Data encoding and compression with help of static and adaptive Huffman coding

Michal Ormoš
Data Coding and Compression 2019/2020

April 24, 2020

1 Intro

Aim of this project is implementing a program for compression of picture data representation with the use of static and adaptive Huffman coding.

2 Proposal

In this section we will introduce and discuss principles of encoding and compression, we will divide it to two subsections, the first will be about Static Huffman coding 2.1 and the second will be about Adaptive Huffman coding 2.2.

2.1 Static Huffman coding

The static algorithm is a double transition algorithm, which means he needs to run over the input stream twice. In the first run, he counts the frequencies of individual symbols. In the second run, he creates the Huffman tree. Every symbol represents the leaf symbol of the tree. In the beginning, two characters are chosen with the lowest frequency and the parent node is added. Parent frequency is the sum of his node frequencies. This new node is saved to set of nodes and the process is repeated. The two new nodes with minimal frequencies are chosen and the same approach is used until only two last nodes are in the set. In the created tree, the path from the root to the leaf node of the symbol contains the Huffman code for that symbol.

2.2 Adaptive Huffman coding

For adaptive Huffman coding, the algorithm Faller-Gallager-Knuth (FGK) was chosen. The algorithm makes use of the sibling's property, which means that every node has a sibling. And nodes are possible to order to monotone succession, by a multiplicity of that node. In a way that every node has a succession after the neighborhood node of his siblings. After the start, he works with the tree that contains only node NYT. After the reading of the new symbol at the place of NYT, the new node is created, which will have the left son of NYT, the right new node representing the desired character. After the reading of the character which is already included in the tree, the tree is actualized in the position of the frequency of that symbol. In case of need, the tree is ordered, so the sibling property is persisted.

3 Implementation

In this section I will introduce my implementation of the Static Huffman conding 2.1 and Adaptive Huffman coding 2.2.

The project was implemented in the language C++, the main program file is the `main.cpp` which reads the arguments, file, operates the model and control the programm flow. The files `helper.cpp/h` containg tree implementation and other supporting functions. Files `static.cpp/h` contains implementation of the static Huffman coding. And files `adaptive.cpp/h` contains implementation of Adaptive huffman coding.

3.1 Static Huffman coding

Implementation of this part is enclosed in the class `Static` in the files `static.cpp/h`. For storing the weight of the symbol the data type `size_t` is used. In the `build` method for every non zero characters, then the node in the tree is created. The algorithm is looking for two nodes with the minimum weight, which he swapped for the new node which has set the weight for the sum of the weight of these two original nodes. The initial nodes are set as son's of this new node. If the only one node is found the algorithm cannot start. This is solved by the detection of one node and an extra node to the tree is then added. This will not add any necessary length to the result.

After the build method finishes, in the method `saveCode` the codded symbols are saved for the character in the tree. The transition of coded symbol to the canonical form is the next step, in the method `makeTreeCanonical`.

Method `encodeBytes` is creating the header and compress bytes of the input file.

- 1st byte carry a number of different length of coded symbols.
- 2nd byte carries the information on how many bits fill the last byte of a file and the possibility of reduction of the header size.
- The following is the sequence of byte where carry the information about how many characters are encoded for the length of the encoded symbol.
- The last part of the header are the symbols of the output alphabet.

After the header is created, the encoding of the bytes of the input file and write off these encoded words for this byte to the output file. In the necessity of filling the space for the bytes, the zeros are written.

In the case of decoding, first, the header is read. Then the decoding is done by the use of algorithms first come first server. The implementation of this algorithm and approach is in the method `decodeBytes`.

3.2 Adaptive Huffman coding

The implementation of the Adaptive Huffman coding is enclosed in the the class `Adaptive` in the files `adaptive.cpp/h`. The class is implementing the already mention algorithm Faller-Gallager-Knuth (FGK).

In the tree initialization, the tree with single node NYT is created. In the method `encodeFile`, every byte of the input stream is encoded with the help of the binary tree and the following actualization of the tree takes place. In the `decodeFile` the process is reversed. Input bits are decoded for the symbol with the help of the binary tree. After every decoded character the actualization of the tree takes place again.

The problem of the last byte filing is the same as in the case of Static Huffman Coding. But here if the last byte filling is zero and the value of the last byte is more than seven, the saving of one byte for information of filing can be saved. As we know the maximum number of filing is seven. If we find out that the value of the last byte is more than seven, we knew that the last byte does not carry the information about the filling, and the actual number of filing is zero.

