Blended Dictionaries for Reduced-Memory Lempel-Ziv Corpus Compression

Jiancong Tong[†‡] jctong@nbjl.nankai.edu.cn
Anthony Wirth[‡] awirth@unimelb.edu.au
Justin Zobel[‡] jzobel@unimelb.edu.au

[†]College of Computer and Control Engineering, Nankai University, China
[‡]Department of Computing and Information Systems, The University of Melbourne, Australia

ABSTRACT

Relative Lempel-Ziv (RLZ) compression has been shown to be effective for compression of large text repositories. It provides high compression ratios with extremely fast atomic decompression of individual documents. However, it depends on a large in-memory dictionary, which is implemented as a contiguous string that must be accessed randomly during the decompression process. In this paper we explore how compressed suffix arrays might reduce the size of the dictionary. These suffix arrays drastically increase the cost of accessing individual characters, however, so we propose splitting of the dictionary: an uncompressed structure for frequently accessed dictionary elements, with compression for the remainder. Our results show that splitting provides a smoothly tuneable trade-off between access time and memory requirements, but does not overcome the inherent limitations of compressed suffix arrays for this application, with decompression time growing by a factor of 10 for even the best combination of parameters. Suffix arrays comprise an attractive option where memory is limited, high compression is paramount, and decompression speed is unimportant.

Categories and Subject Descriptors

E.4 [**Coding and Information Theory**]: [Data compaction and compression]; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Search process*

Keywords

Repository compression; dictionary compression; encoding; document retrieval

1. INTRODUCTION

Compression of repositories is an effective mechanism for reducing the costs of hosting a collection of documents that is to be made available for search. An effective compression mechanism not only reduces the space required to store the documents but can increase the efficiency of retrieval, via reduced disk-to-machine and machine-to-cache bandwidth, and by increasing the number of documents that can be cached.

ADCS '14, November 27–28 2014, Melbourne, VIC, Australia

ACM 978-1-4503-3000-8/14/11 ...$15.00
http://dx.doi.org/10.1145/2682862.2682866

Relative Lempel-Ziv (RLZ) compression has been demonstrated to have these characteristics [3, 4]. It allows independent retrieval of individual documents, provides high compression ratios, and extremely fast decompression. In RLZ, the compression dictionary is held as a single long string; the compressed data is represented as a sequence of substrings drawn from the dictionary, each of specified length at a specified position. The published experiments on RLZ, by ourselves [8] and by Hoobin et al. [4], show that this dictionary needs to be around 1 GB to provide good compression ratios, and that this threshold is reasonably independent of the volume of data to be compressed. Indeed, the threshold seems to depend more on the type of data compressed: with a single 1-GB dictionary, RLZ effectively compresses terabyte-scale collections of English text.

Smaller dictionaries, say 100 MB, lead to poor compression. Fellow researchers have suggested that the RLZ dictionaries could be effectively represented as a compressed suffix array (CSA) [6]. This data structure stores the string and its index structure compactly, and allows each position in the string to be accessed with a fixed number of operations. In principle, this proposal is appealing, because of the scale of the space reductions that are often available with the compression mechanisms underlying CSAs. These mechanisms include, amongst others, a wavelet tree over Burrows-Wheeler transformed text.

A potential drawback is that use of a CSA could drastically increase execution times. Retrieval of each character involves a random access into a data structure whose size is likely to greatly exceed the capacity of the CPU cache. In contrast, in the original, array-based approach, a sequence of characters of arbitrary length can be fetched with a single memory request, an operation that is extremely efficient on current hardware. In the experiments here, this issue is indeed observed: decompression times rise by a factor of around 100.

However, Hoobin et al. [3], amongst others, observed that some parts of the dictionary are referred to more than others. Viewed as a sequence of fixed-length blocks, there is significant variation in the numbers of references to each block. Plausibly, if the frequently referenced blocks were maintained uncompressed, while the others were represented

as a compressed suffix array, we could reduce overall dictionary size while maintaining reasonable decompression speed. We investigate this strategy here.

In particular, we describe a new technique for rearranging the dictionary to allow splitting of *popular* (frequently referenced) blocks from other blocks, and also explain the performance resulting from a mixture of a compressed with an uncompressed dictionary. Our results on two 10-GB text collections show that increasing the proportion of the dictionary that is compressed increases the decompression time. For example if only 20% of the references are to the compressed part of the dictionary then decompression time increases by a factor of 12 on one test collection (and 20 on the other).

