. The first contribution of this paper is nearly tight upper and lower bounds on the compression ratio achievable by one-pass algorithms that use an amount of memory independent of the size of the input. The bounds are worst case and given in terms of the empirical k-th order entropy of the input string. More precisely we prove the following results:

(a) Let $\lambda \geq 1$, $k \geq 0$ and $\epsilon > 0$ be constants and let g be a function independent of n. In the worst case it is impossible to store a string s of length n over an alphabet of size σ in $\lambda H_k(s)n + o(n \log \sigma) + g$ bits using one pass and $O(\sigma^{k+1/\lambda-\epsilon})$ bits of memory.
(b) Given a $(\lambda H_k(s)+o(n \log \sigma)+g)$-bit encoding of s, it is impossible to recover s using one pass and $O(\sigma^{k+1/\lambda-\epsilon})$ bits of memory.
(c) Given $\lambda \geq 1$, $k \geq 0$ and $\mu > 0$, we can store s in $\lambda H_k(s)n + \mu n + O(\sigma^{k+1/\lambda} \log \sigma)$ bits using one pass and $O(\sigma^{k+1/\lambda} \log^2 \sigma)$ bits of memory, and later recover s using one pass and the same amount of memory.

While σ is often treated as constant in the literature, we treat it as a variable to distinguish between, say, $O(\sigma^{k+1/\lambda-\epsilon})$ and $O(\sigma^{k+1/\lambda} \log^2 \sigma)$ bits. Informally, (a) provides a lower bound to the amount of memory needed to compress a string up to its k-th order entropy; (b) tells us the same amount of memory is required also for decompression and implies that the use of a powerful machine for doing the compression does not help if only limited memory is available when decompression takes place; (c) establishes that (a) and (b) are nearly tight. Notice λ plays a dual role: for large k, it makes (a) and (b) inapproximability results — e.g., we cannot use $O(\sigma^k)$ bits of memory without worsening the compression in terms of $H_k(s)$ by more than a constant factor; for small k, it makes (c) an interesting approximability result — e.g., we can compress reasonably well in terms of $H_0(s)$ using, say, $O(\sqrt{\sigma})$ bits of memory. The main difference between the bounds in (a)–(b) and (c) is a $\sigma^\epsilon \log^2 \sigma$ factor in the memory usage. Since μ is a constant, $\mu n \in o(n \log \sigma)$ and the bounds on the encoding's length match. Note that μ can be arbitrarily small, but the term μn cannot be avoided (Lemma 4).

A second contribution of this paper is the proof of lower bounds similar to (a) and (b) for the case in which the input string is generated by a stationary ergodic k-th order Markov source $\mathcal{X}$. We show that, with high probability, a length-n string drawn from $\mathcal{X}$ cannot be stored in $\lambda H(\mathcal{X})n + o(n \log \sigma) + g$ bits using one pass and $o(\sigma^k \log \sigma)$ bits of memory (here λ, σ and g have the same meaning as in (a)). A symmetrical result holds for the space required for recovering a string emitted by $\mathcal{X}$ and compressed up to $\lambda H(\mathcal{X})n + o(n \log \sigma) + g$ bits. Note that an upper bound analogous to (c) automatically holds for strings generated by an ergodic k-th order Markov source $\mathcal{X}$ (the term H_k is simply replaced by $H(\mathcal{X})$).

The final contribution of the paper is a first step in the analysis of the power of compressors when the amount of available working memory grows with the size of the input. We model this behavior assuming that we are given an increasing function $f(t)$ and that after reading t characters the compression algorithm is

allowed to use $\Theta(f(t))$ bits of memory. In this setting, the result (c) above implies (Lemma 7) that for any diverging function f (i.e. $\lim_{t\to\infty} f(t) = +\infty$) it is possible to compress every string up to its k-th order entropy *for any* $k \geq 0$. Given this state of affairs, it is clear that to understand the role played by the rate-of-growth function f, we must go deeper than simply considering whether the compression ratio approaches the k-th order entropy. We initiate this study with the analysis of LZ77 with a sliding window, which is the algorithm at the heart of the gzip tool. We show quantitatively that the rate-of-growth function f influences the convergence rate of the algorithm; that is, the speed at which the algorithm approaches the k-th order entropy. In particular, now treating σ as a constant, we prove that

(d) if LZ77 uses a sliding window that grows as $f(t) = t/\log^2 t$, then for any string s, the output size is bounded by $H_k(s)n + O((n \log\log n)/\log n)$ simultaneously for any $k \geq 0$;
(e) if LZ77 uses a sliding window that grows as $f(t) = \log^{1-\epsilon} t$, with $0 < \epsilon < 1$, then for any $n > 0$ we can build a string $\hat{s}$ of length n such that LZ77's output size is at least $H_0(s)n + \Omega\big((n \log\log n)/\log^{1-\epsilon} n\big)$ bits.

In other words, a faster growing sliding window yields a provably faster rate of convergence. To our knowledge, these are the first results relating the size of LZ77's sliding window and its rate of convergence in the worst case setting. In the probabilistic setting (see below) what it is known [14] is that using a window of *fixed* size W the rate of convergence of LZ77 is $\Theta((n \log\log W)/\log W)$.

2 Notation

In the following we use s to denote the string that we want to compress. We assume that s has length n and is drawn from an alphabet of size σ. Note that in Section 3 we measure memory in terms of alphabet size so σ is considered a variable; conversely, in Section 4 the memory depends on the input size n, so σ is considered a constant that remains hidden in the asymptotic notation.

For $i = 1, 2, \ldots, \sigma$, let n_i be the number of occurrences of the i-th alphabet symbol in s. The *0-th order empirical entropy* of s is defined as $H_0(s) = -\sum_{i=1}^{\sigma}(n_i/|s|)\log(n_i/|s|)$ (throughout this paper we assume that all logarithms are taken to the base 2 and $0 \log 0 = 0$). It is well known that H_0 is the maximum compression we can achieve using a fixed codeword for each alphabet symbol. We can achieve a greater compression if the codeword we use for each symbol depends on the k symbols preceding it. In this case the maximum compression is bounded by the k-th order entropy $H_k(s)$ (see [6] for the formal definition). We use two properties of k-th order entropy in particular: $H_k(s_1)|s_1| + H_k(s_2)|s_2| \leq H_k(s_1 s_2)|s_1 s_2|$ and, since $H_0(s) \leq \log|\{a : a \text{ occurs in } s\}|$, we have $H_k(s) \leq \log \max_{|w|=k}\{j : w \text{ is followed by } j \text{ distinct characters in } s\}$.

We point out that the empirical entropy is defined *pointwise* for any string and can be used to measure the performance of compression algorithms as a function of the *string structure*, thus without any assumption on the input source. For

this reason we say that the bounds given in terms of H_k are *worst case* bounds. Another common approach in data compression is to assume that the input string is generated by a Markov source $\mathcal{X}$. To measure the effectiveness of a compression algorithm in this setting its *average* compression ratio is compared with the entropy of the source $H(\mathcal{X})$. We call this the *probabilistic* setting, and we consider it in Sect. 3.3.

Some of our arguments are based on Kolmogorov complexity [8]; the Kolmogorov complexity of s, denoted $K(s)$, is the length in bits of the shortest program that outputs s; it is generally incomputable but can be bounded from below by counting arguments (e.g., in a set of m elements, most have Kolmogorov complexity at least $\log m - O(1)$). We use two properties of Kolmogorov complexity in particular, as well: if an object can be easily computed from other objects, then its Kolmogorov complexity is at most the sum of theirs plus a constant; and a fixed, finite object has constant Kolmogorov complexity.

