Deplump for Streaming Data

Nicholas Bartlett Frank Wood

Department of Statistics, Columbia University, New York, USA

Abstract

We present a general-purpose, lossless compressor for streaming data. This compressor is based on the deplump probabilistic compressor for batch data. Approximations to the inference procedure used in the probabilistic model underpinning deplump are introduced that yield the computational asyptotics necessary for stream compression. We demonstrate the performance of this streaming deplump variant relative to the batch compressor on a benchmark corpus and find that it performs equivalently well despite these approximations. We also explore the performance of the streaming variant on corpora that are too large to be compressed by batch deplump and demonstrate excellent compression performance.

1 Introduction

Deplump [Gasthaus et al., 2010] is a general purpose, lossless, batch compressor based on a probabilistic model of discrete sequences called the sequence memoizer [Wood et al., 2009]. Gasthaus et al. showed that although deplump is algorithmically similar to the PPM and CTW compression algorithms, particularly their unbounded context lengths variants [Cleary and Teahan, 1997; Willems, 1998], the coherent probabilistic model underlying deplump gives it a consistent empirical advantage. In particular, Gasthaus et al. showed that deplump generally outperformed CTW [Willems, 2009], PPMZ [Peltola and Tarhio, 2002], and PPM* [Cleary and Teahan, 1997] on the large Calgary corpus, the Canterbury corpus, Wikipedia, and Chinese text. Deplump was shown to underperform in comparison to the PAQ family of compressors [Mahoney, 2005], but the point was made that deplump (or more specifically the sequence memoizer) could replace one or all of the mixed, finite-order Markov-model predictors included in PAQ.

When introduced in [Gasthaus et al., 2010], deplump was reposed on a sequence memoizer [Wood et al., 2009] whose space complexity was linear in the length of the input stream, rendering deplump inappropriate for stream compression. Since then, two complimentary approximations to inference in the sequence memoizer have emerged that, when combined as they are in this paper, together result in a constant space sequence memoizer and thus a stream-capable compressor. To review: the sequence memoizer is an incremental method for estimating the conditional distributions in an n-gram model in the limit of $n \to \infty$. The space complexity of the sequence memoizer is a function of the number of instantiated nodes in the corresponding suffix-tree-shaped graphical model and the storage required at each node. Bartlett et al. introduced an approximation to the sequence memoizer in which the number of nodes in the tree remains of asymptotically constant order [Bartlett et al., 2010]. Unfortunately, in that work the storage requirement at each node grew as an uncharacterized but not-constant function of the input sequence length. Gasthaus and Teh later introduced a method for constraining the memory used at each node in the tree to be of constant order [Gasthaus and Teh, 2011] but did so in a way that requires the computational cost of inference to grow as a super-linear function of the length of the training sequence.

The primary contribution of this work is to marry these two approximations such that constant memory, linear time approximate inference in the sequence memoizer is achieved, thereby rendering deplump appropriate for streaming lossless compression applications. In addition to combining these two approximations, a third and final approximation is introduced to constrain the computation required by the model representation introduced in [Gasthaus and Teh, 2011]. As the asymptotic statistical characteristics of the combined approximations are difficult to mathematically characterize, we include significant experimental exploration of the approximation parameter space and the effect on compression performance. Additionally, as the resulting deplump compression algorithm is of fairly high implementation complexity, we have included a nearly comprehensive algorithmic description of it in this paper. This accompanies a reference implementation which can be explored at `http://www.deplump.com/`.

2 Methodology

In this section we first review the sequence memoizer model and the inference algorithm used in the batch deplump compressor. We then review two approximations to this inference algorithm that we combine for the first time in this paper. Taken together the two approximations ensure asymptotically constant model storage complexity. A third approximation, novel to this paper, ensures that the asymptotic computational complexity of the inference algorithm is linear in the length of the input sequence.

