

# **An Asymmetric, Semi-adaptive Text Compression Algorithm**

**Harry Plantinga**

Department of Computer Science  
University of Pittsburgh  
Pittsburgh, PA 15260  
planting@cs.pitt.edu

## **Abstract**

A new heuristic for text compression via digram replacement and a new, fast entropy coding method are described and implementation results are presented. The resulting compression algorithm is asymmetric in the sense that compression requires much time and memory, but decompression is fast and requires little memory. Compression ratios achieved are competitive with gzip for a standard corpus of texts, and better for large files. Finally, since the algorithm is semi-adaptive, random access into the compressed file is available without decompressing the whole file. Thus, the algorithm is well suited to applications for which expensive compression of large files is acceptable but decompression must be inexpensive, and high compression ratios or random access into the compressed file are required.

## Introduction

Most text compression algorithms currently in use adapt their performance according to the statistics of the file seen so far. Adaptive algorithms have many advantages. Files with statistics that vary across the file may be compressed more compactly with an adaptive algorithm, and a dictionary need not be stored along with the compressed text. However, such techniques require reading through the whole file up to a certain point in order to decompress text starting at that point. They are typically used by decompressing a complete file before accessing any part of it.

For some applications, such an algorithm is not practical. For example, a new class of hand-held computing devices is becoming available and large quantities of information such as electronic books or databases will be distributed for and accessed with these devices. For such applications, text compression is desirable if decompression is fast enough and memory requirements are small enough, but there may not be enough memory available to decompress the whole file. For these applications, random access into the compressed file without decompressing the whole file is required. In that case, a more appropriate type of compression algorithm is a *semi-adaptive* algorithm, which adapts to a file but not across the file. Random access into a compressed file may be simulated with adaptive algorithms by breaking large files into smaller pieces before compression, but then compression ratios suffer.

In this paper I present a new approach to text compression called the *digram dictionary* algorithm, that is well suited to such applications. It is *asymmetric* in the sense that it requires a large amount of time and memory to compress files, but decompression is competitive with the fastest algorithms and requires little memory. Small parts of a file compressed with this algorithm may be accessed and decompressed without decompressing the rest of the file. Experimental results show that for a standard corpus of texts [Bell *et al.* 1989], the compression ratios achieved are competitive with and in some cases better than those for the best of the adaptive dictionary-based techniques, especially for large files. Some statistical modeling / arithmetic coding techniques achieve better compression ratios, but they require much more time and memory for decompression and they do not admit random access into the compressed file.

The main idea of the algorithm is straightforward: level-1 statistics for the file are computed, and a digram (pair of tokens, initially bytes) is chosen and replaced throughout the file with a new token. The two replaced tokens are entered into the dictionary at a location indexed by the new token value. This process is repeated until further compression of the file is no longer possible. The result is a dictionary of digrams and a text file of tokens representing digrams in the dictionary. The dictionary and text are then entropy coded. Thus, each token in the text may represent a long string of characters when uncompressed. Decompression consists of reading a token and recursively looking it up in the dictionary. The general approach is not new. Finding the best dictionary is NP-hard [Storer and Szymanski 1982], and many heuristics have been proposed [Lynch 1973; Mayne and

James 1975; Rubin 1976; Schuegraf and Heaps 1973; Wagner 1973; White 1967; Wolff 1978].

The novelty of the approach presented here is in the heuristic used for selecting digrams for replacement and the entropy coding method. The digram chosen for replacement is the one that is estimated to most improve the compression ratio according to an entropy measure. The entropy coding scheme has been designed primarily for speed of decompression; it is likely that Huffman coding could improve the compression ratios slightly at the expense of decompression speed. The entropy coding scheme uses 16 fixed-length codes with the number of entries per code for all but the last one equal to a power of two. For each token, a pair  $(p, q)$  of numbers is used to encode the token.  $p$  is a 4-bit *code number* specifying the fixed-length code, and  $q$  specifies the element of the code.

The algorithm compresses slowly. For the standard corpus consisting of 14 files constituting 3.1 MB, compression required more than 15 hours on a RISC workstation. The memory required may be 3-4 times the size of the file. However, decompression is very fast, and the memory requirements are typically a few percent of the size of the uncompressed file—enough memory to store the dictionary. Compression ratios achieved are relatively better for large files and for files with constant statistics (since the algorithm does not adapt across a single file); for large text files such as electronic books, the algorithm performed significantly better than the best LZ-based techniques on the files tested.

