A Memory-Efficient Adaptive Huffman Coding Algorithm for Very Large Sets of Symbols Revisited

Steven Pigeon
Yoshua Bengio
{pigeon,bengioy}@iro.umontreal.ca

Département d'Informatique et de Recherche opérationelle (DIRO)
Université de Montréal

November 1997

Note

The algorithm described in this paper as “algorithm M^+” is hereby released to the public domain by its authors. Any one can use this algorithm free of royalties provided that any product, commercial or not, includes a notice mentioning the use of this algorithm, the name of the algorithm and the name of the authors.

Abstract

While algorithm M (presented in A Memory-Efficient Huffman Adaptive Coding Algorithm for Very Large Sets of Symbols, by Steven Pigeon & Yoshua Bengio, Université de Montréal technical report #1081 [1]) converges to the entropy of the signal, it also assumes that the characteristics of the signal are stationary, that is, that they do not change over time and that successive adjustments, ever decreasing in their magnitude, will lead to a reasonable approximation of the entropy. While this is true for some data, it is clearly not true for some other. We present here a modification of the M algorithm that allows negative updates. Negative updates are used to maintain a window over the source. Symbols enter the window at its right and will leave it at its left, after *w* steps (the window width). The algorithm presented here allows us to update correctly the weights of the symbols in the symbol tree. Here, we will also have negative migration or *demotion*, while we only had positive migration or *promotion* in M. This algorithm will be called M^+.

1. Introduction

The reader will want to read the technical report #1081, A Memory-Efficient Huffman Adaptive Coding Algorithm for Very Large Sets of Symbols, by Steven Pigeon & Yoshua Bengio, for the introductory material. In this former technical report, we present all the algorithms of M in great detail and some results. Other relevant material is presented in [2,3,4,5,6,7,8,9,10,11,12,13]. Here, however, we will assume that the reader is already familiar with algorithm M, and all of its sub-algorithms, and will therefore only present the changes from M to M^+.

The modification we now present was clearly needed to take into account the non-stationarities of the source. While some data do not exhibit a great difference between order-0 and higher-order entropy, some do. One way to take into account this non-stationarity, is to cumulate the statistics over a moving window of fixed-size. It will turn out that sometimes we will need to demote symbols observed to far in the past — i.e., remove them from the memory of the model. We will do so when they leave the window. Algorithm M only took care of *promotions*, that is, a symbol could only get a higher priority (thus a shorter code) when it was seen once more; *demotion* was virtual in the sense that a set of symbols was demoted only when it was outranked by another (thus getting a longer code). In algorithm M^+, we will need to take explicit actions to ensure that a symbol is also demoted when it leaves the window. We will also note that algorithm M^+ does not include other automatic demotion mechanisms such as *aging*, for example.

2. Algorithm M^+.

Let us now present this new algorithm, algorithm M^+. The difference between the promotion, or *positive update algorithm* and the demotion, or *negative update algorithm* is minimal. Both operate the migration of a symbol from a set to another. The difference is that we move a symbol from a set of frequency n to a set of frequency n-1 rather than from n to n+1. If the set of frequency n-1 exists, the rest of the update is as in algorithm M: we migrate the symbol from its original set to the set of frequency n-1 and we shift up the involved sets (the source's sibling and the destination). If the destination does not exist, we create it and migrate the symbol to this new set. Algorithm 1 shows the negative update procedure.

The complete algorithm that uses both positive and negative updates is called M^+. We will need a windowed buffer, of a priori fixed length[1], over the source. For compression, the algorithm proceeds as follows: As a symbol **a** enters the window, we do a positive update (or **Update_M(a)**, see technical report #1081) and emit the code for **a**. When a symbol **b** leaves the window, we do a negative update (**Negative_Update_M(b)**). On the decompression side, we will do exactly the same. As we decode a symbol **a**, we do a positive update and insert it into the window. This way, the decoder's window is synchronized with the encoder's window. When a symbol **b** leaves the window, we do a negative update (**Negative_Update_M(b)**). Algorithm 2 shows the main loop of the compression program, while algorithm 3 shows the decompression program and algorithm 1 shows the negative update procedure.

```
Procedure Negative_Update_M(a)
 {
  q,p :pointers to leaves ;

  p = find(a) ;  // Leaf/set containing a
  q = find(p's frequency - 1) ; // were to migrate ?

  if (q != ⊥ ) // somewhere to migrate ?
   {
    remove a from p's set ;
    p's weight = p's weight - p's frequency
    Add a to q's set ;
    q's weight = q's weight + p's frequency

    ShiftUp(q);
    If (p = ∅)
         remove p from the tree ;
    else ShiftUp(p's sibling) ;
   }
  else
   {
    create a new node t ;
    t's left child is p ;
    t's right child is a new node n ;
    n's set = {a} ;
    n's weight = p's frequency - 1 ;
    n's frequency = p's frequency - 1 ;
    replace the old p in the tree by t ;

    remove a from p's set ; // t's left child
    p's weight = p's weight - p's frequency ;

    If (p = ∅)
         remove p from the tree ;
    else ShiftUp(p's sibling) ;

    ShiftUp(t) ;
   }
 }
```