2.1 Review

Note that a distribution P over sequences can be factorized as $P(S = [s_0, s_1, \ldots, s_m]) = P(s_0)P_{[s_0]}(s_1)P_{[s_0,s_1]}(s_2)\ldots P_{[s_0,s_1,\ldots,s_{m-1}]}(s_m)$, where $P_{\mathcal{U}}(s) = P(s|\mathcal{U})$. The sequence memoizer [Wood et al., 2009] jointly models these conditional distributions using a hierarchical Bayesian framework in which non-negative, integer parameters $\{c^{\mathcal{U}}_\sigma, t^{\mathcal{U}}_\sigma\}_{\sigma\in\Sigma,\mathcal{U}\in\Sigma^+}$ are used to characterize each $P_{\mathcal{U}}$. If we define $c^{\mathcal{U}} = \sum_{\sigma\in\Sigma} c^{\mathcal{U}}_\sigma$ and $t^{\mathcal{U}} = \sum_{\sigma\in\Sigma} t^{\mathcal{U}}_\sigma$ then the model is

$$P_{\mathcal{U}}(s) = \frac{c^{\mathcal{U}}_s - t^{\mathcal{U}}_s \delta^{\mathcal{U}}}{c^{\mathcal{U}}} + \frac{t^{\mathcal{U}}\delta^{\mathcal{U}} P_{\sigma(\mathcal{U})}(s)}{c^{\mathcal{U}}}$$

where $\sigma([s_1, s_2, s_3, \ldots]) = [s_2, s_3, \ldots]$ and $P_{\sigma([])}$ is the uniform distribution over Σ.

Given a model and observed data, the task of inference is to learn likely values of the latent parameters. The latent parameters of the sequence memoizer are the counts in the set of sets $\mathcal{G} = \{\{c_\sigma^\mathcal{U}, t_\sigma^\mathcal{U}\}_{\sigma\in\Sigma} \mid \mathcal{U} \in \Sigma^+\}$ and δ_n for $n \geq 0$. While $|\mathcal{G}| = \infty$ for sequences of unbounded length, [Wood et al., 2009] show that inference only requires computation on $\mathcal{H} \subset \mathcal{G}$ for any finite training sequence $\mathcal{S}$ such that $|\mathcal{H}| \leq 2|\mathcal{S}|$. Unfortunately the linear growth of $|\mathcal{H}|$ makes using the model intractable for streaming sequences. [Bartlett et al., 2010] demonstrate an approximate inference method that maintains a constant bound on $|\mathcal{H}|$ for arbitrary length sequences yet yields results comparable to inference in the full model. Finally, Gasthaus and Teh [2011] develop an exact representation of $\mathcal{H}$ that requires maintaining at most most $2|\Sigma||\mathcal{H}|$ unique counts. Combining these results yields an inference algorithm with asymptotically constant storage complexity for $\mathcal{H}$.

2.2 Approximation

We introduce two more approximating procedures that combined with those above yield streaming deplump. First, because the data structure used by batch deplump is a suffix tree its storage complexity is linear in the input sequence length. For this reason, streaming deplump cannot maintain a suffix tree representation of the entire input sequence. Instead a fixed-length "reference sequence" is maintained (see Algorithm 2), along with a dynamically updated suffix tree referencing only the suffixes found therein. Deleting nodes to constrain the size of this context tree forces node-removal inference approximations of the sort defined in [Bartlett et al., 2010], operations that are justifiable from a statistical perspective but whose net practical effect on compression performance is characterized for the first time in this paper.

As noted, the efficient representation introduced in Gasthaus and Teh [2011] maintains a single pair of counts $c_\sigma^\mathcal{U}$ and $t_\sigma^\mathcal{U}$ for each $\mathcal{U} \in \mathcal{H}$ and $\sigma \in \Sigma$. During incremental estimation, for each symbol s in the input sequence, $c_s^\mathcal{U}$ is incremented in at least one node on the tree. Clearly $\max_{P\in\mathcal{H},\sigma\in\Sigma}\{c_\sigma^\mathcal{U}\}$ then grows monotonically as a function of the length of the input sequence. Unfortunately incremental construction of the model includes an operation on node elements known as fragmentation (Alg. 5) that requires computation proportional to $\max_{\sigma\in\Sigma}\{c_\sigma^\mathcal{U}\}$ for node $\mathcal{U}$ [Gasthaus and Teh, 2011]. Therefore, to ensure computational complexity that is independent of the sequence length, $c_\sigma^\mathcal{U}$ must be bounded for all $\mathcal{U} \in \mathcal{H}$. The effect of imposing this restriction on compression performance is also characterized for the first time in this paper.

3 Algorithm

Our focus in this section is on providing the most complete implementation reference for deplump to date with the changes necessary to achieve streaming asymptotics highlighted. Many of the individual algorithms that appear in this section are justified and explained in detail in earlier papers. We explain the functionality of each algorithm, but refer the interested reader to the earlier papers for detailed mathematical justifications, in particular [Wood et al., 2009; Gasthaus et al., 2010; Bartlett et al., 2010].

