ADVANCED LOSSLESS TEXT COMPRESSION ALGORITHM BASED ON SPLAY TREE ADAPTIVE METHODS

RADU RĂDESCU, ANDREEA HONCIUC*

Key words: Data compression, Splay Tree, Prefix, Compression ratio.

This paper presents an original version of Splay Tree for lossless compression, a self-adjusting form of binary search trees. It is developed and analyzed in its new form, and then is compared in terms of text compression performance to other algorithms and transforms.

1. **INTRODUCTION**

Splay Trees are usually considered forms of lexicographically ordered binary search trees [1, 2]. The motivation of this paper comes from the fact that the search trees have multiples drawbacks. These data structures are created to reduce the worst-case time per operation. However, in typical applications of search trees [3] are performed several operations, not a single one, and what matters is the total time necessary for the operations, not the individual times of each of them. In such applications, a better goal is to reduce the amortized time of operations, where *amortized time* means the average time of an operation in a worst-case sequence of operations [4]. One way to obtain amortized efficiency is to use a *self-adjusting* data structure. The structure can be in an arbitrary state but, during each operation, a simple restructuring rule is applied to improve the efficiency of the following operations. Self-adjusting data structures have several potential advantages over other balanced structures or with other constraints:

- in an amortized case, when the constant factors are ignored, they are never much worse than constrained structures and since they adjust according to use, they can be more efficient if the pattern used is skewed;
- requires less space, since no balance or constraint information is stored;
- access and update algorithms are easy to implement with a simple concept.

Self-adjusting structures also have some drawbacks:

- they require more local adjustments, especially during accesses;
- individual operations within a sequence can be expensive, which may be a disadvantage especially in real-time applications.

* "Politehnica" University of Bucharest, E-mail: radu.radescu@upb.ro, andreea.honciuc@gmail.com.

2. COMPRESSION USING SPLAY TREES

In an adaptive coding [5], there is no need for pre-transmission statistics and there is no need for two passes. Instead, both the encoder and decoder take some initial probability distribution of symbols and then, when transmitting the message, can change their knowledge after treating the last message symbol or until is too late for the information to be exploited [6].

Cleary and Witten [7] concluded that there is no associated loss of compression using adaptive coding. The need for probability distribution details transmission means that an information message is a limitation that is expected to be achieved if the message is long, but it will not be equaled.

2.1. PREFIX CODES – THE HUFFMAN CODE

The most studied data compression algorithms are probably those based on Huffman codes [8]. In a Huffman code, each source letter is represented in the compressed text by a variable length code. Common source letter are represented by short codes, while uncommon ones are represented by long codes. The codes used in the compressed text must obey the prefix property, that is, a code used in the compressed text may not be a prefix of any other code.

Prefix codes can be thought of as trees with each leaf of the tree associated with a letter of the alphabet source. The figure bellow illustrates a prefix code for a 4-letter alphabet. The prefix code for a letter can be read by following the path from the root of the tree to the letter and associating a 0 with each left branch followed and a 1 with each right branch followed. The code tree for a Huffman code is a weight balanced tree, where each leaf is weighted with the letter frequency and internal nodes have no intrinsic weight. The example tree would be optimal if the frequencies of the letter A, B, C and D were: 0.125, 0.125, 0.25 and 0.5, respectively.

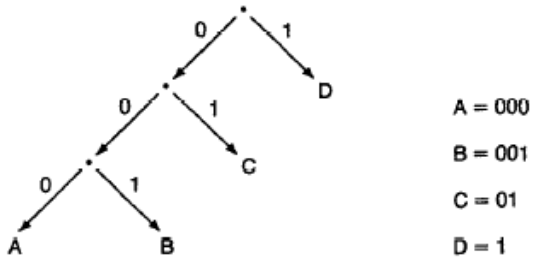

Fig. 1 – A tree representation of a prefix code.

Conventional Huffman codes require either prior information of the letter frequencies or two passes through the data to be compressed – one to obtain the letter

frequencies and one to perform the actual compression. In the latter case, the letter frequencies must be included with the compressed text in order to allow for later expansion. Adaptive compression algorithms operate in a single step. In adaptive Huffman codes, the code used for each letter in the source text to be compressed is based on the frequencies of all letters above, but not including that letter. The basis for efficient implementation of adaptive Huffman codes was established by Gallager [9].

2.2. APPLICATION OF SPLAY TREES TO DATA COMPRESSION

Splay-prefix algorithm is one of the simplest and fastest adaptive compression algorithms based on the use of prefix codes. Data structures used in the splay-prefix algorithms can be also applied on arithmetic data compression.

