Windowed Huffman coding algorithm with size adaptation

H.-C. Huang
J.-L. Wu

Indexing terms: Data compression, Entropy coding, Huffman codes

Abstract: The windowed Huffman algorithm is introduced. The Huffman code tree is constructed based on the probabilities of symbols' occurrences within finite history in this windowed algorithm. A window buffer is used to store the most recently processed symbols. Experimental results show that by choosing a suitable window size, the length of codes generated by the windowed Huffman algorithm is shorter than those generated by the static Huffman algorithm, dynamic algorithms, and the residual Huffman algorithm, and even smaller than the first-order entropy. Furthermore, three policies to adjust window size dynamically are also discussed. The windowed Huffman algorithm with an adaptive-size window performs as well as, or better than, that with an optimal fixed-size window. The new algorithm is well suited for online encoding and decoding of data with varying probability distributions.

1 Introduction

Variable length code, such as the well known Huffman code constructed by Huffman [1], comprises an essential part of most compression algorithms. Since the symbols in the source data do not usually occur equally, the variable length code assigns shorter codewords to more probable symbols and assigns longer codewords to less probable symbols to achieve a shorter average code length. The static Huffman code, which was proposed by Huffman, is a two-pass algorithm. The disadvantages of the static Huffman algorithm are the time overhead of two-pass scans over the source data and the code overhead of code tree transmission. Fallar [2], Gallager [3] and Knuth [4] proposed a one-pass algorithm, called the dynamic Huffman code or algorithm FGK, which uses the probabilities of symbols in encoded data to construct a Huffman tree. Vitter [5] proposed a different dynamic Huffman coding algorithm, called the algorithm V, which constructs the dynamic Huffman tree by minimising the maximal code length to optimise the code efficiency in the worst case. Dynamic Huffman coding algorithms contain an update procedure, which adjusts the Huffman tree when the weight of a node in the code tree is increased. The advantages of dynamic Huffman coding are that it scans source data only once, and does not need to send the code tree. Another coding scheme, called residual Huffman coding, was proposed in Reference 6. Residual Huffman coding uses the probabilities of symbols in source data, which are not yet encoded, to construct the Huffman tree. The residual Huffman code also has the overheads for scanning the source two times and sending the coding tree. A new dynamic Huffman algorithm is introduced in this paper, and is called the windowed Huffman algorithm. The new algorithm has the same advantages as dynamic Huffman coding, and is the most efficient code of all the algorithms described above.

The original idea of the windowed Huffman algorithm was proposed in References 2 and 5, and a similar coding scheme, applied to arithmetic coding, was proposed in Reference 7. The code tree of the windowed Huffman algorithm is constructed by adjusting the tree when the weight of a node is increasing or decreasing. The weight of a leaf node in the windowed Huffman tree is the number of occurrences of the associated symbol in the limited history. In the proposed algorithm, a window with size n is used to store the most recently n encoded symbols. When encoding a symbol, the symbol is added into the window, then the weight of the corresponding node is increased, and the update_increase procedure, which is similar to the update procedures of algorithm V [5] and algorithm FGK [4], is performed to adjust the windowed Huffman tree. When the window is full, the oldest symbol in the window is removed, the weight of the node is decreased, and the update_decrease procedure, which is similar to the update procedure of the residual Huffman algorithm [6], is performed to adjust the tree. If the size of the window is selected properly, the coding efficiency of the windowed Huffman algorithm is the best among all the algorithms described above.

As shown by the experimental results in Section 5, the coding efficiency of the windowed Huffman algorithm varies when the window size is changed. The optimal window size is not always the same for different source data, and even for different parts in source data. Thus, by adjusting the window size dynamically during the encoding process, better coding efficiency is obtained in most cases. In the present paper we propose three policies to adjust the window size. The first one is to adjust the window size using the information obtained from the single window Huffman tree, and is the simplest
approach. The second one is to adjust the window size using the information obtained from multiple Huffman trees with different window sizes. It generates more efficient codes than the first one, but needs a larger memory space to construct the code trees and more time to encode the data. The third one sends the encoded codes after the actual window size is computed. It has the best performance, but needs extra information for decoding and the encoded data can not be transmitted until a block of data is processed.