Given a sequence of symbols $\mathcal{S} = [s_0, s_1, s_2, \ldots]$ where each symbol s_n comes from from an ordered set of symbols $\{\sigma_1, \sigma_2, \ldots\} = \Sigma$, streaming deplump works by repeatedly producing a predictive distribution for the continuation of the sequence given the full

Algorithm 1 Deplump/Plump

```
 1: procedure $\mathcal{O} \leftarrow$ DEPLUMP/PLUMP($\mathcal{I}$)
 2:     $\mathcal{R} \leftarrow [\,]$ ; $\mathcal{O} \leftarrow [\,]$
 3:     $nc \leftarrow 1$                                            ▷ node count
 4:     $\mathcal{D} \leftarrow \{\delta_0, \delta_1, \delta_2, \ldots, \delta_{10}, \alpha\}$        ▷ discount parameters
 5:     for i = 1: $|\mathcal{I}|$ do
 6:         $\mathcal{G} \leftarrow \vec{0}$                 ▷ discount parameter gradients, $|\mathcal{G}| = |\mathcal{D}|$
 7:         $\{\pi, \mathcal{N}\} \leftarrow$ PMFNEXT ($\mathcal{R}$)
 8:         if Plump then
 9:             $\mathcal{O} \leftarrow [\mathcal{O}$, DECODE($\pi, \mathcal{I}$)]
10:         else
11:             $\mathcal{O} \leftarrow [\mathcal{O}$, ENCODE($\Sigma_{j=1}^{\mathcal{I}[i]-1}\pi_j, \Sigma_{j=1}^{\mathcal{I}[i]}\pi_j$)]
12:         UPDATECOUNTSANDDISCOUNTGRADIENTS($\mathcal{N}, s, \pi_s$,TRUE)
13:         $\mathcal{D} \leftarrow \mathcal{D} + \mathcal{G}\eta/\pi_s$                 ▷ update discount parameters
14:         $\mathcal{R} \leftarrow [\mathcal{R}\ s]$                ▷ append symbol to reference sequence
```

preceding context and encoding the next symbol by passing that predictive distribution to an entropy encoder. We assume that if the predictive distribution function is F and the next symbol in the stream is s then an entropy encoder exists that takes $F(s-1)$ and $F(s)$ as arguments and returns a bit stream [Witten et al., 1987]. We use the notation $s-1$ to refer to the symbol prior to s in the symbol set Σ.

The algorithms of streaming deplump run over a constant space, incrementally constructed suffix-tree-like data structure. Note that the fact that this datastructure is of constant size is a significant point of differentiation between this work and that of Gasthaus et al. [2010]. This tree-shaped data structure efficiently encodes a subset of the unique suffixes of all contexts in a sequence. Each edge in the tree has a label which is a subsequence of the input sequence. Each edge label is represented by two indices into a fixed-length suffix of the input sequence which we call the reference sequence (i.e. each edge label is $[r_{i+1}, r_{i+2}, \ldots, r_j]$ for indices i and j into the reference sequence where $[r_1 = s_{n-T}, \ldots, r_T = s_n]$ after the nth input sequence element is processed.) Therefore, each node in the tree corresponds to a subsequence $[s_m, \ldots, s_{m+k}]$ of the input sequence and a (potentially non-sequential) set of subsequences of the reference sequence. We use $\mathcal{N}$ to refer interchangeably to a node and the suffix in the input sequence to which the node corresponds. For a given suffix the corresponding node is accessed by traversing edges of the tree that match the suffix read from right to left.

Streaming deplump is given in Algorithm 1. Deplump processes each element of an input sequence $\mathcal{I}$ incrementally. For each element s_n of $\mathcal{I}$, PMFNEXT computes the predictive probability mass function π (conditioned on the observed context $\mathcal{R}$) needed for compression. The element s_n is then encoded by an entropy encoder with parameter π. PMFNEXT also handles the incremental maintenance of the underlying constant-space datastructures by both restricting the length of the reference sequence and constructing and deleting nodes in the tree as needed. The final steps are to incrementally integrate the observation into the approximated sequence memoizer and to adjust the back-off/smoothing parameters $\mathcal{D}$.