Data compression algorithms can improve the efficiency with which data is stored or transmitted by reducing the amount of redundant data. A compression algorithm takes a source text as input and produces the original source text as output [10]. Most compression algorithms view the source text as consisting of a sequence of letters selected from an alphabet.

Although there are a number of ad hoc approaches to data compression (*e.g.* run length encoding), there are also a number of systematic approaches. Huffman codes are among the oldest of the systematic approaches of data compression [11]. Adaptive Huffman compression algorithms, such as FGK [12] and Vitter [13], require the use of tree balancing schemes [14], which can also be applied to the data structures required by adaptive arithmetic compression algorithms. The present papers states that there is sufficient similarity between the balancing objectives of these schemes and those achieved by Splay Tree to try Splay Trees in both contexts with good results.

Splay Trees are usually considered forms of lexicographically ordered binary search trees, but the trees used in data compression need to have a static order [15]. The removal of the ordering constraint allows the basic splaying operation to be considerably simplified, as showed in the next section, the resulting algorithms being extremely fast and compact. The present paper's original contribution consists in the following statement: when applied to Huffman codes, splaying leads to a locally adaptive compression algorithm that is remarkably simply as well as fast, although it does not achieve optimal compression. When applied to arithmetic codes, the result is near optimal in compression and asymptotically optimal in time.

3. **EXPERIMENTAL RESULTS**

In this chapter, the algorithm will be tested relative to other compression methods, to make it fit in terms of performance and to determine when its use is optimal. Test corpora consist of *Calgary* and *Canterbury Corpora*. They are

composed of a collection of files designed specifically for test applications lossless compression methods. Calgary corpus has 18 type files totalling more than 3.2 million bytes [16].

Calgary corpus was collected in 1987 by several researches to develop, test and compare different methods of compression. Since it worked for a long period on the same case, it was assumed that some methods of compression have been made to be optimal for this case. Standard file format has changed in recent years, has therefore developed a new case: Canterbury [17].

Table 1

Calgary corpus files description

Name	Size (bytes)	Description	Type
Bib	111 261	Bibliography in UNIX format	Text
Pic	513 216	Image	Image
Progc	39 611	C program source	Source

Table 2

Canterbury corpus files description

Name	Size (bytes)	Description	Type
Cp.html	24 603	HTML source	Source
kennedy.xls	1 029 744	Excel Spreadsheet	Excel Document
Ptt5	513 216	CCITT test set	Text –Fax type
aaa.txt	100 000	‘a’ letter repetead 100 000	Text
alphabet.txt	100 000	Alphabet repetitions	Text
random.txt	100 000	100 000 random characters	Text
E.coli	4 638 690	Complete genome of the E.coli bacterium	Text
world192.txt	2 473 400	The CIA world fact book	Text

Table 3

Artificial corpus files description

Name	Size (bytes)	Description	Type
Btws.exe	10 111 48	Burrows-Wheeler transformation executable	Executable
ANN.pdf	2 501 094	Book describing artificial neural networks	PDF
Grass.jpg	27 060	Color image	Image
Negru_mic.bmp	327 474	Black and white image	Image

In addition to these traditional corpora, it is interesting to study the behaviour of algorithms on other corpora. It has been made such a selection on other types of files held in an artificial body. Some of the chosen files – especially the executable and .pdf files – have large dimensions. Next, will be detailed the results obtained after their compression and decompression, some features being remarkable.

3.1. COMPRESSION RATIO, COMPRESSION & DECOMPRESSION TIME

To illustrate the compression process using the Splay Tree algorithm, multiple files of available corpora were compressed. After coding, the results for the corpora files available were obtained according to Table 4.

Table 4

Compression ratio for the corpora files

File	Compression ratio	Compression time	Decompression time
bib	1.2772	0.62	0.1
pic	4.6791	0.2	0.12
progc	1.3512	0.65	0.08
cp	1.2993	0.07	0.06
kennedy	2.0537	0.5	0.5
ptt5	4.6791	0.18	0.15
E.coli	3.3072	0,9	0.71
world192	1.3872	1.07	0.78
XML	1.5883	0.15	0.14
Aaa	7.9955	0.04	0.04
Alphabet	1.4959	0.12	0.07

It is noted that the best results were obtained for E.coli file, .pic type file and for ptt5 file. The E.coli file contains the complete genome of E.coli bacterium, thus a file where some characters have high probabilities of occurrence. ptt5 file is similar case in which some characters have high probabilities of occurrence. There is thus a high probability that they lie near the root or even are the tree roots.