2 Algorithm FGK, algorithm V and the residual Huffman algorithm

In this Section, we present the basic features of the algorithm FGK [2-4], the algorithm V [5] and the residual Huffman algorithm [6], which are important in real implementation.

2.1 Algorithm FGK

The basis of algorithm FGK is the sibling property, defined by Gallager [3]:

Sibling Property: A binary tree with p leaves of non-negative weight is a Huffman tree if and only if:

(a) the p leaves have nonnegative weights, and the weight of each internal node is the sum of the weight of its children, and

(b) the nodes can be numbered in nonincreasing order by weight, so that nodes $2^j - 1$ and $2^j$ are siblings, for $1 \leq j \leq p - 1$, and their common parent node is higher in the numbering.

In algorithm FGK, both sender and receiver maintain dynamically changed Huffman code trees. The leaves of the code tree represent the symbols of the alphabet, and the weights of the leaves represent the number of occurrences of the symbols in processed data. Initially, the code tree consists of a single leaf node, called the 0-node. The 0-node is a special node used to represent the unused symbols of the alphabet, and the weight of the 0-node is zero. For each symbol encoded and decoded, both sender and receiver need to increase the weight of the node corresponding to the symbol, and call the update procedure to adjust the code tree to maintain the sibling property. The pseudo code of algorithm FGK is given in Section 8.1.

2.2 Algorithm V

The basic concepts and the main procedure of algorithm V are the same as those of algorithm FGK except that the update procedure of algorithm V maintains not only the sibling property, but also the implicit numbering and invariance of the code tree [5], which are outlined below:

Implicit Numbering: The nodes of the code tree are numbered in increasing order by level, nodes on one level are numbered lower than the nodes on the following higher level. Nodes on the same level are numbered in increasing order from left to right.

Invariance: For each weight w, all leaves of weight w precede (in the implicit numbering) all internal nodes of weight w.

2.3 Residual Huffman algorithm

The basic idea of the residual Huffman algorithm is that the weights of nodes of the code tree are the number of occurrences in the source data to be processed rather than in the processed source data. The update procedure of the residual Huffman algorithm adjusts the code tree to maintain the sibling property and the invariant when the weight of a node is decreased.

3 Windowed Huffman algorithm with fixed-size window

In this Section, we introduce the simplest version of the windowed Huffman algorithm, which uses a fixed-size window buffer. The motivation of this algorithm is as follows. In general, the probability distribution of symbols is variant in different segments of the source data, especially in video source. Dynamic Huffman algorithms count the numbers of occurrences of symbols in the whole processed data. Since the probability distribution of symbols is changing during the coding process, a better coding scheme forgets the out of date symbols to get a more accurate probability distribution of the current segment of source data.

The windowed Huffman algorithm uses the window buffer to store recently processed symbols. The size of window is equivalent to the size of the memory of the coder. When the variance of the probability distribution of the source data is large, a coder with a small-size window will get better performance. When the variance is small, a coder with a large-size window performs better.

Initially, the window buffer is empty, and the code tree contains the 0-node only. During the coding process, as a symbol enters the coder, the update_increase procedure is called to increase the weight of the corresponding node and the code tree is adjusted to maintain sibling property and invariance. The symbol is then put into the window buffer and is tested to see whether the window is full. If the window is full, the oldest symbol of the window is removed and the update_decrease procedure is called to decrease the weight of the node associated to the removed symbol, and the code tree is also adjusted. The update_increase procedure is similar to the update procedure of algorithm V, and the update_decrease procedure is similar to the update procedure of the residual Huffman algorithm. The procedure of the windowed Huffman algorithm is listed in Section 8.2.

4 Windowed Huffman algorithm with adaptive-size window

In the previous Section, the windowed Huffman algorithm with a fixed-size window was introduced. Because the performance of the algorithm depends on the size of the window, and the optimal size of the window depends on source data, it is hard to determine a suitable window size for all types of data. Thus, an adaptive version of the windowed Huffman algorithm is proposed in this Section. The adaptive windowed Huffman algorithm changes the window size dynamically during the coding process. The new algorithm is given in Section 8.3.