In this paper we consider space-conscious compressors, that is, algorithms that are allowed to use a limited amount of memory during their execution. We assume that the algorithms are one-pass in the sense that they are allowed to read each input symbol only once. Hence, if an algorithm needs to access (portions of) the input more than once it must store it—consuming part of its precious working memory. In Section 5 we briefly comment on the possibility of extending our results to multi-pass algorithms. Being space-conscious ourselves, most of the proofs are reported in the Appendix.

3 Compressing with memory independent of length

Move-to-front compression [2] is probably the best example of a compression algorithm whose space complexity is independent of the input length: keep a list of the characters that have occurred in decreasing order by recency; store each character in the input by outputting its position in the list (or, if it has not occurred before, its index in the alphabet) encoded in Elias' δ code, then move it to the front of the list. Move-to-front stores a string s of length n over an alphabet of size σ in $\big(H_0(s) + O(\log H_0(s))\big)\, n + O(\sigma \log \sigma)$ bits using one pass and $O(\sigma \log \sigma)$ bits of memory. Note that we can store s in $\big(H_k(s) + O(\log H_k(s))\big)\, n + O(\sigma^{k+1} \log \sigma)$ bits by keeping a separate list for each possible context of length k; this increases the memory usage by a factor of at most σ^k.

In this section we first use a more complicated algorithm to get a better upper bound: given constants $\lambda \geq 1$, $k \geq 0$ and $\mu > 0$, we can store s in $(\lambda H_k(s) + \mu)n + O(\sigma^{k+1/\lambda} \log \sigma)$ bits using one pass and $O(\sigma^{k+1/\lambda} \log^2 \sigma)$ bits of memory. We then prove that $\mu > 0$ is necessary and that we need to know k. We use the idea from these proofs to prove a nearly matching lower bound for compression: in the worst case it is impossible to store a string s of length n over an alphabet of size σ in $\lambda H_k(s)n + o(n \log \sigma) + g$ bits, for any function g independent of n, using one encoding pass and $O(\sigma^{k+1/\lambda-\epsilon})$ bits of memory. We prove a symmetric lower bound for decompression, and close with slightly weaker lower bounds for when the input comes from a stationary Markov source.

3.1 A nearly tight upper bound

The main drawback of move-to-front is the $O(\log H_0(s))$ in its analysis (or $O(\log H_k(s))$ using contexts of length k); we now show how we can replace this by any given constant $\mu > 0$. We start with the following lemma about storing an approximation Q of a probability distribution P in few bits, so that the relative entropy between P and Q is small. The relative entropy $D(P\|Q) = \sum_{i=1}^{\sigma} p_i \log(p_i/q_i)$ between $P = p_1, \ldots, p_\sigma$ and $Q = q_1, \ldots, q_\sigma$ is the expected redundancy per character of an ideal code for Q when characters are drawn according to P.

Lemma 1 ([3]). *Let s be a string of length n over an alphabet of size σ and let P be the normalized distribution of characters in s. Given s and constants $\lambda \geq 1$ and $\mu > 0$, we can store a probability distribution Q with $D(P\|Q) < (\lambda - 1)H(P) + \mu$ in $O(\sigma^{1/\lambda} \log(n + \sigma))$ bits using $O(\sigma^{1/\lambda} \log(n + \sigma))$ bits of memory.* □

Armed with this lemma, we adapt arithmetic coding [12] to use $O(\sigma^{1/\lambda} \log(n+\sigma))$ bits of memory with a specified redundancy per character:

Lemma 2. *Given a string s of length n over an alphabet of size σ and constants $\lambda \geq 1$ and $\mu > 0$, we can store s in $(\lambda H_0(s)+\mu)n+O(\sigma^{1/\lambda} \log(n+\sigma))$ bits using $O(\sigma^{1/\lambda} \log(n + \sigma))$ bits of memory.* □

We boost our space-conscious arithmetic coding algorithm to achieve a bound in terms of $H_k(s)$ instead of $H_0(s)$ by running a separate copy for each possible k-tuple, just as we boosted move-to-front compression:

Lemma 3. *Given a string s of length n over an alphabet of size σ and constants $\lambda \geq 1$, $k \geq 0$ and $\mu > 0$, we can store s in $(\lambda H_k(s)+\mu)n+O(\sigma^{k+1/\lambda} \log(n+\sigma))$ bits using $O(\sigma^{k+1/\lambda} \log(n + \sigma))$ bits of memory.* □

To make our algorithm use one pass and to change the $\log(n + \sigma)$ factor to $\log \sigma$, we process the input in blocks $s_1, \ldots, s_b$ of length $O(\sigma^{k+1/\lambda} \log \sigma)$. Notice each individual block s_i fits in memory — so we can apply Lemma 3 to it — and $\log(|s_i| + \sigma) = O(\log \sigma)$.

Theorem 1. *Given a string s of length n over an alphabet of size σ and constants $\lambda \geq 1$, $k \geq 0$ and $\mu > 0$, we can store s in $(\lambda H_k(s)+\mu)n+O(\sigma^{k+1/\lambda} \log \sigma)$ bits using one pass and $O(\sigma^{k+1/\lambda} \log^2 \sigma)$ bits of memory, and later recover s using one pass and the same amount of memory.* □

3.2 Lower bounds

Theorem 1 is still weaker than the strongest compression bounds that ignore memory constraints, in two important ways: first, even when $\lambda = 1$ the bound on the compression ratio does not approach $H_k(s)$ as n goes to infinity; second, we need to know k. It is not hard to prove these weaknesses are unavoidable when using fixed memory, as follows.

Lemma 4. *Let $\lambda \geq 1$ be a constant and let g be a function independent of n. In the worst case it is impossible to store a string s of length n in $\lambda H_0(s)n + o(n) + g$ bits using one encoding pass and memory independent of n.*

Proof. Let A be an algorithm that, given λ, stores s using one pass and memory independent of n. Since A's future output depends only on its state and its future input, we can model A with a finite-state machine M. While reading $|M|$ characters of s, M must visit some state at least twice; therefore either M outputs at least one bit for every $|M|$ characters in s — or $n/|M|$ bits in total — or for infinitely many strings M outputs nothing. If s is unary, however, then $H_0(s) = 0$. □

Lemma 5. *Let λ be a constant, let g be a function independent of n and let b be a function independent of n and k. In the worst case it is impossible to store a string s of length n over an alphabet of size σ in $\lambda H_k(s)n + o(n \log \sigma) + g$ bits for all $k \geq 0$ using one pass and b bits of memory.*

Proof. Let A be an algorithm that, given λ, g, b and σ, stores s using b bits of memory. Again, we can model it with a finite-state machine M, with $|M| = 2^b$ and M's Kolmogorov complexity $K(M) = K(\langle A, \lambda, g, b, \sigma \rangle) + O(1) = O(\log \sigma)$. (Since A, λ, g, and b are all fixed, their Kolmogorov complexities are $O(1)$.)

Suppose s is a periodic string with period $2b$ whose repeated substring r has $K(r) = |r| \log \sigma - O(1)$. We can specify r by specifying M, the states M is in when it reaches and leaves any copy of r in s, and M's output on that copy of r. (If there were another string r' that took M between those states with that output, then we could substitute r' for r in s without changing M's output.) Therefore M outputs at least

$$K(r) - K(M) - O(\log |M|) = |r| \log \sigma - O(\log \sigma + b) = \Omega(|r| \log \sigma)$$

bits for each copy of r in s, or $\Omega(n \log \sigma)$ bits in total. For $k \geq 2b$, however, $H_k(s)$ approaches 0 as n goes to infinity. □

The idea behind these proofs is simple — model a one-pass algorithm with a finite-state machine and evaluate its behaviour on a periodic string — but, combining it with the following simple results, we can easily show a lower bound that nearly matches Theorem 1. (In fact, our proofs are valid even for algorithms that make preliminary passes that produce no output — perhaps to gather statistics, like Huffman coding [4] — followed by a single encoding pass that produces all of the output; once the algorithm begins the encoding pass, we can model it with a finite-state machine.)