In image, is rarely that many consecutive pixels of a line have the same intensity, but in a textured region of the image, can be used a static probability distribution to describe the intensities distribution. As the algorithm compress consecutive pixels in a line, it assigns the pixel intensities common short codes in current context. When it is moved from one textured region to another, short codes are assigned to the common intensities in the new region, while the codes for the unutilized intensities become now more widely.

For the other file types, compression ratio is close to 1.5 and is due to a very large number of characters that have the same frequency. It can also be seen that if the original alphabet is reduced, the compression ratio will increase.

The files from Canterbury Corpus have special properties unlike natural files. Thus, „aaa.txt" file contains the letter 'a' repeated 100 000 time, and the file „alphabet.txt" contains the alphabet letters repeated until is reached 100 000 characters. As can be seen from previous tables, the larger compression is obtained for „aaa.txt" file. However, a compression ratio of up to 10 is considered very high.

After $\log_2 n$ repetitions of a letter from an n letter alphabet, the algorithm will assign a code of 1 bit of that letter. Compression ratio is therefore limited to N.

This explains the results of the aaa.txt file. In this case, are read symbols of 8 bits, so can be get a maximum compression ratio of 8/1. Since the tree storage is necessary in the destination file, it will get a compression ratio of less than 8.

Moreover, if the letters in a sub-tree of the code tree are referenced repeatedly, the algorithm will shorten the codes for all letters in that sub-tree.

Instead, a small compression ratio is obtained in „alphabet.txt" file case, because there are characters with higher frequencies than others.

For the artificial corpus, the obtained results are presented in Table 5.

Table 5

The compression ratio for the artificial corpus files

File	Compression ratio	Compression time [s]	Decompression time [s]
ANN.pdf	0.8895	1.46	1.14
BWTS.exe	7.5799	0.781	0.73
negru_mic.bmp	7.8316	0.07	0.06
Grass.jpg	0.8394	0.07	0.06

It is noted that the times compression are very low, which shows that Splay Tree algorithm is extremely fast both in compression and the decompression. However, for larger files, the algorithm has time compression slightly higher. For example, for the E.coli and world192 files unlike the other files in the same corpora time compression is not much higher than the one of the other files. It may be noted that E.coli file compression only lasts for more than one second.

For files from the arbitrary corpus, the time compression is slightly higher for larger size files. For example, the executable BWTS.exe file and the ANN.pdf file with sizes of 10 111 488 and 2 501 094 bytes, the Splay Tree method records a time compression of 1.46 s and 0.7 s while for smaller files records lower times compression.

These low times are resulted from the fact that the algorithm makes just only one pass through the file, and not two or more as needed to statistical methods. In the case of decompression time, the algorithm has a similar reaction: the times are low and have a certain dependence on the size of the compressed files.

3.2. IMPROVMENTS USING PRECOMPRESSION TRANSFORMS

One way to improve the compression files is to apply transforms before lossless algorithms [18]. This preprocessing modifies the file structure so that the new structure can be applied to other compression methods.

3.2.1. Improving compression using the burrows-wheeler and move-to-front transform

Burrows-Wheeler transform rearranges the data set using a sorting algorithm. The transformation result contains the same data processing, but arranged differently.

The transformation is reversible, without loss of information. Since the data set may be too large to be full processed at a time, is recommended subdividing it into blocks and processing them sequentially. Processing block size must be large enough to exploit the source redundancy [19].

Burrows-Wheeler Transform application improves compression to time compression increase detriment. As for large size files, the compression time is high. the use of such transforms is justified. A further improvement is the application of Move-to-Front Transform on the file already transformed with Burrows-Wheeler Transform. Move-to-Front Transform is efficient for individual application. because typically increase the symbol occurrence frequency.

For a compression using this method, the algorithm is as follows: on a given text is applied the Burrows-Wheeler Transform. resulting in a new text with rearranged characters in a suitable form for the Move-To-Front encoding (frequently appear sequences with identical characters) [6]. The result of the Move-to-Front encoding is a representation of the original text in which the repetition cases have been replaced with very low or even zero values. Then will be applied the compression operation on the output file from the MTF transformation.

The obtained results are presented in Table 6.