On current architectures, therefore, RLZ with a dictionary partly present as a CSA gives poor outcomes. Its performance is less attractive in both time and space overhead than a simple alternative such as `gzip`. While the method was, we believe, worth exploring, and there may be combinations of parameters and architectures where it can be used, in this setting it was essentially a failure.

2. APPROACH

Relative Lempel-Ziv coding has been described in detail in several recent papers [3, 4, 8]. In brief, a collection to be compressed can be regarded as a single string. A *dictionary* is created by sampling substrings from the collection and concatenating them, within a certain predetermined size ; in these and previous experiments, the sampling extracts substrings of a fixed length (say 1 KB) at fixed intervals. To obtain a dictionary of 1000 MB, a total of 1,024,000 substrings would need to be gathered. (In a 1-TB collection, these substrings would be approximately 1 MB apart.) These are the parameters used in the experiments reported here.

To compress a document, the compressor would find the longest substring in the dictionary matching the document's prefix, that is, the longest substring corresponding to the document's initial sequence of characters. This substring would occur at position p in the dictionary and be of length l. The compressor can then emit the pair $\langle p, l \rangle$ as a *reference*, and proceed to find a match to the next sequence of characters. A compressed document is then represented as a sequence of references, which can readily be stored as, say, Elias or Golomb codes [5].

If the dictionary is represented as an array, decompression then consists of processing each reference in turn; for each reference, the decoder need only jump to position p in the array and output the next l characters. It is the simplicity of this decoding that gives RLZ its high performance, with decompression speed much greater than that of other mechanisms [4].

Other experiments have consistently found that the main determinant of compression effectiveness for RLZ is dictionary size. It is thus attractive to build a large dictionary, of say 1 GB or more; but this size also makes it attractive to investigate ways of reducing dictionary size.

One approach is to prune the dictionary, as explored in our previous work [8], which showed that dictionary size can be reduced by a factor of four or so with only limited impact on compression effectiveness; this pruning is based, in effect, on removal of redundant substrings.

Another possible approach is to directly compress the dictionary by representing it as a compressed suffix array, as we explore here. Unfortunately, we cannot simply apply `gzip` or `7zip` to compress the dictionary, because we could like to extract substrings from the dictionary, as needed, efficiently. We therefore incorporate a compressed suffix array (CSA), as implemented in the SDSL library [2]. There are several standard options for the CSA, including a wavelet tree based FM-Index [1], and the approach of Sadakane [7], in which the ψ-array, of adjacent sorted suffixes, is compressed with an Elias code.

However, as we report below, the impact on decompression time is dramatic. In a variety of experiments we found that time increased by a factor of 100 or more, in exchange for compression of a 1000 MB dictionary to 300 MB or so. Though there may be circumstances in which this is an effective trade-off, we hypothesised that better outcomes might be available.

3. SPLITTING THE DICTIONARY

The approach explored here is to split the dictionary into two components, one compressed with a CSA and the other uncompressed (as in the original structure). As some blocks are more popular than others, it is plausible that, by keeping just the popular blocks uncompressed, we can obtain some of the compression gains without incurring the speed penalties.

The task then is to effectively split the dictionary, that is, to gather together the popular blocks and leave behind the remainder. What appeared to be a straightforward problem, however, turned out to have several challenges. A simple one was that, in most cases where the matching algorithm was attempting to locate the longest string, there were several candidates, and it was often the case that these ran across two blocks. If the blocks were rearranged, what were popular blocks could become less frequently referenced; this problem was so acute that in our initial experiments, after rearrangement what had been the most popular blocks were no more frequently accessed than the remainder.

Another challenge was the assessment of popularity. Intuition suggested that the right way to count was by the number of references that lay within a block, but reflection (and experimentation) showed that the important, dominant cost was character decoding, and so popularity needed to be measured by numbers of characters accessed; this shifted the measure away from short, common references to long, less common references.

An elementary challenge was that, in the implementation we used, an additional step was required to ensure that the match chosen in the dictionary had the smallest address from amongst the options available.

With popularity assessed by this *byte-wise reference counting*, we could then sort the blocks from most to least pop-

ular, and partition the dictionary into two parts at an arbitrary point. The left-hand part could be stored as an array, and the right-hand part stored as a CSA.