Lemma 6 ([3]). *Let $\lambda \geq 1$, $k \geq 0$ and $\epsilon > 0$ be constants and let r be a randomly chosen string of length $\lfloor \sigma^{k+1/\lambda-\epsilon} \rfloor$ over an alphabet of size σ. With high probability every possible k-tuple is followed by $O(\sigma^{1/\lambda-\epsilon})$ distinct characters in r.* □

Corollary 1. *Let $\lambda \geq 1$, $k \geq 0$ and $\epsilon > 0$ be constants. There exists a string r of length $\lfloor \sigma^{k+1/\lambda-\epsilon} \rfloor$ over an alphabet of size σ with $K(r) = |r| \log \sigma - O(1)$ but $H_k(r^i) \leq (1/\lambda - \epsilon) \log \sigma + O(1)$ for $i \geq 1$.* □

Consider what we get if, for some $\epsilon > 0$, we allow the algorithm A from Lemma 5 to use $O(\sigma^{k+1/\lambda-\epsilon})$ bits of memory, and evaluate it on the periodic string r^i from Corollary 1. Since r^i has period $\lfloor \sigma^{k+1/\lambda-\epsilon} \rfloor$ and its repeated substring r has $K(r) = |r| \log \sigma - O(1)$, the finite-state machine M outputs at least

$$K(r) - K(M) - O(\log |M|) = |r| \log \sigma - O(\sigma^{k+1/\lambda-\epsilon}) = |r| \log \sigma - O(|r|)$$

bits for each copy of r in r^i, or $n \log \sigma - O(n)$ bits in total. Because $\lambda H_k(r^i) \leq (1-\epsilon) \log \sigma + O(1)$, this yields the following nearly tight lower bound; notice it matches Theorem 1 except for a $\sigma^\epsilon \log^2 \sigma$ factor in the memory usage.

Theorem 2. *Let $\lambda \geq 1$, $k \geq 0$ and $\epsilon > 0$ be constants and let g be a function independent of n. In the worst case it is impossible to store a string s of length n over an alphabet of size σ in $\lambda H_k(s)n + o(n \log \sigma) + g$ bits using one encoding pass and $O(\sigma^{k+1/\lambda-\epsilon})$ bits of memory.* □

With a good bound on how much memory is needed for compression, we turn our attention to decompression. Good bounds here are equally important, because often data is compressed once by a powerful machine (e.g., a server or base-station) and then transmitted to many weaker machines (clients or agents) who decompress it individually. Fortunately for us, compression and decompression are essentially symmetric. Recall Theorem 1 says we can recover s from a $\big(\lambda H_k(s) + \mu)n + O(\sigma^{k+1/\lambda} \log \sigma)\big)$-bit encoding using one pass and $O(\sigma^{k+1/\lambda} \log^2 \sigma)$ bits of memory. Using the same idea about finite-state machines and periodic strings gives us the following nearly matching lower bound:

Theorem 3. *Let $\lambda \geq 1$, $k \geq 0$ and $\epsilon > 0$ be constants and let g be a function independent of n. There exists a string s of length n over an alphabet of size σ such that, given a $(\lambda H_k(s)n + o(n \log \sigma) + g)$-bit encoding of s, it is impossible to recover s using one pass and $O(\sigma^{k+1/\lambda-\epsilon})$ bits of memory.* □

3.3 Markov sources

As many classic analyses assume the data comes from a Markov source, we close this section with versions of Theorems 2 and 3 that have slightly weaker bounds on memory usage — we show $o(\sigma^k \log \sigma)$ bits of memory are insufficient, instead of $O(\sigma^{k+1/\lambda-\epsilon})$ — but apply when the data are drawn from a such a source. (All other things being equal, upper bounds are stronger when proven in terms of empirical entropy, without any assumptions about the source; conversely, lower bounds are stronger when they hold even with such assumptions.) The proofs of these theorems are slightly different and involve de Bruijn sequences; a σ-ary de Bruijn sequence of order k contains every possible k-tuple exactly once and, so, has length $\sigma^k + k - 1$. This property means every such sequence has kth-order empirical entropy 0 and, equivalently, can be generated by a deterministic kth-order Markov source. Rosenfeld [13] proved there are $(\sigma!)^{\sigma^{k-1}}$ such sequences so, by Stirling's formula, a randomly chosen one d has expected Kolmogorov complexity $\mathrm{E}[K(d)] = \log(\sigma!)^{\sigma^{k-1}} - O(1) = |d| \log \sigma - O(|d|)$.

Theorem 4. *Let $\lambda \geq 1$ and $k \geq 0$ be constants and let g be a function independent of n. There exists a stationary ergodic kth-order Markov source $\mathcal{X}$ over an alphabet of size σ such that, if we draw a string s of length n from $\mathcal{X}$, then with high probability it is impossible to store s in $\lambda H(\mathcal{X})n + o(n \log \sigma) + g$ bits using one pass and $o(\sigma^k \log \sigma)$ bits of memory.* □

Theorem 5. *Let $\lambda \geq 1$ and $k \geq 0$ be constants and let g be a function independent of n. There exists a stationary ergodic kth-order Markov source $\mathcal{X}$ over an alphabet of size σ such that, if we draw a string s of length n from $\mathcal{X}$, then with high probability it is impossible to recover s from a $(\lambda H(\mathcal{X})n + o(n \log \sigma) + g)$-bit encoding of s using one pass and $o(\sigma^k \log \sigma)$ bits of memory.* □

4 Compressing with (slowly) growing memory

In the previous section we have given upper and lower bounds to the amount of memory required to compress up to the k-th order entropy for a fixed k. It is well known that the best compressors, e.g. LZ77, LZ78, and BWT-based tools, are able to compress up to the k-th order entropy *for all* $k \geq 0$ *simultaneously*. That is, for any $k \geq 0$ and for any string s, their output is bounded by $\lambda H_k(s)n + g_k(n)$, with $g_k(n) \in o(n)$. Intuitively this means that these algorithms can take advantage of an "order-k regularity" for an arbitrarily large k. Unfortunately, Lemma 5 tells us that using memory indepedendent of n, it is impossible to compress up to $\lambda H_k(s)n$ for any $k \geq 0$.

For the above reasons, in this section we study compression algorithms in which the available memory grows with the size of the input. Given an increasing function f, we define the class C_f of one-pass compressors in which the working space grows according to f in the sense that when the algorithm has read t characters it is allowed to use a working space of size $\Theta(f(t))$ bits. Our first result shows that if $\lim_{t\to\infty} f(t) = \infty$ the algorithms in C_f can compress up to $\lambda H_k(s)n$ for any $k \geq 0$.