Table 6

Compression ratio after the BWT application

File	Compression ratio without BWT	Compression ratio with BWT	Compression gain [%]	Compression ratio with BWT and MTF	Compression gain [%] over Splay using BWT+MTF	Compression gain [%] over BWT using BWT+MTF
pic	4.67916	5.2107	10	4.8237	10	7
progc	1.3512	2.7954	52	2.6543	52	5
cp	1.2993	2.7597	53	2.6084	53	5
kennedy	2.0537	3.5373	42	5.5521	42	–57
ptt5	4.6791	5.2107	10	4.8237	10	7
E.coli	3.3072	3.4384	4	3.3351	4	3
world192	1.3872	3.8993	64	3.6483	64	6
aaa	7.9955	7.9955	0	7.9955	0	0
XML	1.5883	7.4090	79	7.5638	79	–2

The results show a great improvement for the XML file, where it has reached a compression ratio of 7.56 and especially for kennedy.xls file. This improvement appears from the identical character alignment, which in Splay Tree can be found at the top of the tree. However, for some files such as XML and Excel file. only the Burrows-Wheeler Transform has a superior behavior over the BWT+MTF application. For example, for „kennedy.xls" file is not obtained a compression gain and is thus more efficient to use just the BWT Transform.

The only files for which do not appear great improvements are E.coli and aaa.txt. In these files, from the beginning, it exists a large number of characters of high frequencies, and, after the transform application, results have not improved significantly because there were obtained other characters with frequencies close to the original (for example, in aaa.txt file the characters will be the same).

However, compression gain is quite high for some files, although the transformation only orders in a different way the characters in files, unchanging their frequencies. However, even this sorting has benefits especially on files with characters with similar frequencies. It concludes that for files containing the same characters the compression gain is reduced. In compression algorithms based on context, transforms change the original structure, decreasing hereby the performance.

For the files from the arbitrary corpora, the results are presented in Table 7.

Table 7

Compression ratio of the files from the arbitrary corpora

File	Compression ratio without BWT	Compression ratio with BWT	Compression gain	Compression ratio with BWT and MTF	Compression gain over Splay using BWT+MTF	Compression gain over BWT using BWT+MTF
ANN	0.8895	0.9944	11%	0.9884	11%	1%
BWTS	7.5799	7.7132	2%	7.6839	2%	0%
little black	7.8316	7.7936	0%	7.7853	0%	0%
grass	0.8394	0.8522	1%	0.8276	1%	3%

In conclusion, the simultaneous application of transforms Move-to-Front and Burrows-Wheeler is justified for text files especially, but with not spectacular results, in the next order: Burrows-Wheeler Transform and the Move-to-Front.

3.2.2. Comparison between different compression algorithms

Compression algorithm performances are evaluated by using Splay Tree algorithm relative to other compression algorithms. We choose to test an arithmetic coding algorithm, an algorithm that combines Huffman trees coding method with Lempel-Ziv method and a commercial algorithm that uses different methods to optimize compression performances (WinRAR).

The obtained results are presented in Table 8.

From Table 8 it can be seen that the algorithms have a very good reaction on Canterbury Large corpus files, for example aaa.txt and alphabet.txt and the XML file. However, for aaa.txt file Splay Tree algorithm has a compression ratio of about 8, while for the same file. WinRAR has a compression ratio of 98, and arithmetic coding even reach the value 184.

As can be seen, Splay Tree compression algorithm is exceeded by other compression methods in most cases (except for E.coli). The algorithm with the

biggest medium ratio is WinRAR, followed by Lempel-Ziv and arithmetic coding. This poor result shows that Splay Tree is an inefficient algorithm used by it, although it is a very fast algorithm. It is noted that all algorithms are effective on the same type of files: Excel file kennedy.xls. pic image and ppt5 file.

Table 8

Comparison of different compression algorithms by compression ratio

File	Compression ratio ST	Compression ratio Arithmetic Coding	Compression ratio LZ	Compression ratio WinRAR
bib	1.2772	1.79	2.5952	3.30
pic	4.6791	8.7	4.6915	10.24
progc	1.3512	1.57	2.4375	3.00
cp	1.2993	1.57	2.5	3.13
grammar	1.4878	1.21	2	2.00
kennedy	2.0537	2.65	3.3092	24.54
ptt5	4.6791	8.7	4.6915	10.24
E.coli	3.3072	4	2.9339	3.50
world192	1.3872	1.85	2.6260	4.65
aaa	7.9955	184	8.1666	98.00
alphabet	1.4959	6.96	8.1666	98.00
XML	1.5883	2.42	7.0869	54.33
Medium ratio	1.8135	2.5247	2.9715	7.02

4. **CONCLUSIONS**

Splay Tree algorithm is not optimal, but has some useful properties. Generally. the algorithm is easy to implement and can be executed relatively quickly. Data structures are compact; the code is simple and requires only three arrays, unlike Vitter's algorithm, which requires eleven arrays. As Jones observed in his work and as can be seen from the experimental results, the results of the splay encoding of characters are lower than those of a conventional system of character encoding using an encoding with minimum redundancy on some text files.