Algorithm 2 PMFNext

```
1: function $\{\pi, \mathcal{N}\} \leftarrow$ PMFNEXT($\mathcal{R}$)
2:     while $|\mathcal{R}| \geq T$ do
3:         $\mathcal{R} \leftarrow \sigma(\mathcal{R})$
4:     while $nc > (L-2)$ do
5:         Delete random leaf node
6:         $nc \leftarrow nc - 1$
7:     $\mathcal{N} \leftarrow$ GETNODE($\mathcal{R}, \mathcal{T}$)
8:     $\pi \leftarrow$ PMF($\mathcal{N}, \vec{0}, 1.0$)    $\triangleright |\vec{0}| = |\Sigma|$
```

Algorithm 3 DrawCRP

```
1: function $t \leftarrow$ DRAWCRP($n, d, c$)
2:     $t \leftarrow 1$
3:     for i = 2 : n do
4:         $r \leftarrow 0$
5:         $r \leftarrow 1$ w.p. $\frac{td+c}{i-1+c}$
6:         $t \leftarrow t + r$
```

Algorithm 4 PMF

```
1: function $\pi \leftarrow$ PMF($\mathcal{N}, \pi, m$)
2:     if $c^{\mathcal{N}} > 0$ then
3:         for $\sigma \in \Sigma$ do
4:             $\pi_\sigma \leftarrow \pi_\sigma + m(\frac{c^{\mathcal{N}}_\sigma - t^{\mathcal{N}}_\sigma d^{\mathcal{N}}}{c^{\mathcal{N}}})$
5:     if PA($\mathcal{N}$) $\neq$ null then
6:         $\pi \leftarrow$ PMF(PA($\mathcal{N}$), $\pi, d^{\mathcal{N}} m$)
7:     else
8:         $\pi \leftarrow (1 - d^{\mathcal{N}} m)\pi + d^{\mathcal{N}} m P_{\sigma([])}$
```

Algorithm 5 GetNode

```
1: function $\mathcal{S} \leftarrow$ GETNODE($\mathcal{S}, \mathcal{T}$)
2:     Find the node $\mathcal{M}$ in the tree sharing the longest suffix with $\mathcal{S}$ and update indices on edges in the path from $\mathcal{M}$ to the root to point to more recent sections of $\mathcal{R}$
3:     if $\mathcal{M}$ is a suffix of $\mathcal{S}$ then
4:         if $\mathcal{S} \neq \mathcal{M}$ and $|\mathcal{M}| < D$ then
5:             $\mathcal{S} \leftarrow$ CREATENODE($\mathcal{S}, \mathcal{M}$)
6:             $nc \leftarrow nc + 1$
7:     else
8:         $\mathcal{P} \leftarrow$ FRAGMENTNODE($\mathcal{M}, \mathcal{S}$)
9:         $\mathcal{S} \leftarrow$ CREATENODE($\mathcal{S}, \mathcal{P}$)
10:        $nc \leftarrow nc + 1$
```

Algorithm 6 ThinCounts

```
1: function THINCOUNTS($\mathcal{N}$)
2:     while $c^{\mathcal{N}} > k$ do
3:         $\pi = [\frac{c^{\mathcal{N}}_{\sigma_1}}{c^{\mathcal{N}}}, \frac{c^{\mathcal{N}}_{\sigma_2}}{c^{\mathcal{N}}}, \ldots, \frac{c^{\mathcal{N}}_{\sigma_{|\Sigma|}}}{c^{\mathcal{N}}}]$
4:         $s \leftarrow$ MULTINOMIAL($\pi$)
5:         $\phi \leftarrow$ PARTITION($c^{\mathcal{N}}_s, t^{\mathcal{N}}_s, d^{\mathcal{N}}$)
6:         $\psi \leftarrow (\frac{1}{c^{\mathcal{N}}_s})\phi$
7:         $i \leftarrow$ MULTINOMIAL($\psi$)
8:         if $\phi_i = 1$ then
9:             $t^{\mathcal{N}}_s \leftarrow t^{\mathcal{N}}_s - 1$
10:        $c^{\mathcal{N}}_s \leftarrow c^{\mathcal{N}}_s - 1$
```

This involves updating counts in the tree and calculating a stochastic gradient for $\mathcal{D}$. Both of these are done in UPDATECOUNTSANDDISCOUNTGRADIENTS. The organization of functions in this way minimizes the number of tree traversals that need to be performed, tree traversals being a major component of the computational cost of this algorithm.