Lemma 7. *For any increasing and diverging function f there exists an algorithm in C_f achieving the compression ratio given in Theorem 1 for any $k \geq 0$.*

Proof. For a given k let n' be such that $f(n')$ is greater than the working space of the algorithm in Theorem 1. Consider now the procedure that outputs the first n' characters without compression and then executes the algorithm of Theorem 1. Since the space for the initial n' characters is just a constant overhead, for sufficiently long strings this procedure asymptotically achieves the space bound of Theorem 1 as claimed. □

The proof of Lemma 7 suggests that although any diverging working space suffices to get a compression ratio close to H_k for any $k \geq 0$, the rate of growth of the working space is likely to influence the rate of convergence, that is, the speed with which the compression ratio approaches the entropy. The quantitative study of this problem in the general setting appears to be a rather challenging task. In

the following we initiate this study by exploring the relationship between working space and rate of convergence for the important special case of the algorithm LZ77 with a growing sliding window.

4.1 Window size vs. convergence rate for LZ77

In the following we assume that the alphabet size is a constant (see comment at the beginning of Section 2). The LZ77 algorithm works by parsing the input string s into a sequence of words $w_1, w_2, \ldots, w_d$ and by encoding a compact representation of these words. For any non-empty string w let w^- denote the string w with the last character removed, and, if $|w| > 1$, let $w^{--} = (w^-)^-$. Assuming the words $w_1, w_2, \ldots, w_{i-1}$ have been already parsed, LZ77 selects the i-th word as the longest word w_i that can be obtained by adding a single character to a substring of $(w_1 w_2 \cdots w_i)^{--}$. Note that although this is a recursive definition there is no ambiguity. In fact, if $|w_i| > 1$ at least the first character of w_i belongs to $w_1 w_2 \cdots w_{i-1}$.

In the algorithm LZ77 with sliding window ($\mathsf{LZ77}_{sw}$ from now on) the word w_i is selected using a sliding window of size L_i, that is, w_i^- must be a substring of $(z_i w_i)^{--}$ where z_i is the length-L_i suffix of $w_1 w_2 \cdots w_{i-1}$. In practical implementations the sliding window length is usually fixed (for example it is equal to 2^{15} in gzip) but for our analysis we will consider a sliding window which grows with the size of the parsed string. Once w_i has been found, it is encoded with the triplet (p_i, ℓ_i, α_i), where p_i is the starting position of w_i^- in the sliding window, $\ell_i = |w_i|$, and α_i is the last character of w_i. In the following we assume that encoding p_i takes $\log L_i + O(1)$ bits, encoding ℓ_i takes[1] $\log \ell_i + O(\log\log \ell_i)$ bits, and encoding α_i takes $\log \sigma + O(1)$ bits[2]. If we store the already parsed portion of the input in a suffix tree, the algorithm LZ77 runs in linear time and uses a working space of $\Theta(n \log n)$ bits. The same result holds for $\mathsf{LZ77}_{sw}$ as well: the only difference is that we use a truncated suffix tree [7, 11] to maintain the sliding window so the working space is $\Theta(L \log L)$ bits, where L is the maximum size of the sliding window.

For the algorithm LZ77 we know (see [6, Th. 4.1]) that for any $k \geq 0$ and for any string s:

$$|\mathsf{LZ77}(s)| \leq H_k(s) n + O\left(n \frac{\log\log n}{\log n}\right) \tag{1}$$

which implies that the convergence rate is $O((n \log\log n)/\log n)$.

In the following we say that $\mathsf{LZ77}_{sw}$ uses an $f(t)$-size sliding window to denote that when t characters have been read, $\mathsf{LZ77}_{sw}$ maintains a sliding window of size $\lceil f(t) \rceil$ (hence, for $f(t) = t$ we have the original LZ77 algorithm). We prove that for $f(t) = t/\log^2 t$ the convergence rate is still $O((n \log\log n)/\log n)$, whereas for $f(t) = \log^{1-\epsilon} t$, with $0 < \epsilon < 1$, the convergence rate is $\Omega\big((n \log\log n)/\log^{1-\epsilon} n\big)$.

[1] Since we cannot bound in advance the size of ℓ_i, we are assuming we code it using Elias' δ code.

[2] Other encodings are possible but we believe our analysis can be adapted to all "reasonable" encodings.

To bound the convergence rate of $\mathsf{LZ77}_{sw}$ with a sliding window growing as $(t/\log^2 t)$, we first bound the number of times the same word can appear in the parsing of the input string.

Lemma 8. *Let $g(t)$ denote an increasing and diverging function. The $\mathsf{LZ77}_{sw}$ algorithm with window size $f(t) = t/g(t)$ produces a parsing of the input string in which the same word appears at most $O(g(n)\log(n))$ times.* □

The next lemma relates the number of words in the parsing with the k-th order entropy, and Lemma 10 gives an upper bound to the total number of words.

Lemma 9 ([6, Lemma 2.3]). *Let $y_1, \ldots, y_d$ denote a parsing of a string s in which each word y_i appears at most M times. For any $k \geq 0$ we have*

$$d \log d \leq |s| H_k(s) + d \log\left(\frac{|s|}{d}\right) + d \log M + \Theta(d)\,. \qquad \square$$

Lemma 10. *Let $y_1, \ldots, y_d$ denote a parsing of a string s in which each word y_i appears at most M times. We have $d = O(n/\log(n/M))$.* □

Theorem 6. *The algorithm $\mathsf{LZ77}_{sw}$ with a sliding window growing as $f(t) = (t/\log^2 t)$ produces an output bounded by $nH_k(s) + O((n \log\log n)/\log n)$.*

Proof. Let $w_1 \cdots w_d$ denote the $\mathsf{LZ77}_{sw}$ parsing of s. Recall that for each word w_i $\mathsf{LZ77}_{sw}$ outputs a triple (p_i, ℓ_i, α_i) whose encoding is described above. Using elementary calculus it is easy to show that if $\mathsf{LZ77}_{sw}$ parses s into d words, the output size is bounded by

$$|\mathsf{LZ77}_{sw}(s)| \leq d \log n + d \log(n/d) + O(d \log\log n)\,.$$

Recall that by Lemma 8 each word appears at most $O\big(\log^3 n\big)$ times in the parsing. Since $d \log n = d \log d + d \log(n/d)$, using Lemma 9 we get

$$\begin{aligned} |\mathsf{LZ77}_{sw}(s)| &\leq d \log d + 2n \log(n/d) + O(d \log\log n) \\ &\leq H_k(s) n + 3d \log(n/d) + O(d \log\log n)\ . \end{aligned}$$

Finally, by Lemma 10 we have $d = O(n/\log n)$, hence

$$|\mathsf{LZ77}_{sw}(s)| \leq nH_k(s) + O\left(\frac{n \log\log n}{\log n}\right)$$

as claimed. □

Now we show that the algorithm $\mathsf{LZ77}_{sw}$ with a sliding window of size $o(\log t)$ has a convergence rate $\omega(n \log\log n/\log n)$. To this end, for any ϵ, with $0 < \epsilon < 1$, we consider the $\mathsf{LZ77}_{sw}$ algorithm with a sliding window of size $f(t) = \log^{1-\epsilon}(t)$. Fix $n > 0$ and let $b = 1 + \lceil \log^{1-\epsilon}(n) \rceil$. Note that $b - 1$ is the maximum window size reached when compressing a string of length n. We define $\hat{s} = 0^j (10^{b-1})^h$ where $h = \lfloor n/b \rfloor$ and $j < b$ is such that $|\hat{s}| = n$.