Algorithm 1. Negative update algorithm.

[1] We do not address the problem of finding the optimal window length in algorithm M^+.

```
while (not eof(f))
 {
  read c from f;
  put c in window/buffer ;

  w = code for c ;

  Emit w into f2 ; // output file

  Update_M(c) ;

  get d from the other end of window/buffer ;
  Negative_Update_M(d) ;
 }
```

Algorithm 2. Encoding loop.

```
for i=1 to w step 1
 {
  c= Decode from f;
  Update_M(c) ;

  put c in window/buffer ;

  output c into destination file ;

  get d from the other end of window/buffer ;
  Negative_update_M(d) ;
}
```

Algorithm 3. Decoding loop.

3. Results and Complexity Analysis.

We present here the new results. As in technical report #1081, we used the files of the Calgary Corpus as test set. The tried window sizes were 8, 16, 32, 64, 128, 256, 512, and 1024. The results are presented in table 1. For some files, a window size smaller than file size gave a better result, for some other, only a window as large as the file itself gave the best results. Intuitively, one can guess that files that are compressed better with a small window do have non-stationarities and those who do not compress better either are stationary or do have non-stationarities but that they are not captured: they go unnoticed or unexploited because either way the algorithm can't adapt fast enough to take them into account.

Algorithm M^+ is about twice as slow as algorithm M. While it is still $O(\lg n_s)$ per symbol, we have the supplementary operation of demotion, which is $O(S(s_1,s_2) + \lg n_s)$, where $S(s_1,s_2)$ is the complexity of moving a symbol from set s_1 to set s_2, and n_s is the number of sets. Since migration processes are symmetric (promotion and demotion are identical : one migration and two shift up) algorithm M^+ is exactly twice as costly as algorithm M.

Since algorithm M^+ is $O(S(s_1,s_2) + \lg n_s)$, we will remind the reader that she will want the complexity of $S(s_1,s_2)$ to be as low as possible, possibly in $O(\max(\lg |s_1|, \lg |s_2|))$, which is reasonably fast[2].

		M^+ window size									
File	Size	8	16	32	64	128	256	512	1024	M	Huff*
BIB	111261	5.87	5.50	5.49	5.42	5.37	5.35	5.31	5.29	5.33	5.30
BOOK1	768771	4.86	4.73	4.66	4.66	4.64	4.63	4.63	4.62	4.61	4.57
BOOK2	610856	5.01	4.90	4.91	4.90	4.90	4.88	4.87	4.93	4.91	4.83
GEO	102400	7.55	7.60	7.52	7.07	6.83	6.71	6.24	6.14	5.82	5.75
NEWS	377109	5.38	5.33	5.40	5.31	5.30	5.31	5.33	5.30	5.31	5.25
OBJ1	21504	7.92	7.39	7.02	6.92	6.70	6.58	6.51	6.46	6.19	6.35
OBJ2	246814	7.79	7.09	6.71	6.52	6.47	6.48	6.48	6.43	6.40	6.32
PAPER1	53161	5.61	5.41	5.29	5.19	5.16	5.13	5.15	5.21	5.12	5.17
PAPER2	82199	5.11	4.86	4.77	4.76	4.71	4.73	4.83	4.74	4.73	4.73
PAPER3	46526	5.18	4.93	4.87	4.81	4.81	4.82	4.79	4.79	4.86	4.87
PAPER4	13286	5.54	5.18	5.15	5.03	4.93	5.00	4.97	4.96	4.98	5.35
PAPER5	11954	5.93	5.47	5.35	5.26	5.24	5.25	5.20	5.25	5.20	5.65
PAPER6	38105	5.81	5.38	5.42	5.25	5.30	5.22	5.26	5.19	5.15	5.25
PIC	513216	1.78	1.72	1.71	1.70	1.70	1.68	1.68	1.68	1.68	1.68
PROGC	39611	5.87	5.69	5.63	5.45	5.41	5.47	5.37	5.40	5.38	5.44
PROGL	71646	5.14	5.12	4.95	4.92	4.95	4.98	4.99	4.89	4.92	4.91
PROGP	49379	5.61	5.28	5.12	5.13	5.03	5.02	5.03	5.02	5.00	5.07
TRANS	93695	6.29	5.86	5.75	5.75	5.69	5.71	5.70	5.64	5.69	5.66
	μ	5.68	5.41	5.32	5.23	5.17	5.16	5.13	5.11	5.07	5.12