### The Digram-Dictionary Text Compression Algorithm

The Digram-Dictionary algorithm works by repeatedly replacing all occurrences of a digram in the text with a single new token. The success of the method depends on being able to make a good estimate of the effect on the compression ratio of a proposed digram replacement. This is estimated by using an entropy computation to estimate the number of bits that would be saved for the combined text and dictionary *after entropy coding*. For a file consisting of tokens indexed by  $i$ , with relative frequencies  $p_i$ , the entropy of the file is given by

$$-\sum p_i \log p_i \quad (1)$$

Therefore, the number of bits required to represent the file according to the entropy estimate is

$$\sum c_i (\log n - \log c_i) \quad (2)$$

where  $n$  is the size of the file and  $c_i$  is the number of occurrences of token  $i$ . This can be rewritten

$$n \log n - \sum c_i \log c_i \quad (3)$$

Now suppose that tokens  $j$  and  $k$  are replaced with a new token  $m$ . Let  $c_m$  be the number of occurrences in the text of token  $m$ . The new length of the text and

dictionary will be  $n' = n - c_m + 2$ , since two tokens will be added to the dictionary. The new number of occurrences of tokens  $j$  and  $k$  will be  $c_j' = c_j - c_m + 1$  and  $c_k' = c_k - c_m + 1$ . The number of bits saved is estimated as the decrease in the number of bits required according to the entropy estimate, namely

$$n \log n - n' \log n' - c_j \log c_j + c_j' \log c_j' - c_k \log c_k + c_k' \log c_k' + c_m \log c_m \quad (4)$$

In order to find the best digram for replacement, first-order statistics of the file must be computed. This is accomplished by inserting all digrams into a hash table and maintaining a count for each. Then for all digrams in the hash table, the number of bits that would be saved upon replacement is computed according to Eq. (4). The best digram is selected for replacement. Note that the best digram is usually not the most common digram, since very common tokens require a small number of bits to represent. Digram-Dictionary encoding proceeds as in Figure 1:

---

```

repeat
  count the occurrences of all digrams
  find the digram that would yield best savings
  replace all occurrences of the digram in the text with a new token
  add an entry in the dictionary indexed by the new token,
    containing the two tokens being replaced.
until savings are no longer possible.
```

---

**Figure 1. Digram-dictionary compression.**

## K-Coding Entropy Coding

In principle, any static entropy coding method could have been selected. However, since the design goals for this algorithm were maximum speed and small memory requirements for decompression, a new entropy coding method called  $K$ -Coding was devised. It does not achieve optimal entropy coding in the sense of Huffman coding, but it also does not require bit-level operations on the compressed data for decompression. Decompression involves reading two variable-length words for each token in the compressed text. Additionally, if a digram dictionary is to be stored in memory in any case, the additional data structure requirements are minimal—two  $K$ -element arrays, in this implementation, with  $K = 16$ .

The idea behind  $K$ -coding is that  $K$  fixed-length codes are selected. Each token is encoded as a pair  $(p, q)$ , where  $p$  is a  $K$ -bit number specifying the code and  $q$  is a number specifying an element within the code. The length of  $q$  depends  $p$ . For example, suppose the tokens to be represented are shown in Table 1.

Token	Freq.
5	.28
3	.23
6	.15
2	.13
4	.11
8	.07
1	.03

**Table 1. Tokens to be encoded and relative frequencies.**

This set of tokens could be represented with a 2-Code, i.e. with 4 fixed-length codes. The first two codes would contain one item each, so the length of  $q$  would be zero bits. The third code would contain two items, so the length of  $q$  for the third code would be 1 bit. The fourth code must represent the remaining three items, so two bits are required. Since  $p$  is two bits long, 5 and 3 would be encoded with two bits total, 6 and 2 with three bits, and 4, 8, and 1 with four bits.

Token	$p$	$q$
5	00	
3	01	
6	10	0
2	10	1
4	11	00
8	11	01
1	11	10

**Table 2. The 2-Coding of the tokens.**

The particular 2-Code used would be represented by writing the number of bits for  $q$  in each code element, in this case, (0, 0, 1, 2), and the number of different tokens, in this case 7. Note that smallest code length with which a token can be represented using  $K$ -coding is  $K$  bits.

The prototype implementation uses 4-Coding. The 4-code is represented in the compressed file with a sequence of 16, 5-bit numbers and another number representing the total number of dictionary entries. The tokens in the text and dictionary are renumbered in decreasing order of frequency of occurrence, and the dictionary and text are encoded. Byte values are encoded by using a "literal" escape code as the first dictionary entry and the literal byte value as the second entry.

The method used to determine the length in bits to assign to each of the 16 codes is guided by the principle that each of the codes should represent approximately 1/16 of the file. Initially, the first 1, 2, 4, 8, ... tokens are selected and the total number of occurrences of those tokens is computed, until the number as close to 1/16 of the file as possible is found. For the second code length, a similar test is used to find the number of tokens resulting in approximately 1/15 of the file, and so on. The last code must be long enough to represent the rest of the tokens. Encoding is then a matter of re-numbering all of the tokens in the file in decreasing order of relative frequency, and for each token, determining ( $p$ ,  $q$ ) for that token.

Static Huffman coding could also have been used. If the dictionary entries were sorted in decreasing order of token occurrence frequency, the Huffman code could be represented implicitly by writing the number of tokens represented with 1 bit, 2 bits, and so on, up to the length of the longest token's encoding. A canonical Huffman tree could then be constructed for decoding. It is likely that static Huffman coding would improve compression ratios slightly at the expense of slightly slower decoding and a more complex data structure for decoding.