The adjust_window_size procedure is used to recompute the window size. Ideally, this procedure needs to compute the optimal window size at any time instance of the coding process, but it is hard to compute the optimal window size, even to determine whether the current window size is suitable. We propose three policies to adjust the window size in the following. To gather enough information and make the algorithm faster, the source data is partitioned into several blocks. The window size is recomputed only when the algorithm is going to process the first symbol of a block, and the
information gathered in the current block is used for computing the next window size. Therefore, the block size can also be adaptive.

**Policy I:** Adjust window size with single window and code tree: Under this policy, our algorithm contains a window buffer and a code tree. The total communication cost of the jth block is costj, and directionj indicates the adjusting direction (increase or decrease) of the window size when processing the jth block. The guideline of window adjusting is as follows: If the communication cost is increased when increasing (or decreasing) the window size, we assume the window size may be too large (or too small), and thus the window size is decreased (or increased).

Both sender and receiver use the same policy to adjust their window sizes, so no extra information has to be transmitted. The above assumption is based on the first-order Markov model and can be extended to nth order cases by using information gathered from n recently processed blocks. In our experiments, the fifth-order Markov model is chosen.

**Policy II:** Adjust window size with multiple windows and code trees: We can use multiple code trees to process source data, and different code trees in the coder are constructed using different sizes of windows. During the coding process, only the codes generated by the code tree with the selected window size are sent. When adjusting the window size, the window size corresponding to the shortest codes among all code trees is chosen to process the next block. Define \( \Delta s \) as the difference between the window size used in the previous block and that which is to be used in the following block, and all window buffers in both sender and receiver adjust their size by \( \Delta s \). As in policy I, no extra information is required in this approach. The coder using this policy performs better than that using policy I in most cases, but it needs more memory space to construct multiple code trees and window buffers.

**Policy III:** Adjust window size with one block delay: Multiple code trees are used in this policy, too. The codes generated are buffered in code buffers associated with code trees rather than sent immediately. After a block is processed, the code tree with the shortest codes is selected. The code tree number is sent to the receiver before the buffered codes. The receiver uses the received tree number to adjust its window size, and decode the received data. When using this policy, extra information for indicating code tree number is needed, and the encoded codes will not be sent until the entire block is processed. Although the policy requires side information, its performance is the best among the three.

A coder with policy I is the simplest, smallest and fastest coder of all, but its performance is not good enough. A coder with policy II performs better than that with policy I, but it needs more space to construct and store the multiple code trees. A coder with policy III performs the best, but it sends extra information and delays by one block unit.

5 Experimental results

In Fig. 1a and 1b, the curves respectively show the compression ratios obtained by the windowed Huffman algorithm using different window sizes to encode a binary file and source data containing several parts of different probability distributions of symbols, called hybrid data. The first-order entropy and the lengths of codes generated by the static algorithm, algorithm FGK, and the adaptive windowed Huffman algorithms based on policies I, II, and III are also shown in these figures. The block size used in the windowed Huffman algorithm is 256 bytes, the adaptive windowed Huffman algorithm based on policy I uses a fifth-order Markov model and those based on policies II and III use nine windows with different sizes, and the initial window size of the adaptive windowed Huffman algorithms is 6400 bytes.

In the second experiment, several types of source data such as hybrid data, TIFF file, NTSC signal, C source code, document, binary executable file and dumped core data are tested. We use E, S, FGK, V and R to denote the first-order entropy and lengths of codes generated by the static algorithm, algorithm FGK, algorithm V and residual Huffman algorithm. \( W(n) \) denotes the code length of the windowed Huffman algorithm using a window with precomputed optimal size \( 256 \times n \) bytes, and AW-I, AW-II and AW-III denote the code lengths of the adaptive windowed Huffman algorithms based on policies I, II and III. As the experimental results shown in Table 1, the windowed Huffman algorithms perform better than other algorithms including the first-order entropy, and the algorithm based on policy III is the best of them. Fig. 2 shows the same results.
Table 1: Experimental results of several algorithms performing on different kinds of data