Lemma 11. *We have* $H_0(\hat{s})n \leq (n/b)\log b + \Theta(n/b)$. □

Lemma 12. *Let* $\hat{s} = w_1 w_2 \dots w_d$ *denote the* $\mathsf{LZ77}_{sw}$ *parsing of* $\hat{s}$. *Then, if the word* w_i *contains the character 1, the word* w_{i+1} *contains only 0's.*

Proof. It is easy to see that if 1 appears in w_i it must be the last character of w_i. As a consequence, the next word will be $w_{i+1} = 0^\ell$ where ℓ is the current window size. □

Lemma 13. *For the* $\mathsf{LZ77}_{sw}$ *algorithm with a sliding window of size* $f(t) = \log^{1-\epsilon}(t)$ *we have*

$$|\mathsf{LZ77}_{sw}(\hat{s})| \geq 3(n/b)\log b - O(n/b).$$

Proof. Observe that when we have read $t \geq \sqrt{n}$ characters, the sliding window has size $\lceil \log^{1-\epsilon} t \rceil \geq ((\log n)/2)^{1-\epsilon}$. From that point on, encoding a position in the sliding window—which must be done for each word in the parsing—takes at least

$$(1-\epsilon)\log((\log n)/2) = (1-\epsilon)\log\log n - (1-\epsilon) \geq \log b - 2$$

bits. In addition, by the proof of Lemma 12 we see that each other word will have length equal to the window size; encoding each one of these lengths will again cost at least $\log b - 2$ bits. By Lemma 12, after we have read $\sqrt{n}$ characters there are still $2(n-\sqrt{n})/b$ words to be parsed. The above observations imply that their encoding takes at least $3(n/b)\log b - O(n/b)$ bits. □

Theorem 7. *For the* $\mathsf{LZ77}_{sw}$ *algorithm with a sliding window growing as* $f(t) = \log^{1-\epsilon}(t)$, *we can build an arbitrarily long string* $\hat{s}$ *such that*

$$|\mathsf{LZ77}_{sw}(\hat{s})| \geq H_0(\hat{s})n + \Omega\big((n\,\log\log n)/\log^{1-\epsilon} n\big)\,.$$

Proof. By Lemmas 11 and 13 we have

$$|\mathsf{LZ77}_{sw}(\hat{s})| - H_0(\hat{s})n \geq \frac{2n\log b}{b} - O\Big(\frac{n}{b}\Big) = \frac{2n\,\log\log n}{\log^{1-\epsilon} n} - O\bigg(\frac{n}{\log^{1-\epsilon} n}\bigg)\,. \quad \square$$

Comparing Theorems 6 and 7 we see that the penalty we pay for using a smaller window is a slower convergence rate. Further work is needed to narrow the huge gap between the rate of growth of the sliding window in the two theorems. In particular, it would be interesting to determine the smallest rate of growth that guarantees an output size bounded by $H_k(s)n + O((n\,\log\log n)/\log n)$ as in (1).

5 Future work

We plan to generalize the results in Section 3 to multipass algorithms. Munro and Paterson [9] introduced a model for multipass algorithms in which the data is "stored on a one-way read-only tape. [...] Initially the storage is empty and the

tape is placed with the reading head at the beginning. After each pass the tape is rewound to this position with no reading permitted." Among other things, they proved sorting a set of n distinct elements in p passes takes $\Theta(n/p)$ memory locations (each of which can hold a single element).

It seems we can modify the algorithm in Theorem 1 so that, allowed p passes, during each pass it processes only those character following a $(1/p)$-fraction of the possible contexts; any character following a different context is ignored. This way, the algorithm might need only a $(1/p)$-fraction as much memory during each pass. On the other hand, consider a p-pass compression algorithm compressing the string r^i from Corollary 1: we can specify r by specifying the algorithm, the algorithm's memory configurations when it enters and leaves a particular copy of r in r^i during each pass (i.e., $2p$ configurations in all), and its output while reading that copy during each pass. Thus, allowing the algorithm in the proof of Theorem 2 to use p passes but only $O((1/p) \cdot \sigma^{k+1/\lambda-\epsilon})$ bits of memory seems not to affect the proof.

References

1. B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proceedings of the 21st Symposium on Principles of Database Systems*, pages 1–16, 2002.
2. J. L. Bentley, D. D. Sleator, R. E. Tarjan, and V. K. Wei. A locally adaptive data compression scheme. *Communications of the ACM*, 29:320–330, 1986.
3. T. Gagie. Large alphabets and incompressibility. *Information Processing Letters*, 99:246–251, 2006.
4. D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40:1098–1101, 1952.
5. R. Karp, S. Shenker, and C. Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *ACM Trans. Database Syst.*, 28(1):51–55, 2003.
6. R. Kosaraju and G. Manzini. Compression of low entropy strings with Lempel–Ziv algorithms. *SIAM Journal on Computing*, 29(3):893–911, 1999.
7. N. J. Larsson. Extended application of suffix trees to data compression. In *DCC '96: Proceedings of the Conference on Data Compression*, page 190, Washington, DC, USA, 1996. IEEE Computer Society.
8. M. Li and P. Vitányi. *An Introduction to Kolmogorov Complexity and Its Applications*. Springer-Verlag, 2nd edition, 1997.
9. J. I. Munro and M. S. Paterson. Selection and sorting with limited storage. *Theoretical Computer Science*, 12:315–323, 1980.
10. S. Muthukrishnan. *Data Streams: Algorithms and Applications*. Now Publishers, 2005. See also: http://www.nowpublishers.com/tcs/.
11. J. C. Na, A. Apostolico, C. Iliopoulos, and K. Park. Truncated suffix trees and their application to data compression. *Theor. Comput. Sci.*, 304(1-3):87–101, 2003.
12. J. Rissanen. Generalized Kraft inequality and arithmetic coding. *IBM Journal of Research and Development*, 20:198–203, 1976.
13. V. R. Rosenfeld. Enumerating De Bruijn sequences. *MATCH Communications in Mathematical and in Computer Chemistry*, 45:71–83, 2002.
14. A. J. Wyner. The redundancy and distribution of the phrase lengths of the fixed-database LempelZiv algorithm. *IEEE Transactions on Information Theory*, 43:1452–1464, 1997.

A Appendix

A.1 Proof of Lemmas 1, 2 and 3 and Theorem 1

Lemma 1. *Let s be a string of length n over an alphabet of size σ and let P be the normalized distribution of characters in s. Given s and constants $\lambda \geq 1$ and $\mu > 0$, we can store a probability distribution Q with $D(P\|Q) < (\lambda-1)H(P)+\mu$ in $O(\sigma^{1/\lambda}\log(n+\sigma))$ bits using $O(\sigma^{1/\lambda}\log(n+\sigma))$ bits of memory.*

Proof. Suppose $P = p_1, \ldots, p_\sigma$. We can use an $O(n)$-time algorithm due to Karp, Papadimitriou and Shenker [5] to find the $t \leq r\sigma^{1/\lambda}$ values of i such that $p_i \geq 1/(r\sigma^{1/\lambda})$, where $r = 1 + \frac{1}{2^{\mu/2}-1}$, using $O(\sigma^{1/\lambda}\log\max(n,\sigma))$ bits of memory; or, since we are not concerned with time in this paper, we can simply make σ passes over s to find these t values. For each, we store i and $\lfloor p_i r^2\sigma \rfloor$; since r depends only on μ, in total this takes $O(\sigma^{1/\lambda}\log\sigma)$ bits. (As an aside, since $r < 1 + 2/(\mu \ln 2)$, if we were treating μ as a variable, the bound would be $O(\log(1/\mu)\cdot\sigma^{1/\lambda}\log\sigma)$.) This information lets us later recover $Q = q_1, \ldots, q_\sigma$ where

$$q_i = \begin{cases} \dfrac{(1-1/r)\lfloor p_i r^2\sigma\rfloor}{\sum\left\{\lfloor p_j r^2\sigma\rfloor \,:\, p_j \geq 1/(r\sigma^{1/\lambda})\right\}} & \text{if } p_i \geq 1/(r\sigma^{1/\lambda}), \\ \dfrac{1}{r(\sigma-t)} & \text{otherwise.} \end{cases}$$