PMFNEXT (Alg. 2) starts by enforcing the bounds on $|\mathcal{R}|$ and $|\mathcal{H}|$. Both of these bounds require nodes to be removed from the tree (the first implicitly as a result of edge labels becoming undefined and the second explicitly). Removing nodes from the underlying datastructure is simple to do, but as an operation on the corresponding sequence memoizer graphical model some care must be taken because node removal has ramifications in terms of what kinds of approximations are being imposed on inference. Bartlett et al. [2010] showed that removing leaf nodes from the sequence memoizer results in a coherent approximate inference algorithm for the sequence memoizer. So, enforcing the bound on $|\mathcal{H}| < L$ is as simple as incrementally removing leaf nodes uniformly at random. To facilitate random deletion we maintain a count in each node of the number of leaf nodes in the subtree below it. A random leaf node can then be obtained by traversing a weighted random path down the

Algorithm 7 UpdateCountsAndDiscountGradients

```
1: function UPDATECOUNTSANDDISCOUNTGRADIENTS($\mathcal{N}$, $s, p$, BackOff)
2:     $pp \leftarrow p$
3:     if $c^{\mathcal{N}} > 0$ then
4:         $pp \leftarrow (p - \frac{c_s^{\mathcal{N}} - t_s^{\mathcal{N}} d^{\mathcal{N}}}{c^{\mathcal{N}}})(\frac{c_{\mathcal{N}}}{t^{\mathcal{N}} d^{\mathcal{N}}})$
5:         $w \leftarrow c_s^{\mathcal{N}} + d^{\mathcal{N}}(t^{\mathcal{N}} pp - t_s^{\mathcal{N}})$
6:     if BackOff and $c > 0$ then
7:         $c_s^{\mathcal{N}} \leftarrow c_s^{\mathcal{N}} + 1$
8:         BackOff $\leftarrow 0$
9:         BackOff $\leftarrow 1$ w.p. $pp(\frac{t^{\mathcal{N}} d^{\mathcal{N}}}{w})$          ▷ w.p abbreviates "with probability"
10:        if BackOff then
11:            $t_s^{\mathcal{N}} \leftarrow t_s^{\mathcal{N}} + 1$
12:    else if BackOff then
13:        $c_s^{\mathcal{N}} \leftarrow c_s^{\mathcal{N}} + 1$; $t_s^{\mathcal{N}} \leftarrow t_s^{\mathcal{N}} + 1$
14:    UPDATEDISCOUNTPARAMETERGRADIENTS
15:    UPDATECOUNTSANDDISCOUNTGRADIENTS(PA($\mathcal{N}$), $s, pp$, BackOff)
16:    THINCOUNTS($\mathcal{N}$)
```

tree.

The removal of nodes due to their edge labels becoming undefined is an unfortunate consequence of having to constrain the reference sequence $\mathcal{R}$ to be of constant length. Note that without a bound $|\mathcal{R}|$ would grow in length by one as each symbol of the input sequence is incorporated into the model. To maintain the upper bound on $|\mathcal{R}|$ we shorten $\mathcal{R}$ by one as in a fixed length FIFO queue. Since the labels of edges in the tree index into $\mathcal{R}$ then, when it is shortened, some edges may end up having labels that index beyond the retained elements of $\mathcal{R}$ resulting in "undefined" edges. Nodes below any undefined edges in the tree must be removed because they can no longer be accessed. Their removal is justified in the same way as before because all subtree removals can be implemented as a cascade of leaf node removals. It is desirable to minimize the number of nodes removed due to the incessant shortening of $\mathcal{R}$. One way to do this is to update all edge indices on all paths traversed in the process of inserting nodes into the tree (Alg. 5). PMFNEXT finishes by accessing the correct node for prediction using GETNODE (Alg. 5) and the predictive probability mass function is calculated recursively by PMF (Alg. 4).

Incremental construction of the tree is handled by GETNODE within PMFNEXT which creates two or fewer nodes in the tree for every call. Within the function GETNODE, nodes are created by CREATENODE and FRAGMENTNODE. The function CREATENODE($\mathcal{N}$,$\mathcal{M}$) simply creates a node $\mathcal{N}$ with parent $\mathcal{M}$. FRAGMENTNODE (Alg. 8) implements the fragmentation operation developed in [Wood et al., 2009] required to ensure proper sequence memoizer inference in the case where an edge must be split and a node inserted, but does so in the constant space node representation given in [Gasthaus and Teh, 2011]. This new intervening node corresponds to a predictive context and the counts corresponding to a proper estimate of this predictive distribution must be inferred. Practically this means that value must be assigned to all counts $t_s^{\mathcal{P}}, c_s^{\mathcal{P}}, t_s^{\mathcal{M}}, c_s^{\mathcal{M}}$ for $s \in \Sigma$. FRAGMENTNODE uses two