A further step was then required, however, to give significant bias to the references. Suppose we have found a match of length n to a substring, but the match is in the right-hand part of the dictionary. It is then attractive to see if there is a match of length $n-1$ in the left-hand part, slightly reducing compression effectiveness but increasing decoding efficiency. This *selective early termination* of a match significantly increases the number of references to the left-hand part of the dictionary.

The effect of these steps is illustrated in Figure 1, for the first 10 GB of each of our two test collections, GOV2 and a Wikipedia crawl (as described in our earlier paper [8]). The figure shows the cumulative percentage of references to the dictionary, from left to right. The black line shows the original distribution; unsurprisingly, the distribution is extremely flat. More surprisingly, this is very close to the distribution achieved by simply counting the frequency of references and reordering; that is, simple reordering methods are ineffective. The red curve shows the distribution achieved with byte-wise reference counting and selective early termination.

The redistribution is significant. Around 50% of the references are to the first 20% of the dictionary, and 80% of the references are in the first 50% of the dictionary. This distribution gives us a promising basis for testing RLZ with a dictionary that is split into an uncompressed component and a compressed component.

4. OUTCOMES

Using a split dictionary, divided at a range of different places, we evaluated decompression time. These experiments took a stream of references representing the whole collection, and sequentially accessed the referred-to substrings, thus decompressing all the documents in turn. As described above, for each dataset, the original dictionary size is 1000 MB, and the two datasets used are the first 10 GB from each of the collections.

For efficiency of experimentation, we fixed the boundary for selective early termination of matches at 20% of the collection; varying this boundary to match the size of the left-hand part would slightly improve speed of decompression, but (in an initial experiment) not significantly, while greatly increasing the time required to run the series of experiments.

The exact incantation of the data structure was

```
csa_type = csa_sada <enc_vector
<coder::elias_delta, 256>,100000000,64>;
```

which means that the kind of CSA used in this experiment was a Sadakane [7] style, incorporating Elias delta codes (in blocks of 256 integers at a time) to compress the ψ function. The last two integers 100,000,000 and 64 refer to the sample rate of the CSA.

In a suffix array, $SA[i]$ is the original location of the ith suffix in sorted order, while $SA^{-1}[j]$ is the sorted position of the jth suffix in original (text) order. To allow fast access to both $SA[i]$ and $SA^{-1}[j]$, samples of the uncompressed $SA[]$ and $SA^{-1}[]$ arrays have to be stored periodically in the CSA. There is a tradeoff between space and access speed, and in fact only every 64th entry of the inverse array is stored uncompressed. During initial experiments, several implementations of CSA were tried, with a range of parameter settings; while there were implementations that gave slightly better compression, decoding times were much greater.

Experimental results, in Table 1, show that a fully compressed dictionary (0,100) is about 100 times slower than a fully uncompressed dictionary (100,0), on each collection. Perhaps the most promising point in each case is (50,50), where the dictionary size is reduced by about a third, over 80% of the references are to the uncompressed left-hand part of the dictionary, and compression time is increased by a factor of 12 (Wikipedia) or 20 (GOV2). Total times are consistently greater, and compressed dictionary sizes consistently larger, than we anticipated, giving outcomes that do not support incorporating a CSA in RLZ repository compression.

Our expectation is that the same results (in terms of trend) would be observed on collections of any size, as compression performance and behaviour is not dependent on collection size [8]. A larger initial dictionary would allow greater bias, and also slightly better compression, but would also to some extent negate the aim of reducing memory consumption.

Overall, the results are markedly less impressive than dictionary pruning [8]. They are also less impressive than we would expect from a simple approach of gathering documents into blocks and compressing them with `gzip`, a conclusion we draw by inference from results reported in earlier work [4].

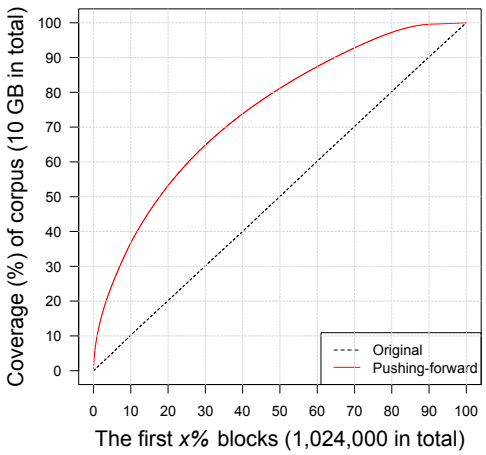
Moreover, the compression effectiveness of a CSA on the dictionary is surprisingly poor. Other methods for reducing dictionary size are based on removal of redundant material [8], a process that would leave a remainder that is likely to be even less compressible. We therefore conclude that this investigation demonstrates that use of a CSA for the dictionary in RLZ, while worthy of investigation, is not effective in practice.