Suppose $p_i \geq 1/(r\sigma^{1/\lambda})$; then $p_i r^2\sigma \geq r$. Since $\sum\left\{\lfloor p_j r^2\sigma\rfloor \,:\, p_j \geq 1/(r\sigma^{1/\lambda})\right\} \leq r^2\sigma$,

$$\begin{aligned} & p_i \log(p_i/q_i) \\ & \leq p_i \log\left(\frac{r}{r-1}\cdot\frac{p_i r^2\sigma}{\lfloor p_i r^2\sigma\rfloor}\right) \\ & < 2p_i \log\frac{r}{r-1} \\ & = p_i\mu\ . \end{aligned}$$

Now suppose $p_i < 1/(r\sigma^{1/\lambda})$; then $p_i\log(1/p_i) > (p_i/\lambda)\log\sigma$. Therefore

$$\begin{aligned} & p_i \log(p_i/q_i) \\ & < p_i \log\left((\sigma-t)/\sigma^{1/\lambda}\right) \\ & \leq (\lambda-1)(p_i/\lambda)\log\sigma \\ & < (\lambda-1)p_i\log(1/p_i)\ . \end{aligned}$$

Since $p_i\log(p_i/q_i) < (\lambda-1)p_i\log(1/p_i) + p_i\mu$ in both cases, $D(P\|Q) < (\lambda-1)H(P)+\mu$. □

Lemma 2. *Given a string s of length n over an alphabet of size σ and constants $\lambda \geq 1$ and $\mu > 0$, we can store s in $(\lambda H_0(s) + \mu)n + O(\sigma^{1/\lambda} \log(n + \sigma))$ bits using $O(\sigma^{1/\lambda} \log(n + \sigma))$ bits of memory.*

Proof. Let P be the normalized distribution of characters in s, so $H(P) = H_0(s)$. First, as described in Lemma 1, we store a probability distribution Q with $D(P\|Q) < (\lambda - 1)H(P) + \mu/2$ in $O(\sigma^{1/\lambda} \log \sigma)$ bits using $O(\sigma^{1/\lambda} \log(n + \lambda))$ bits of memory. Then, we process s in blocks $s_1, \ldots, s_b$ of length $\lceil 4/\mu \rceil$ (except s_b may be shorter). For $1 \leq i < b$, we store s_i as the first $\lceil \log(2/\Pr[X = s_i]) \rceil$ bits to the right of the binary point in the binary representation of

$$\begin{aligned} f(s_i) &= \Pr[X < s_i] + \Pr[X = s_i]/2 \\ &= \sum_{j=1}^{\lceil 4/\mu \rceil} \Pr\Big[X[1] = s_i[1], \ldots, X[j-1] = s_i[j-1], X[j] < s_i[j]\Big] \\ &\qquad + \Pr[X = s_i]/2 \ , \end{aligned}$$

where X is a string of length $\lceil 4/\mu \rceil$ chosen randomly according to Q, $X < s_i$ means X is lexicographically less than s_i, and $X[j]$ and $s_i[j]$ indicate the indices in the alphabet of the jth characters of X and s_i, respectively. Notice that, since $|f(s_i) - f(y)| > \Pr[X = s_i]/2$ for any string $y \neq s_i$ of length $\lceil 4/\mu \rceil$, these bits uniquely identify $f(s_i)$ and, thus, s_i. Also, since the probabilities in Q are $O(\log \sigma)$-bit numbers, we can compute $f(s_i)$ from s_i with $O(\sigma)$ additions and $O(1/\mu) = O(1)$ multiplications using $O(\log \sigma)$ bits of memory. (In fact, with appropriate data structures, $O(\log \sigma)$ additions and $O(1)$ multiplications suffice.) Finally, we store s_b in $|s_b| \lceil \log \sigma \rceil = O(\log \sigma)$ bits. In total we store s in

$$\begin{aligned} &\sum_{i=1}^{b-1} \lceil \log(2/\Pr[X = s_i]) \rceil + O(\sigma^{1/\lambda} \log \sigma) \\ &\leq \sum_{i=1}^{b-1} \left(\sum_{j=1}^{\lceil 4/\mu \rceil} \log(1/q_{s_i[j]}) + 2 \right) + O(\sigma^{1/\lambda} \log \sigma) \\ &= n \sum_{i=1}^{\sigma} p_i \log(1/q_i) + 2(b-1) + O(\sigma^{1/\lambda} \log \sigma) \\ &\leq n(D(P\|Q) + H(P)) + \mu n/2 + O(\sigma^{1/\lambda} \log \sigma) \\ &\leq (\lambda H_0(s) + \mu)n + O(\sigma^{1/\lambda} \log \sigma) \end{aligned}$$

bits using $O(\sigma^{1/\lambda} \log(n + \sigma))$ bits of memory. □

Lemma 3. *Given a string s of length n over an alphabet of size σ and constants $\lambda \geq 1$, $k \geq 0$ and $\mu > 0$, we can store s in $(\lambda H_k(s) + \mu)n + O(\sigma^{k+1/\lambda} \log(n + \sigma))$ bits using $O(\sigma^{k+1/\lambda} \log(n + \sigma))$ bits of memory.*

Proof. We store the first k characters of s in $O(\log \sigma)$ bits then apply Lemma 2 to subsequences $s_1, \ldots, s_{\sigma^k}$, where s_i consists of the characters in s that immediately follow occurrences of the lexicographically ith possible k-tuple. Notice that although we cannot keep $s_1, \ldots, s_{\sigma^k}$ in memory, enumerating them as many times as necessary in order to apply Lemma 2 takes $O(\log \sigma)$ bits of memory. □

Theorem 1. *Given a string s of length n over an alphabet of size σ and constants $\lambda \geq 1$, $k \geq 0$ and $\mu > 0$, we can store s in $(\lambda H_k(s) + \mu)n + O(\sigma^{k+1/\lambda} \log \sigma)$ bits using one pass and $O(\sigma^{k+1/\lambda} \log^2 \sigma)$ bits of memory, and later recover s using one pass and the same amount of memory.*

Proof. Let c be a constant such that, by Lemma 3, we can store any substring s_i of s in $(\lambda H_k(s_i) + \mu/2)|s_i| + c\sigma^{k+1/\lambda} \log \sigma$ bits using $O(\sigma^{1/\lambda} \log(|s_i| + \sigma))$ bits of memory. We process s in blocks $s_1, \ldots, s_b$ of length $\lceil (2c/\mu)\sigma^{k+1/\lambda} \log \sigma \rceil$ (except s_b may be shorter). Notice each block s_i fits in $O(\sigma^{k+1/\lambda} \log^2 \sigma)$ bits of memory. When we reach s_i, we read it into memory, apply Lemma 3 to it — using

$$O\left(\sigma^{k+1/\lambda} \log\left(\lceil (2c/\mu)\sigma^{k+1/\lambda} \log \sigma \rceil + \sigma\right)\right) = O(\sigma^{k+1/\lambda} \log \sigma)$$

bits of memory — then erase it from memory. In total we store s in

$$\begin{aligned}
&\sum_{i=1}^{b} \left((\lambda H_k(s_i) + \mu/2)|s_i| + c\sigma^{k+1/\lambda} \log \sigma\right) \\
&\leq (\lambda H_k(s) + \mu/2)n + bc\sigma^{k+1/\lambda} \log \sigma \\
&\leq (\lambda H_k(s) + \mu)n + c\sigma^{k+1/\lambda} \log \sigma
\end{aligned}$$

bits using $O(\sigma^{k+1/\lambda} \log^2 \sigma)$ bits of memory.