Algorithm 8 FragmentNode

```
1: function $\mathcal{P} \leftarrow$ FRAGMENTNODE($\mathcal{M}, \mathcal{S}$)
2:     $\mathcal{P} \leftarrow$ maximum overlapping suffix of $\mathcal{M}$ and $\mathcal{S}$
3:     $\mathcal{P} \leftarrow$ CREATENODE($\mathcal{P}$, PA($\mathcal{M}$))
4:     $nc \leftarrow nc + 1$
5:     PA($\mathcal{M}$) $\leftarrow \mathcal{P}$
6:     for $\sigma \in \Sigma$ do
7:         $\phi \leftarrow$ PARTITION $(c_\sigma^{\mathcal{M}}, t_\sigma^{\mathcal{M}}, d^{\mathcal{M}})$
8:         $t_\sigma^{\mathcal{P}} \leftarrow t_\sigma^{\mathcal{M}}; t_\sigma^{\mathcal{M}} \leftarrow 0$
9:         for $i = 1 : |\phi|$ do
10:            $a \leftarrow$ DRAWCRP$(\phi[i], d^{\mathcal{M}}/d^{\mathcal{P}}, -d^{\mathcal{M}})$
11:            $t_\sigma^{\mathcal{M}} \leftarrow t_\sigma^{\mathcal{M}} + a$
12:        $c_\sigma^{\mathcal{P}} \leftarrow t_\sigma^{\mathcal{M}}$
```

Algorithm 9 Partition

```
1: function $\phi \leftarrow$ PARTITION$(c, t, d)$
2:     $M \leftarrow t \times c$ matrix of zeros
3:     $M(t, c) \leftarrow 1.0$
4:     for $j = (c-1) : 1$ do
5:         for $i = 1 : (t-1)$ do
6:             $M(i, j) \leftarrow M(i+1, j+1) + M(i, j+1)(j - id)$
7:         $M(d, j) \leftarrow M(t, j+1)$
8:     $\phi \leftarrow [1, 0, 0, \ldots, 0]$                                ▷ $|\phi| = t$
9:     for j = 2 : c do
10:        $M(k, j) \leftarrow M(k, j)(j - 1 - kd)$
11:        $r \leftarrow 0$
12:        $r \leftarrow 1$ w.p. $\frac{M(k+1,j)}{M(k+1,j)+M(k,j)}$
13:        if r = 1 then
14:            $k \leftarrow k + 1$
15:            $\phi[k] \leftarrow 1$
16:        else
17:            $i \leftarrow$ MULTINOMIAL$([\frac{\phi[1]-d}{j-1-kd}, \frac{\phi[2]-d}{j-1-kd}, \ldots, \frac{\phi[k]-d}{j-1-kd}])$
18:            $\phi[i] \leftarrow \phi[i] + 1$
```

functions to do this: PARTITION(c, t, d) (Alg. 9) and DRAWCRP(n, d, c) (Alg. 3). The net effect of FRAGMENTNODE is to create a new branching node at the shared suffix of $\mathcal{M}$ and $\mathcal{S}$. The mathematical explanation of these algorithms is given in [Gasthaus and Teh, 2011].

Finally UPDATECOUNTSANDDISCOUNTS (Alg. 7) integrates the observation into the underlying probability model, adjusting counts in potentially all nodes on the path from the context in which the observation was made to the root of the context tree. UPDATE-COUNTSANDDISCOUNTS also computes a partial stochastic gradient for the discount parameter of each node on this path (this gradient is used in DEPLUMP to update $\mathcal{D}$). The details of the discount computation performed in UPDATEDISCOUNTPARAMETERGRADI-