On the other hand, for files that contain images, splay encoding gives a superior efficacy of compression because the open surfaces models and dark areas in typical images represent the MTF effects processing on short segments of symbols.

Maximum compression factor cannot exceed 8, and to be able to increase it should read from the file larger symbols (16. 32. 64 bits). Moreover, is obtained a good quality of compression for most types of files if are applied before Burrows-Wheeler transform or Move-to-Front with Burrows-Wheeler.

The algorithm is very fast regardless of existing files, with compression times higher than other compression algorithms. Although its implementation is so fast, the

splay encoder does not require much memory space. Storage requirements for this algorithm are around 9.7 kb (or 2 kB). Static compression methods also provide fast operations at the expense of compression efficiency.

The original contribution of the paper consists in proving the advantage of this new version of splay encoding: the proposed algorithm applies an improved balanced splay-tree that is operating the frequency variations of the inhomogeneous messages.

Received on March 15, 2011

REFERENCES

1. R.E. Tarjan, D.D. Sleator, *Self-Adjusting Binary Search Trees*, J. ACM, **52**, *3*, pp. 652-686, 1985.
2. B. Allen, I. Munro, *Self-Organizing Search Trees*, J. ACM, **25**, *4*, pp. 526-535, 1978.
3. D. Salomon, *Data Compression – The Complete Reference*, 3rd Edition, Springer, 2003.
4. M. Burrows, D.J. Wheeler, *A Block-Sorting Lossless Data Compression Algorithm*, 1994, Report available at: http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-124.html.
5. J.L. Bentley, D.D. Sleator, R.E. Tarjan, V.K. Wei, *A Locally Adaptive Data Compression Scheme*, ACM 29, April 4, 1986, pp. 320-330.
6. M. Nelson, *Data Compression with the BWT*, Dr. Dobb's Journal, Sept. 1996.
7. J.G. Cleary, I.H. Witten, *A Comparison Of Enumerative and Adaptive Codes*, IEEE Transactions on Information Theory, **IT-30**, *2*, pp. 306-315, 1984.
8. M. Nelson, *The Data Compression Book*, 2nd Edition, M&T Books, 1995.
9. A. Moffat, A. Turpin, *Compression and Coding Algorithms*, Kluwer Academic, 2002.
10. T.C. Bell, J.G. Cleary, I.H. Witten, *Text Compression*, Prentice Hall, Englewood Cliffs, NJ, 1990.
11. G. Gallager, *Variations On A Theme By Huffman*, IEEE Transactions on Information Theory, **IT-24**, *6*, pp. 668-674, 1978.
12. D.E. Knuth, *Dynamic Huffman Coding*, Journal of Algorithms, **6**, *2*, pp. 163-180, 1985.
13. J.S. Vitter, *Algorithm 673: Dynamic Huffman Coding*, ACM Transactions on Mathematical Software, **15**, *2*, pp. 158-167, 1989.
14. R. Rădescu, *Lossless Compression – Methods and Applications*, Matrix Rom, Bucharest, 2003.
15. R. Rădescu, G. Liculescu, *Efficient Implementation of Adaptive Huffman Methods in Lossless Compression*, Proceedings of the 5th International Workshop on Optimal Codes and Related Topics, Balchik, Bulgaria, 16-22 June 2007, pp. 209-215.
16. Calgary Corpus: ftp://ftp.cpsc.ucalgary.ca/pub/projects/text.compression.corpus/.
17. Canterbury Corpus: http://corpus.canterbury.ac.nz/.
18. R. Rădescu, *Text Compression Using Predictive Methods and Transforms*, Matrix Rom, Bucharest, 2012 (to be published).
19. D.W. Jones, *Application of Splay Trees to Data Compression*, Communications of the ACM, **31**, *8*, pp. 996-1007, 1988.
20. R. Rădescu, I. Bălăşan, *Recent Results in Lossless Text Compression Using the Burrows-Wheeler Transform*, Proceedings of IEEE International Conference on Communications 2004, Bucharest, Romania, 3–5 June 2004, pp. 105-110.