Notice the encoding of each block s_i also fits in $O(\sigma^{k+1/\lambda} \log^2 \sigma)$ bits of memory. To decode each block later, we read its encoding into memory, search through all possible strings of length $\lceil (2c/\mu)\sigma^{k+1/\lambda} \log \sigma \rceil$ in lexicographic order until we find the one that yields that encoding — using $O(\sigma^{k+1/\lambda} \log^2 \sigma)$ bits of memory — and output it. □

The method for decompression in the proof of Theorem 1 above takes exponential time but is very simple (recall we are not concerned with time here); reversing each step of the compression takes linear time but is slightly more complicated.

A.2 Proofs of Lemma 6, Corollary 1 and Theorems 2 and 3

Lemma 6. *Let $\lambda \geq 1$, $k \geq 0$ and $\epsilon > 0$ be constants and let s be a random string of length $\lfloor \sigma^{k+1/\lambda-\epsilon} \rfloor$ over an alphabet of size σ. With high probability every possible k-tuple is followed by $O(\sigma^{1/\lambda-\epsilon})$ distinct characters in s.*

Proof. Consider a k-tuple w. For $1 \leq i \leq n-k$, let $X_i = 1$ if the ith through $(i+k-1)$st characters of s are an occurrence of w and the $(i+k)$th character in s does not occur in w; otherwise $X_i = 0$. Notice w is followed by at most $\sum_{i=1}^{n-k} X_i + k$ distinct characters in s and $\Pr[X_i = 1 \,|\, X_j = 1] \leq 1/\sigma^k$ and $\Pr[X_i = 1 \,|\, X_j = 0] \leq 1/(\sigma^k - 1)$ for $i \neq j$. Therefore, by Chernoff bounds and the union bound, with probability greater than

$$1 - \frac{\sigma^k}{2^{6\lfloor \sigma^{k+1/\lambda-\epsilon} \rfloor / (\sigma^k - 1)}} \geq 1 - \sigma^k / 2^{6\sigma^{1/\lambda-\epsilon}}$$

every k-tuple is followed by fewer than $6\lfloor \sigma^{k+1/\lambda-\epsilon} \rfloor / (\sigma^k - 1) + k \leq 12\sigma^{1/\lambda-\epsilon} + k$ distinct characters. □

Corollary 1. *Let $\lambda \geq 1$, $k \geq 0$ and $\epsilon > 0$ be constants. There exists a string s of length $\lfloor \sigma^{k+1/\lambda-\epsilon} \rfloor$ over an alphabet of size σ with $K(s) \geq |s| \log \sigma - O(1)$ but $H_k(s^i) \leq (1/\lambda - \epsilon) \log \sigma + O(1)$ for $i \geq 1$.*

Proof. If s is randomly chosen, then $K(s) \geq |s| \log \sigma - 1$ with probability greater than $1/2$ and, by Lemma 6, with high probability every possible k-tuple is followed by $O(\sigma^{1/\lambda-\epsilon})$ distinct characters in s; therefore there exists an s with both properties. Every possible k-tuple is followed by at most k more distinct characters in s^i than in s and, thus,

$$\begin{aligned} H_k(s^i) &\leq \log \max_{|w|=k} \left\{ j : \begin{array}{l} w \text{ is followed by } j \\ \text{distinct characters in } s^i \end{array} \right\} \\ &\leq \log O(\sigma^{1/\lambda-\epsilon}) \\ &\leq (1/\lambda - \epsilon) \log \sigma + O(1) \ . \end{aligned}$$

□

Theorem 2. *Let $\lambda \geq 1$, $k \geq 0$ and $\epsilon > 0$ be constants and let g be a function independent of n. In the worst case it is impossible to store a string s of length n over an alphabet of size σ in $\lambda H_k(s) n + o(n \log \sigma) + g$ bits using one encoding pass and $O(\sigma^{k+1/\lambda-\epsilon})$ bits of memory.*

Proof. Let A be an algorithm that, given λ, k, ϵ and σ, stores s while using one encoding pass and $O(\sigma^{k+1/\lambda-\epsilon})$ bits of memory; we prove that in the worst case A stores s in more than $(\lambda H_k(s) + \mu)n + o(n \log \sigma) + g$ bits. Again, we can model it with a finite-state machine M, with $|M| = 2^{O(\sigma^{k+1/\lambda-\epsilon})}$ and $K(M) = O(\log \sigma)$. Let r be a string of length $\lfloor \sigma^{k+1/\lambda-\epsilon} \rfloor$ with $K(r) \geq |r| \log \sigma - O(1)$ and $H_k(x^i) \leq (1/\lambda - \epsilon) \log \sigma + O(1)$ for $i \geq 1$, as described in Corollary 1, and suppose $s = r^i$ for some i. We can specify r by specifying M, the states M is in when it reaches and leaves any copy of r in s, and M's output on that copy. Therefore M outputs at least

$$K(r) - K(M) - O(\sigma^{k+1/\lambda-\epsilon}) = |r| \log \sigma - O(|r|)$$

bits for each copy of r in s, or $n \log \sigma - O(n)$ bits in total — which is asymptotically greater than $\lambda H_k(s)n + o(n \log \sigma) + g \leq (1-\epsilon)n \log \sigma + o(n \log \sigma) + g$. □

Theorem 3. *Let $\lambda \geq 1$, $k \geq 0$ and $\epsilon > 0$ be constants and let g be a function independent of n. There exists a string s of length n over an alphabet of size σ such that, given a $(\lambda H_k(s)n + o(n \log \sigma) + g)$-bit encoding of s, it is impossible to recover s using one pass and $O(\sigma^{k+1/\lambda-\epsilon})$ bits of memory.*

Proof. Let r be a string of length $\lfloor \sigma^{k+1/\lambda-\epsilon} \rfloor$ with $K(r) = |r| \log \sigma - O(1)$ but $H_k(r^i) \leq (1/\lambda - \epsilon) \log \sigma + O(1)$ for $i \geq 1$, as described in Corollary 1, and suppose $s = r^i$ for some i. Let A be an algorithm that, given λ, k, ϵ, σ and a $(\lambda H_k(s)n + o(n \log \sigma) + g)$-bit encoding of s, recovers s using one pass; we prove A uses $\omega(\sigma^{k+1/\lambda-\epsilon})$ bits of memory. Again, we can model A with a finite-state machine M, with $\log |M|$ equal to the number of bits of memory A uses and $K(M) = O(\log \sigma)$. We can specify r by specifying M, the state M is in when it starts outputting any copy of r in s, and the bits of the encoding it reads while outputting that copy of r; therefore

$$\begin{aligned} K(r) &\leq K(M) + O(\log |M|) + \left(\lambda H_k(s)n + o(n \log \sigma) + g\right)/i \\ &\leq O(\log \sigma) + O(\log |M|) + |r| \left((1-\epsilon) \log \sigma + o(\log \sigma) + g/n\right) \\ &\leq (1-\epsilon)|r| \log \sigma + o(|r| \log \sigma) + O(\log |M|) + g/n \ , \end{aligned}$$

so

$$O(\log |M|) + g/n \geq \epsilon |r| \log \sigma - o(|r| \log \sigma) = \Omega(\sigma^{k+1/\lambda-\epsilon} \log \sigma) \ .$$

The theorem follows because n can be arbitrarily large compared to g. □

A.3 Proofs of Theorems 4 and 5

Before proving Theorems 4 and 5 we prove two preliminary lemmas, somewhat similar to Lemma 6 and Corollary 1.