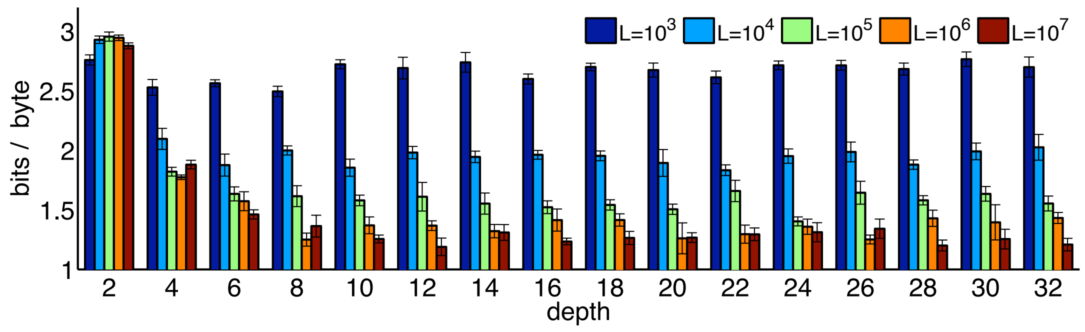

Figure 1: Average ($\pm$ std.) streaming deplump compression performance as measured in bits in compressed output vs. bytes in uncompressed input. Here the depth limit (D) and node limit (L) are varied. From this we conclude that setting the depth limit to $D \geq 16$ and the node limit to the largest value possible given physical memory constraints leads to optimal compression performance.

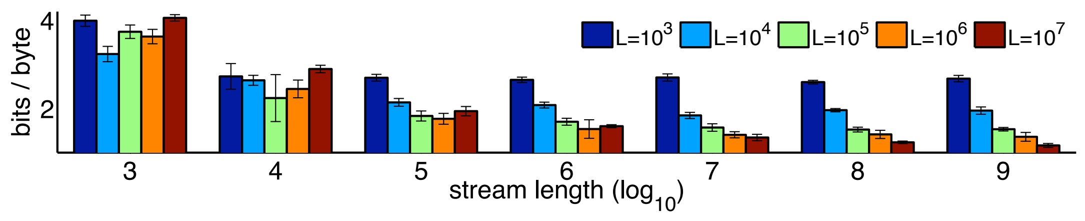

Figure 2: Average ($\pm$ std.) streaming deplump compression performance as measured in bits in compressed output vs. bytes in uncompressed input. Here the input stream length and node limit (L) are varied. From this we observe that average deplump streaming compression performance monotonically improves as the input sequence grows in length but asymptotes at a different value per node limit Also, it can be seen that large node limits may actually hurt compression performance for small input sequences.

ENTS are too long to include. UPDATECOUNTSANDDISCOUNTS also uses THINCOUNTS (Alg. 6) to enforce the bound on the counts in each node required to ensure computational asymptotics appropriate for streaming compression.

To use streaming deplump a choice of approximation parameters must be made. The full set of these parameters consists of $\mathcal{D}$, D, T, k, L, η. $\mathcal{D} = [\delta_0, \delta_1, \ldots, \delta_{10}, \alpha]$ is a list of discount parameters, each taking a real value in $(0, 1)$. If we define $\delta_n = \delta_{10}^{\alpha^{n-10}}$ for $n \geq 10$, and δ_n for $n \leq 10$ then $d^{\mathcal{N}} = \Pi_{i=|\text{PA}(\mathcal{N})|+1}^{|\mathcal{N}|} \delta_i$. D is the maximum depth of the suffix tree which corresponds to both the maximum context length used in modeling and the maximum recursion depth of all of the algorithms. T is the bound on the length of the reference sequence $\mathcal{R}$ and is typically set to a multiple of the upper bound on the number of node instances in the suffix tree L. The parameter k is the upper bound on the total count $c^{\mathcal{N}}$ in each node. The parameter η is a learning rate for the updating of the discount parameters and is typically set to a very small value.

4 Experiments

As the performance of batch deplump was established relative to other lossless compressors for a variety of datatypes in [Gasthaus et al., 2010], we focus our experiments on establishing a) that the approximations to inference in the sequence memoizer combined in this paper in order to make deplump a streaming compressor do not significantly adversely affect compression performance, b) that the resulting compressor can indeed compress extremely long sequences as the asymptotic guarantees would suggest it should, and c) which settings of the approximation parameters produce the best compression performance. Most of the experiments included in this paper use a complete Wikipedia text content dump [Wikipedia, 2010] as a test corpus (26.8Gb uncompressed, 7.8Gb gziped both with default parameters, and 3.9Gb deplumped).