<table>
<thead>
<tr>
<th>File Source</th>
<th>E</th>
<th>S</th>
<th>FGK</th>
<th>V</th>
<th>R</th>
<th>W(n)</th>
<th>AW-I</th>
<th>AW-II</th>
<th>AW-III</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hybrid</td>
<td>2081</td>
<td>1450</td>
<td>1456</td>
<td>1460</td>
<td>1459</td>
<td>1455</td>
<td>1192</td>
<td>1221</td>
<td>1198</td>
</tr>
<tr>
<td>Tiff</td>
<td>449</td>
<td>936</td>
<td>367</td>
<td>369</td>
<td>372</td>
<td>374</td>
<td>377</td>
<td>381</td>
<td>382</td>
</tr>
<tr>
<td>NTSC</td>
<td>350</td>
<td>764</td>
<td>348</td>
<td>107</td>
<td>346</td>
<td>926</td>
<td>559</td>
<td>442</td>
<td>445</td>
</tr>
<tr>
<td>C++</td>
<td>330</td>
<td>266</td>
<td>268</td>
<td>270</td>
<td>271</td>
<td>274</td>
<td>276</td>
<td>281</td>
<td>284</td>
</tr>
<tr>
<td>Doc</td>
<td>163</td>
<td>142</td>
<td>145</td>
<td>149</td>
<td>146</td>
<td>145</td>
<td>141</td>
<td>143</td>
<td>141</td>
</tr>
<tr>
<td>Bin</td>
<td>80</td>
<td>78</td>
<td>76</td>
<td>74</td>
<td>72</td>
<td>70</td>
<td>80</td>
<td>80</td>
<td>80</td>
</tr>
<tr>
<td>Core</td>
<td>2081</td>
<td>1450</td>
<td>1456</td>
<td>1460</td>
<td>1459</td>
<td>1455</td>
<td>1192</td>
<td>1221</td>
<td>1198</td>
</tr>
</tbody>
</table>

Fig. 2 Comparison of various algorithms for various source data

6 Conclusions and open problems

The proposed windowed Huffman algorithm uses a window buffer to store the most recently processed data and constructs a dynamic Huffman tree based on the probability distribution of symbols in the window. By choosing a suitable window size, the length of codes generated by the windowed Huffman algorithm is shorter than those generated by the static Huffman algorithm, dynamic algorithms and the residual Huffman algorithm, and is even smaller than the first-order entropy. To obtain better performance when encoding different source data, the window size must be adjusted dynamically. Three policies for adjusting the window size were proposed in this paper. Among the three policies, the windowed Huffman algorithm with adaptive-sized window performed as well as, or better than, that with an optimal fixed-size window. The new algorithm is well suited for online encoding and decoding of data with varying probability distributions.

As discussed in Section 4, it is hard to compute the optimal window size during the encoding process. To find an algorithm which can accurately and efficiently compute the optimal window size requires further investigation.

7 References


Appendixes

8.1 Algorithm FGK

procedure update(k: symbol);
begin
  q  := node corresponding to symbol k;
  if (q is the 0-node) then begin
    Split q and increase the weight of q by 1;
    q := father of q;
  end;
  while (q is not equal to root) do begin
    Interchange q with the highest numbered leaf of the same weight;
    Increase the weight of q by 1;
    q := father of q;
  end;
end (*main program*)

Initial the code tree with 0-node;
while (not end of source data) do begin
8.2 Windowed Huffman algorithm
program windowed_Huffman_algorithm;
begin (*main program*)
Initial code tree with a 0-node, empty the window buffer;
while (not end of source data) do begin
Encode [or decode] the next symbol new_symbol;
Update_increase (new_symbol);
Put new_symbol into the window buffer;
if (the window is full) then begin
Remove the oldest_symbol from window buffer;
Update_decrease (oldest_symbol);
end;
end;
end.

8.3 Adaptive windowed Huffman algorithm
program adaptive_windowed_Huffman_algorithm;
begin (*main program*)
Initial code tree with a 0-node, empty the window buffer;
while (not end of source data) do begin
Encode [or decode] the next symbol new_symbol;
Update_increase (new_symbol);
Put new_symbol into the window buffer;
if (the window is full) then begin
Remove the oldest_symbol from window buffer;
Update_decrease (oldest_symbol);
end;
Adjust_window_size;
end;
end.