Dynamic LZW for Compressing Large Files

Chung-E Wang
Department of Computer Science
California State University, Sacramento
Sacramento, CA 95819-6021

Abstract. *The amount of data stored digitally continues to grow dramatically across many fields, along with the need for algorithms to efficiently compress this data for storage and transmission. In this paper, we describe an improvement of LZW data compression. We employ a dynamic dictionary, in which least recently used and aging algorithms are used to replace infrequently used entries. We demonstrate that these pruning techniques result in significant gains in compression ratios for large data files.*

Keywords. LZW data compression, dynamic dictionary, table pruning, least recently used, aging replacement.

1. Introduction

Data compression algorithms are widely used for data storage and data transmission. A popular lossless method known as Lempel-Ziv (LZ) compression [1] replaces a string of characters with an index into a dictionary that is built during the compression process. There are many modifications of the original LZ compression algorithm, many of which are feature different implementations of the dictionary [1]-[6].

Lempel-Ziv-Welch (LZW) compression [4] is Terry Welch's modification of LZ compression. This algorithm uses a string table to implement the dictionary. Initially, the string table contains all strings of length 1. During the process of compression, the algorithm adds every new string it sees to the string table. To compress, the algorithm scans the input data for the longest matching string in the string table and outputs the index of that string as the result of the compression. Compression occurs when a long string of characters is replaced by a shorter index.

One difficulty in using LZW compression on large data files is in managing the dictionary, as the size of the string table often surpasses that of available memory. Here we propose a new method called table pruning for managing the dictionary. We have demonstrated our method with least recently used and aging replacement algorithms and improved the compression ratio obtained from using LZW alone. Finally, we discuss some factors we observed to be crucial to compression ratios.

2. Handling the Ever Growing String Table

One drawback to be considered in implementing the LZW algorithm is the ever-growing string table; as more data is analyzed the dictionary becomes increasingly large. The table must be managed, as computer memory is limited. Two existing methods for handling the ever-growing string table [1], [9] are discussed below.

2.1 Table Freezing

This is the method used by the original LZW algorithm. This method picks a size of the string table and does not allow the table to grow beyond that size. Instead, it continues the compression according to the frozen table. It is simple and easy but it doesn't work well with large files.

2.2 Table Flushing

This is the method used in [9]. This method computes the current compression ratio periodically. When the table is full and the current compression ratio drops below some predetermined threshold value, it flushes the string table. That is, the algorithm abandons the current string table and builds a new one when compressing the remaining input data.

Flushing can get rid of infrequently used entries. However, this drastic operation also flushes out frequently used entries. Thus, it doesn't improve compression ratios for a lot of input files.

2.3 Table Pruning

We propose to prune the string table. Once the string table becomes full and an additional entry is needed, we replace an infrequently used entry with the new entry and the compression continues. However, the problem of selecting an infrequently used entry for pruning is non-trivial.

3. Selecting an Infrequently Used Entry for Replacement

Many strategies exist for selecting infrequently used entries, a problem similar to selecting replacement pages for virtual memory management systems. Here we utilize principals from two of these so-called "page replacement algorithms": Least Recently Used and Aging Replacement.

3.1 Least Recently Used (LRU)

In LRU, the entry which has not been accessed for the longest is selected as the replacement entry. In our implementation, we use a self-organizing list to select the least recently used entry. This list contains an index to every entry of the string table. During the compression, every time an entry is accessed, the corresponding index is moved to the front of the list. When a replacement entry is needed, it's selected from the end of the list.

3.2 Aging Replacement

In addition to LRU, we use the aging replacement algorithm to manage the string table. In this algorithm, we keep a value called time to live (TTL) for every table entry. When an entry is created the corresponding TTL is initialized to some predetermined value. Periodically, the TTL is decreased. When the TTL becomes zero, the entry is deleted from the string table. In order to let table accesses closer to the present time have more impact than table accesses long ago, when an entry is accessed, its TTL is reset to (current value/2+initially value). When a replacement entry is needed, an unused entry or the one with the smallest TTL will be selected.

4. Implementation Complicatedness

The implementation of our idea is somewhat complicated mainly due to the representation and management of the string table.

In order to speed up the process of searching the string table, the double hashing technique is used to implement the string table. In order to achieve a good performance of the hash table, the size of the hash table is 25% bigger than the needed size of the string table.

Because of hashing, deleting or replacing entry of the string table cannot be done directly. To replace an entry, we need to mark an entry as deleted and use an unused entry for the new entry. Because of this, we need to clean up marked entries before the hash table gets full. To do so, we need to recreate the hash table periodically.

Moreover, if LRU algorithm is used to select infrequently used entries, a linked list is added to implement the self-organizing list. If the aging replacement algorithm is used, a heap is added to accelerate the process of finding the entry with the smallest TTL.

5. Factors That Affect the Compression Ratio

We found the following factors to be crucial to the resulting compression ratio, the ratio of the compressed file size to the original file size.

5.1 The maximum size of the string table

The maximum size of the string table determines the number of bits needed to represent a code word, i.e. an index to the string table. The larger the size the greater number of bits will be required to represent an index. To compress a small file, a smaller table results in a smaller compressed file. To compress a large file, a smaller table holds less strings and thus less chance of using an index to encode a long string of characters and thus reduce the compression efficacy. Algorithms in [7]-[9] reduce the size of the compressed file by using variable length tables. According to [9], the maximum number of bits can be saved is 3840. For large files with millions of bytes, this is insignificant.