To establish what approximation parameters produce the best compression results we first ran streaming deplump on the first 100Mb section of the corpus limiting the depth to two different values ($D = 16$ and $D = 1024$) with a fixed limit on the number of nodes in the tree ($L = 10^6$). We observed that in both cases only very few nodes had high total counts ($c > 8,192$) suggesting that it might be possible to set the count upper bound conservatively ($k = 8,192$) without significant compression performance degradation. To ensure that compression performance did in fact not suffer, we compressed ten 100Mb subsections of the corpus (sampled randomly with replacement) for multiple values of k between 128 and 32,768 (fixing $L = 10^6$ and $D = 16$). We observed that average compression performance indeed varied insignificantly over all values of k.

The interplay between limiting the number of nodes in the tree and restricting the depth of the tree was explored (results for which are shown in Figure 1). Here k was set to 8,192, while L and the depth of the tree D were varied as indicated in the figure. The interplay between stream length and node limit was also explored (Figure 2). In this experiment k remained fixed at the same value and the depth was set to $D = 16$ while L varied as shown in the figure. In both experiments, subsections of the corpus were sampled with replacement. In the first experiment the sampled subsections were all of size 100Mb, while in the second the sampled subsections varied in length.

Figure 1 indicates that compression performance becomes essentially indistinguishable for depth restrictions greater than $D = 10$. However, this figure also suggests that compression performance improves as a function of the number of nodes in the tree for depths 6 and greater. Figure 2 illustrates that the algorithm not only scales to very long sequences, but average compression performance continues to improve as the sequence grows. Using a large value of L appears to be beneficial for very long sequences.

Considering these results, we chose values for the approximating parameters ($D = 32$, $L = 10^7$, $k = 8,192$, $T = 10^8$, $\mathcal{D} = [.5, .7, .8, .82, .84, .88, .91, .92, .93, .94, .95, .93]$, and $\eta = 0.0001$) then compared streaming deplump to to batch deplump. In this experiment we achieved compression performance of 1.67 bites per byte on the 100Mb Wikipedia corpus excerpt used for the Hutter Prize [Hutter, 2006] using the streaming variant of deplump. Batch deplump [Gasthaus et al., 2010] achieved 1.66 bits per byte. If we were to have instead taken the most obvious approach to streaming compression, namely, compressing the sequence in blocks until a memory bound is hit and then starting over, performance falls to 1.88 bits per byte (limiting the sequence length of each block such that the resulting batch deplump compressor used a maximum amount of memory roughly equivalent to the amount of memory used by the streaming variant). These results further demonstrate that the streaming variant of deplump is a significant improvement over the naïve approach to compression of streaming data using batch deplump.

5 Discussion

In this paper we developed a new streaming variant of deplump. We showed that the approximations introduced to make deplump stream-capable had almost no effect on compression performance relative to batch deplump, meaning that streaming deplump likewise outperforms unbounded context CTW, PPMZ, among others yet can be used to compress streaming data of unbounded length. It remains the case that streaming deplump could be integrated into the PAQ family of compressors.

References

Bartlett, N., Pfau, D., and Wood, F. (2010). Forgetting counts: Constant memory inference for a dependent hierarchical Pitman-Yor process. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 63–70.

Cleary, J. G. and Teahan, W. J. (1997). Unbounded length contexts for PPM. *The Computer Journal*, 40:67–75.

Gasthaus, J. and Teh, Y. W. (2011). Improvements to the sequence memoizer. In *Proceedings of Neural Information Processing Systems*.

Gasthaus, J., Wood, F., and Teh, Y. W. (2010). Lossless compression based on the Sequence Memoizer. In *Data Compression Conference 2010*, pages 337–345.

Hutter, M. (2006). Prize for compression human knowledge. URL: `http://prize.hutter1.net/`.

Mahoney, M. V. (2005). Adaptive weighing of context models for lossless data compression. Technical report, Florida Tech. Technical Report CS-2005-16, 2005.

Peltola, H. and Tarhio, J. (2002). URL: `http://cs.hut.fi/u/tarhio/ppmz`.

Wikipedia (2010). URL: `http://download.wikimedia.org/enwiki/`.

Willems, F. M. J. (1998). The context-tree weighting method: Extensions. *IEEE Transactions on Information Theory*, 44(2):792–798.

Willems, F. M. J. (2009). CTW website. URL: `http://www.ele.tue.nl/ctw/`.

Witten, I. H., Neal, R. M., and Cleary, J. G. (1987). Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540.

Wood, F., Archambeau, C., Gasthaus, J., James, L., and Teh, Y. W. (2009). A stochastic memoizer for sequence data. In *Proceedings of the 26th International Conference on Machine Learning*, pages 1129–1136, Montreal, Canada.