## Experimental Results

This algorithm has been implemented in C and tested on Macintosh and DEC work stations. Little effort has been taken in optimizing the runtime of the compression or decompression prototype code.

Since the smallest number of bits used to represent a token with 4-coding is four, the method as described above does not perform well for files in which after compression there is a token constituting significantly more than 1/16 of the file. Therefore in the prototype a second stage of compression was added. The second stage is exactly like the first except that in the estimate of the number of bits saved, a minimum of four bits per token is assumed. This change improved compression ratios for some files slightly and for "pic" from more than four bits per byte to 0.85 bits per byte. The drastic change in that case is due to the fact that after compression, the file consisted mostly of zero bytes, each of which was represented with four bits.

File	Size	compress	gzip	comp-2 -o 4	DD
bib	111261	3.35	2.52	2.02	2.53
book1	768771	3.46	3.26	2.35	2.69
book2	610856	3.28	2.71	2.08	2.39
geo	102400	6.07	5.35	5.11	4.67
news	377109	3.86	3.07	2.56	3.02
obj1	21504	5.23	3.84	4.01	4.41
obj2	246814	4.17	2.65	2.62	3.12
paper1	53161	3.77	2.80	2.48	3.08
paper2	82199	3.52	2.90	2.45	2.85
pic	513216	0.97	0.88	0.88	0.85
progc	39611	3.87	2.68	2.60	3.11
progl	71646	3.03	1.82	1.85	2.62
progp	49379	3.11	1.82	1.86	2.73
trans	93695	3.27	1.62	1.64	2.20
Total:	3141622	1259141	1061830	848885	965620
Avg:		3.64	2.71	2.47	2.82
Large file avg:		3.59	2.92	2.52	2.75
Comp. time		0:00:27	0:01:01	0:08:09	15:53:10
Decomp. time		0:36	0:35	8:18	0:30

**Table 3. Compression results for several algorithms.**

The corpus of texts is that used by Bell *et al.* [1989]. The version of the UNIX compress program used is 4.0 and the version of gzip is 1.2.3. comp-2 is an experimental implementation [Nelson, 1991] of Prediction by Partial Match and Arithmetic Coding [Cleary and Witten, 1984; Moffat 1988], using 4th-order statistics. All runtimes presented are the total times reported by the *time* facility on a DECstation 5000/200. The large file averages above are for the seven files of size greater than 100KB.

While compression ratios overall were 4% worse than those for gzip on average, compression ratios for files larger than 100KB were 6% better. However, the main advantage of this approach over adaptive algorithms for some applications is the availability of random access into the compressed text without decompressing the whole text. Compression times are much longer than for any of the other algorithms tested. With the DD algorithm, compression becomes an overnight process for files in the 1MB size range. However, this is not a crucial problem when other requirements outweigh the compression time, as for example when compressed files are to be distributed widely on read-only media. Decompression times were slightly better than those for the other dictionary-based algorithms tested.

The Digram Dictionary compression technique therefore seems well suited to applications such as mass distribution of large files, especially when random access into the compressed files is desirable. Decompression is fast and memory requirements for decompression are not large.

### References

- Bell, T., I. H. Witten, and J. G. Cleary, "Modeling for text compression," *ACM Computing Surveys* **21**(4), December 1989, pp. 557-592.
- Cleary, J. G., and Witten, I. H., "Data compression using adaptive coding and partial string matching," *IEEE Trans. Commun. COM-32*, 4, April 1984, pp. 396-402.
- Lynch, M. F., "Compression of bibliographic files using an adaptation of run-length coding," *Inf. Storage Retrieval* **9**, 207-214.
- Mayne, A., and E. B. James, "Information compression by factorizing common strings," *Comput. J.* **18**(2), 1975, pp. 157-160.
- Moffat, A., "A note on the PPM data compression algorithm", Res. Rept. 88/7, Dept. of Computer Science, Univ. of Melbourne, Victoria Australia.
- Nelson, M. R., "Arithmetic Coding and Statistical Modeling," *Dr. Dobb's Journal*, February, 1991, p. 16.
- Rubin, F., "Experiments in text file compression," *Commun. ACM*, **19** (11), 1976, pp. 617-623.
- Schuegraf, E. J. and H. S. Heaps, "Selection of equiprevalent word fragments for information retrieval," *Inf Storage retrieval* **9**, 1973, 697-711.
- Storer, J. A. and T. G. Szymanski, "Data compression via textual substitution," *J. ACM* **29** (4), October 1982, pp. 928-951.
- Wagner, R. A., "Common phrase and minimum-space text storage," *Commun. ACM* **16** (3), 1973, pp. 148-152.
- White, H. E., "Printed English compression by dictionary encoding," *Proc. IEEE* **55** (3), 1967, pp. 390-396.
- Wolff, J. G., "Recoding of natural language for economy of transmission or storage," *Comput J.* **21** (1), 1978, pp. 42-44.