Table 1. Results of Algorithm M+ against Huffman.

4. Conclusion

We see that the enhanced adaptivity of algorithm M^+ can give better result than simple convergence under the hypothesis of stationarity. The complexity of the algorithm remains the same, that is, $O(S(s_1,s_2) + \lg n_s) = O(\max(S(s_1,s_2),\lg n_s))$ while the so-called 'hidden' constant only grows by a factor of 2. In our experiments, M^+ can encode or decode about 10000 to 50000 symbols per seconds, depending on many factors such as compiler, processor and operating system. The reader may keep in mind that this prototype program wasn't written with any hacker-style optimizations. We kept the source as clean as possible for the eventual reader. We feel that with profiling and fine-tuning the program could run at least twice as fast, but that objective goes far off a simple feasibility test.

[2] The current implementation of our sets is in worst case $O(|s|)$, but is most of the time in $O(\lg |s|)$.

5. Acknowledgements

We would like to thank Léon Bottou, of AT&T Research, Newmann Springs Lab, N.J., USA, for our (often animated) discussions on diverse windowed adaptation strategies.

References

[1] Steven Pigeon, Yoshua Bengio — A Memory-Efficient Huffman Adaptive Coding Algorithm for Very Large Sets of Symbols — Université de Montréal, Rapport technique # 1081

[2] D.A. Huffman — A method for the construction of minimum redundancy codes — in *Proceedings of the I.R.E.*, v40 (1951) p 1098-1101

[3] R.G. Gallager — Variation on a theme by Huffman — *IEEE. Trans. on Information Theory*, IT-24, v6 (nov 1978) p 668-674

[4] N. Faller — An Adaptive System for Data Compression — Records of the 7th Asilomar conference on Circuits, Systems & Computers, 1973, p. 393-397

[5] D.E. Knuth — Dynamic Huffman Coding — Journal of Algorithms, v6, 1983 p. 163-180

[6] J.S. Vitter — Desing and analysis of Dynamic Huffman Codes — Journal of the ACM, v34 #4 (oct. 1987) p. 823-843

[7] T.C. Bell, J.G. Cleary, I.H. Witten — Text Compression — Prentice Hall 1990 (QA76.9 T48 B45)

[8] G. Seroussi, M.J. Weinberger — On Adaptive Strategies for an Extended Family of Golomb-type Codes — in DCC 97, Snowbird, IEEE Computer Society Press.

[9] A. Moffat, J. Katajainen — In place Calculation of Minimum Redundancy Codes — in Proceeding of the Workshop on Algorithms and Data Structures, Kingston University, 1995, LNCS 995 Springer Verlag.

[10] Alistair Moffat, Andrew Turpin — On the Implementation of Minimum-Redundancy Prefix Codes — (extended abstract) in DCC 96, p. 171-179

[11] J.S. Vitter — Dynamic Huffman Coding — Manuscript available on the author's web page, with the source in PASCAL

[12] Steven Pigeon — A Fast Image Compression Method Based on the Fast Hartley Transform — AT&T Research, Speech and Image Processing Lab 6 Technical Report (number ???)

[13] Douglas W. Jones — Application of Splay Trees to Data Compression — CACM, v31 #8 (August 1988) p. 998-1007