To fully utilize all possible combinations of bits of compressed codeword, the size of the string table is a power of 2. After experimenting with different table sizes ranging from 2^{12} through 2^{22}, we found that a table of size 2^{16}, i.e. 65536 works well with large text files.

5.2 The period of recreating the hash table

The hash table must be recreated before the hash table becomes full. However, if the table is recreated too often, the program speed is greatly decreased. Moreover, according to our observations, different lengths of period result in different compression ratios.

According to our study, for a table of size 65536, the optimal period to recreate the string table is after compressing 4096 strings.

5.3 The interval of decreasing TTLs

Recreating the hash table is a time consuming process in which every entries of the table must be accessed. In order to reduce the speed impact of managing the hash table, we paired the task of recreating the hash table with the task of decreasing TTLs. That is, recreating the hash table and decreasing the TTLs are done at the same time.

5.4 The initial value of TTLs

If the initial value of TTLs is too small, many entries of the string table will be deleted too soon and thus the table pruning method has the same draw back as the table flushing method.

After some experiments, we found the optimal initial TTL value to be the size of the table divided by 1024. That is, for a table of 65536, the best initial TTL value is 64.

6. Emperical Results

To evaluate the effectiveness of our methods, we test our methods with test files from the web site Canterbury Corpus. (http://corpus.canterbury.ac.nz). The Canterbury Corpus is a benchmark to enable researchers to evaluate lossless compression methods.

We present our results in the following tables. The three test files *E.coli*, *bible.txt* and *world192.txt* are in the large corpus collection of the Canterbury Corpus. In these experiments, we have used string tables of size 65536, hash table recreating period of 4096, and TTL initial value of 64.

Table 1: Compressed file sizes

	E.coli	bible.txt	world192.txt
Original file size (bytes)	4,638,690	4,047,392	2,473,400
LZW	1,213,588	1,417,762	925,826
LZW/ Aging	1,199,245	1,242,153	804,493
LZW /LRU	1,234,866	1,291,120	850,560

Table 2: Compression ratios

	E.coli	bible.txt	world192.txt
LZW	3.82	2.85	2.67
LZW/aging	3.87 (+1%)	3.26 (+12%)	3.07 (+13%)
LZW /LRU	3.76 (-1%)	3.13 (+9%)	2.91 (+8%)

Besides the test files from The Canterbury Corpus, we have also tested our methods with other text files. Compression tests on these files yielded the following findings:

- LZW/aging does better than LZW/LRU 90% of the time.
- LZW/aging can improve the compression ratio over LZW by 10-15% for 90% of the files tested.

Preliminary tests of our methods with video and image files also gave promising results. The original LZW consistently inflate video and image files by about 25%. Our LZW/aging can deflate video and image files by 1% consistently. In other words, LZW/aging can improve the compression gain by 26% for large video or image files over the original LZW.

7. Decompression

Decompression is a simple task relative to compression. Since there is no need to search the string table, the hashing technique is not required and thus there is no need to recreate the hash table periodically. However, a heap or a self-organizing list is still needed for LZW/aging and LZW/LRU respectively. The purpose of including a heap or a self-organizing list is to synchronize the decompression string table with the compression string table so the two tables use the same sequence of replacement entries.

8. Conclusions

We have described an improvement of LZW data compression which use table pruning techniques. With more efficient management of the dynamic dictionary, a better compression ratio may be achieved. Specifically, we show that LZW/aging can significantly improve the compression ratio for most large files.

According to our experiments, we identified four factors that are crucial to the compression ratios of LZW/aging and LZW/LRU. These factors are the size of the string table, the period of recreating the hash table, the interval of decreasing TTLs and the initial value of TTLs. Further work needs be done to characterize the combinatorial effects of these factors and determine their optimal combinations.

While the aging algorithm provided considerable improvement over LZW compression alone, additional replacement algorithms should be explored. Finally, we will explore more on how the compression methods perform on different types of data files such as video and image files.

9. References

[1] Ziv, J. and Lempel A. 1977. “A universal algorithm for sequential data compression”. IEEE Trans. Inf. Theory 23, 3

(May), 337-343.
[2] Ziv, J., & Lempel, A. 1978. “Compression of individual sequences via variable-rate coding”, IEEE Trans. Inform. Theory, 24(5), 530-536.
[3] Storer, J.A., & Szymanski, T.G. (1982) “Data Compression via Textual Substitution,” Journal of ACM, 29(4), 928-951.
[4] Welch, T. A. 1984. “A technique for high-performance data compression”. Computer 17, 6 (June), 8-19.
[5] Willard, L., Lempel, A., Ziv, J. & Cohn, M. (1984) “Apparatus and method for compressing data signals and restoring the compressed data signals”, US patent - US4464650.
[6] Horspool, R.N. (1991) "Improving LZW," Proc. Data Compression Conference (DCC 91), Snowbird, Utah, IEEE Computer Society Press, Los Alamitos, CA, pp. 332-341.
[7] Ouaissa, K., Abdat, M. and Plume, P. 1995. “Adaptive limitation of the dictionary size in LZW data compression”. Proceedings 1995 IEEE International Symposium on Information Theory.
[8] Chai, Z. and Chen W. 2004. “An adaptive LZWCompression algorithm using changeable maximum-code-length”. Fourth International Conference on Computer and Information Technology (CIT'04) pp. 1175-1180.
[9] Raghuwanshi, B.S., Jain, S. Chawda, D. and Varma, B. 2009. “New dynamic approach for LZW data compression”. IJCNS Vol. 1, No. 1 (October), 22-26.