The actualization of the tree is implemented in the method `editTree` and represents the update tree method the same as in the FGK algorithm. As I found out this method takes the most time, I concern about my optimization efforts here.

3.3 Model

The model implementation has two phases which are implemented in the `main.cpp` file.

The first function `applyModel` will transform every byte of the input stream to the new value of that byte, which is counted as difference value actual byte and value of the previous byte.

The opposite function `reverseModel` will do the opposite process, the right value is taken from the addition of the previous byte to the value of the actual byte.

4 Evaluation

Table 1 carry the desired experiment information's. The values were measured on the laptop with processor `Intel® Core`^{TM} `i5-8350U CPU @ 1.70GHz  8`. The times of the run were taken with the help of program `perf`.

File	**HS time**	**HS size**	**HSm time**	**HSm size**	**HA time**	**HA size**	**HAm time**	**HAm size**
hd01.raw	0,011493164	3.88003540039	0,000931544	3.88003540039	0,013117742	3.88403320313	0,002504101	3.88403320313
hd02.raw	0,012544028	3.70489501953	0,001128757	3.70489501953	0,021813989	3.70935058594	0,000495063	3.70935058594
hd07.raw	0,012675347	5.61282348633	0,000506873	5.61282348633	0,030431990	5.61672973633	0,002408117	5.61672973633
hd08.raw	0,007237028	4.23666381836	0,001315923	4.23666381836	0,022339443	4.24084472656	0,000911639	4.24084472656
hd09.raw	0,010330646	6.65982055664	0,002013895	6.65982055664	0,028830546	6.66598510742	0,003109382	6.66598510742
hd12.raw	0,019987824	6.20028686523	0,002535436	6.20028686523	0,035888063	6.20599365234	0,003677254	6.20599365234
nk01.raw	0,037741660	6.50857543945	0,001572816	6.50857543945	0,036070320	6.51275634766	0,001021338	6.51275634766
average	0.01600138529	5.257585798	0.001429320571	5.257585798	0.02692744186	5.262241908	0.002018127714	5.262241908

Table 1: HS - Static Huffman, HSm - Static Huffman with the model, HA - Adaptive Huffman, HAm - Adaptive Huffman with the model. Time - the run time of the program for the compression. Size - compression ratio of the files.

Appendices

A How to use this program

```
This is program for Static and Adaptive Huffman Coding
Version 1.0.0

Usage:
  -c
      Application will compress input file
  -d
      Application will decompress input file
  -a
      If enabled, adaptive Huffman coding will be used. Other wise static Huffman coding will b
  -m
      Activate model
  -i <input_file>
      Name of input stream for compression/decompression.
  -o <output_file>
      Name of output stream for compression/decompression.
  -w <width_value>
      Set the width of the image, width >= 1.
  -h
      Show this help windows.

Created by @xormos00 for KKO, BUT FIT.
```

B References

- Úvod do problematiky kódování a komprese dat, Zdeněk VAŠÍČEK, FakultaInformačních Technologí, Vysoké Učení Technické v Brně

  `https://wis.fit.vutbr.cz/FIT/st/cfs.php.cs?file=%2Fcourse%2FKKO-IT%2Flectures%2FKKO-01.pdf&cid=13390`

- Context models, Zdeněk VAŠÍČEK, FakultaInformačních Technologí, Vysoké Učení Technické v Brně

  `https://wis.fit.vutbr.cz/FIT/st/cfs.php.cs?file=%2Fcourse%2FKKO-IT%2Flectures%2FKKO-06.pdf&cid=13390`

- Komprese dat v paměťové hierarchii, Zdeněk VAŠÍČEK, FakultaInformačních Technologí, Vysoké Učení Technické v Brně

  `https://wis.fit.vutbr.cz/FIT/st/cfs.php.cs?file=%2Fcourse%2FKKO-IT%2Flectures%2FKKO-07.pdf&cid=13390`

- Adaptive Huffman coding - FGK

  `http://www.stringology.org/DataCompression/fgk/index_en.html`

- Huffman Coding, Steven Pigeon, Universite de Montreal

  `http://stevenpigeon.com/Publications/publications/HuffmanChapter.pdf`