Acknowledgements

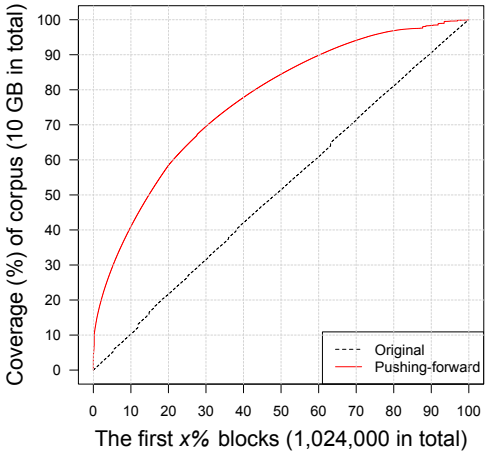
Simon Gog, for his SDSL library. Matthias Petri, for his instructions and advice on how to use and tune the data structures in SDSL. We are also greatly indebted to Matthias for his explanations of the subtleties of and ideas behind compressed suffix arrays. Alistair Moffat, for his feedback on earlier presentations and his questions about RLZ in general. This work was supported by the Australian Research Council (DP1093665) and by the China Scholarship Council.

References

[1] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. An alphabet-friendly FM-index. In *11th International Conference on String Processing and Information Retrieval (SPIRE 2004)*, pages 150–160. Springer, 2004. doi:10.1007/978-3-540-30213-1_23.



(a) Distribution of factors on GOV2 (first 10 GB).



(b) Distribution of factors on Wikipedia (first 10 GB).

Figure 1: Distribution of factors after rearranging the dictionary using byte-wise reference counting and selective early termination. In each case, the size of the dictionary is 1000 MB.

Table 1: Recovering the text collection using a split dictionary. Quantity "%Access" is the proportion of reference accesses, measured as the number of bytes extracted, to the uncompressed part of the dictionary.

Dataset	Uncompressed, Compressed	Uncompressed dict size (MB)	Compressed dict size (MB)	Total (MB)	Decoding time (s)	%Access (bytes)
GOV	100, 0	1000	0	1000	89	100.0
	90, 10	900	24	924	112	99.6
	80, 20	800	53	853	255	97.2
	50, 50	500	166	666	1623	81.2
	20, 80	200	285	485	4547	53.3
	10, 90	100	318	418	7183	35.4
	0, 100	0	346	346	9411	0.0
Wiki	100, 0	1000	0	1000	65	100.0
	90, 10	900	23	923	92	98.4
	80, 20	800	48	848	139	96.9
	50, 50	500	137	637	816	84.4
	20, 80	200	236	436	2876	58.5
	10, 90	100	266	366	4318	40.9
	0, 100	0	292	292	5849	0.0

[2] S. Gog, T. Beller, A. Moffat, and M. Petri. From theory to practice: Plug and play with succinct data structures. In *13th International Symposium on Experimental Algorithms, (SEA'14)*, pages 326–337, 2014. doi:10.1007/978-3-319-07959-2_28.

[3] C. Hoobin, S. J. Puglisi, and J. Zobel. Sample selection for dictionary-based corpus compression. In *Proceedings of the 34th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'11)*, pages 1137–1138, 2011. doi:10.1145/2009916.2010087.

[4] C. Hoobin, S. J. Puglisi, and J. Zobel. Relative Lempel-Ziv factorization for efficient storage and retrieval of web collections. *Proceedings of the VLDB Endowment*, 5(3): 265–273, 2011. doi:10.14778/2078331.2078341.

[5] A. Moffat and A. Turpin. *Compression and Coding Algorithms*. Kluwer, 2002. ISBN 0-7923-7668-4.

[6] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys (CSUR)*, 39(1):2, 2007. doi:10.1145/1216370.1216372.

[7] K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, 48(2): 294–313, 2003. doi:10.1016/S0196-6774(03)00087-7.

[8] J. Tong, A. Wirth, and J. Zobel. Principled dictionary pruning for low-memory corpus compression. In *Proceedings of the 37th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'14)*, pages 283–292, 2014. doi:10.1145/2600428.2609576.