Lemma 14. *Let $k \geq 0$ be a constant. There exists a string s of length σ^k over an alphabet of size σ with $K(s) \geq |s| \log \sigma - O(|s|)$ but $H_k(s^i) = 0$ for $i \geq 1$.*

Proof. If s consists of the first σ^k characters of a σ-ary de Bruijn sequence of order k, then s followed by the first $k-1$ characters in s contains every possible k-tuple exactly once and, thus, $H_k(s^i) = 0$ for $i \geq 1$. There are $(\sigma!)^{\sigma^{k-1}}$ such de Bruijn sequences, however [13], so there exists such an s with $K(s) \geq \log(\sigma!)^{\sigma^{k-1}}$ — which, by Stirling's Formula, is at least $|s| \log \sigma - O(|s|)$. □

Lemma 15. *Let $\lambda \geq 1$ and $k \geq 0$ be constants. There exists a stationary ergodic kth-order Markov source $\mathcal{X}$ over an alphabet of size σ with $H(\mathcal{X}) < (\log \sigma)/(4\lambda) + 1$ such that, if we draw a string s of length n from $\mathcal{X}$ and divide it into blocks $s_1, \ldots, s_b$ of length σ^k, then with high probability $K(s_i) \geq |s_i| \log \sigma - O(|s_i|)$ for at least $b/3$ values of i.*

Proof. Let d be a string of length σ^k with $K(d) \geq |d| \log \sigma - O(|d|)$ but $H_k(d^i) = 0$ for $i \geq 1$, as described in Lemma 14. Let z be a constant with $0 < z \leq 1/(4\lambda)$ and $(1-z)^{\sigma^k} \geq 1/2$. Let $\mathcal{X}$ be the kth-order Markov source that, after emitting a k-tuple w, emits the unique character that follows w in d^2 with probability $1-(1-1/\sigma)z$ and emits each other character with probability z/σ. Notice that, since with probability $z^k > 0$ the next k characters are chosen randomly and independently of preceding characters, $\mathcal{X}$ is stationary and ergodic. Also $\mathcal{X}$ is equivalent to the following process: we flip a coin biased to show heads with probability $1-z$ and tails with probability z; on heads, we choose the unique character that follows w in d^2; on tails, we choose the next character randomly. Therefore $H(\mathcal{X}) = H(1-z, z) + zH(1/\sigma, \ldots, 1/\sigma) < (\log \sigma)/(4\lambda) + 1$. We see σ^k consecutive heads with probability $(1-z)^{\sigma^k} \geq 1/2$, so each block s_i is a cyclic shift of d with probability at least $1/2$; therefore, by Chernoff bounds, $K(s_i) \geq |s_i| \log \sigma - O(|s_i|)$ for at least $b/3$ values of i with probability at least $1 - \exp(-b/36)$. □

Proof of Theorem 4. Let $\mathcal{X}$ be a Markov source as described in Lemma 15 and let A be an algorithm that, given σ, λ, k and ϵ, stores s while using one encoding pass and $o(\sigma^k \log \sigma)$ bits of memory; we prove that with high probability A stores s in more than $\lambda H(\mathcal{X}) + o(n \log \sigma) + g$ bits. Again, we can model A with a finite-state machine, with $|M| = 2^{o(\sigma^k \log \sigma)}$ states and $K(M) = O(\log \sigma)$. Consider s in blocks $s_1, \ldots, s_b$ of length σ^k (except s_b may be shorter). As for r in the proof of Theorem 2, we can specify a block s_i by specifying M, the states M is in when it reaches and leaves s_i in s, and M's output on s_i. Therefore, for each s_i with $K(s_i) \geq |s_i| \log \sigma - O(|s_i|)$, M outputs at least

$$K(s_i) - K(M) - o(\sigma^k \log \sigma) = |s_i| \log \sigma - o(|s_i| \log \sigma)$$

bits, or with high probability at least $(n \log \sigma)/3 - o(n \log \sigma)$ bits in total — which is asymptotically greater than $\lambda H(\mathcal{X}) n + o(n \log \sigma) + g \leq (n \log \sigma)/4 + o(n \log \sigma) + g$. □

Proof of Theorem 5. Let $\mathcal{X}$ be a Markov source as described in Lemma 15 and let A be an algorithm that, given σ, λ, k, ϵ and a $(\lambda H(\mathcal{X}) n + o(n \log \sigma) + g)$-bit encoding of s, recovers s using one pass; we prove A uses $\Omega(\sigma^k \log \sigma)$ bits of memory. Again, we can model A with a finite-state machine M, with $\log |M|$ equal to the number of bits of memory A uses and $K(M) = O(\log \sigma)$. As in the proof of Theorem 4, consider s in blocks $s_1, \ldots, s_b$ of length σ^k (except s_b may be shorter); we can specify s_i by specifying M, the state M is in when it starts outputting s_i, and the bits of the encoding it reads while outputting s_i. Therefore

$$\sum_{i=1}^{b} K(s_i) \leq \big(K(M) + O(\log |M|)\big)\, b + \lambda H(\mathcal{X}) n + o(n \log \sigma) + g \ ,$$

so

$$(n \log \sigma)/3 - O(n) \leq (n \log \sigma)/4 + o(n \log \sigma) + g + O\left((n \log |M|)/\sigma^k\right)$$

and $\log |M| = \Omega(\sigma^k \log \sigma)$. □

A.4 Proofs of lemmas in Section 4

Proof of Lemma 8. Assume $\mathsf{LZ77}_{sw}$ outputs the word w exactly k times and let $p_1, p_2, \ldots, p_k$ denote the starting positions of such occurrences. Note that for $i > 1$, $\mathsf{LZ77}_{sw}$ can output the word w at position p_i only if the previous occurrence of w is outside the current sliding window. This means when the algorithm reaches position p_i, p_{i-1} must be outside the sliding window. Since the sliding window at p_i has size $p_i/g(p_i)$, we must have $p_i - p_{i-1} > p_i/g(p_i)$ or, equivalently: $p_i \left(1 - \frac{1}{g(p_i)}\right) > p_{i-1}$. Applying the above inequality for $i = 2, \ldots, k$ and using the fact that $g(p_i) < g(n)$ we get $p_k \left(1 - \frac{1}{g(n)}\right)^{k-1} > p_1$. Since $p_1 \geq 1$, and $p_k < n$ we have $n \left(1 - \frac{1}{g(n)}\right)^{k-1} > 1$ and taking the logarithm

$$\log n + (k-1) \log \left(1 - \frac{1}{g(n)}\right) > 0.$$

Since for vanishing ϵ, $\log(1-\epsilon) = -\epsilon + O(\epsilon^2)$, we get $(k-1) \leq g(n) \log n + o(\log n)$ an the thesis follows. □

Proof of Lemma 10. It is well known that if the words in the parsing are distinct then $d = O(n/\log n)$. Assuming that each word is repeated exactly M times, the total number of words is roughly M times the number of distinct words used for a parsing of a string of length n/M, that is $O(M(n/M \log(n/M))) = O(n/\log(n/M))$. □

Proof of Lemma 11. We have

$$\begin{aligned} nH_0(\hat{s}) &\leq (n/b) \log b + n(\tfrac{b-1}{b}) \log(b/(b-1)) \\ &= (n/b)[\log b + (b-1) \log(1 + \tfrac{1}{b-1})] = (n/b) \log b + \Theta(n/b) \end{aligned}$$

where the last equality holds since the term $(b-1) \log(1 + \frac{1}{b-1})$ is bounded by $\log e$. □