COP 5536/ Advanced Data Structures Programming Project Report

UFID : 77656111

Last Name : Narikimilli

FirstName : Naga Satya Karthik

This report is about Huffman Coding Algorithm in order to optimize the data that is transferred from one server to another. This optimization is achieved in 2 parts Encoder & Decoder. Encoder takes a single large file and builds a Huffman tree and code table. Using code table and input file we build the encoded data by traversing the Huffman tree which is stored in encoded.bin. Once this file is built which is compressed in size than the original encode.bin we send this encoded.bin to the remote server and then decode it over there using the code table(we also need to push the code table to the server – one time task). On the remote server we build the Huffman tree from the code table and then parse the encoded binary file along the tree (that is built on the remote server) and decode into the original input file.

Utilities Used :

Node data structures :

TreeNode **implements** Comparable<TreeNode>

Int data, TreeNode left, right

LeafNode **extends** TreeNode

Int value // value of the key that the encoded string is mapped to.

Heap Data Structures:

Heaps: Generic abstract Data structure that is implemented by varying types of heaps. (pairing heaps & Dary Heaps)



Types of Heaps :

Pairing Heaps:



1. Dary Heaps:

This file alone will be serving as binary / 4-way / 4way-cache optimized. (with varying parameters while initializing).



Algorithm: // pseudo code

Encoder :

Input : input\_file

Output: encoded.bin & code\_table.txt

Steps:

1. Builds a frequency map of the input file (counts of each word in the file).
   1. Iterates the input file using a input stream line by line and updates the counter of the key.
2. Construct HuffMan Tree using the frequency Map Built (with the best performing heap)



1. Write the Output encoded.bin & code\_table.txt.

Using DataOutputStream for generating the binary file and BufferedOutputStream for code\_table.txt. (code is placed in encoder .. pretty straightforward to understand).

**Decoder**:

Input : encoded.bin code\_table.txt (these are the output of the encoder)

Output: decoded.txt

Steps:

1. Construct Decode Tree



1. Use the encoded.bin & decodeTree(a.k.a Huffman Tree) and construct the decoded.txt.

Read the encoded.bin using the datainputStream reads into byte arr and converts each byte into binary string with proper 0 and 1’s padded to the end the stringBuilder. At the same time we also iterate the tree based on the characters 0 or 1. If in case we have reached end of the stringBuilder and still we havent’ reached the LeafNode .. then we read again and update the stringBuilder. The whole process is present in **InputBuffer:getNextItem**.

Conclusions:

After performing careful analysis on the best kind of heap which would result in opmitized encoder performance, I came up with 4-way cache optimized as the best way (compared to pairing heap & binary heap). Below table gives a brief overview of the performance measurement over 10M lines of integer values for building the tree using different heaps. Time taken is the average measurement over 10 iterations. (1M sample2/files)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Iteration | 4way Heap | 4Way CahceOptimized | Pairing Heap | Binary Heap |
| 1 | 1.5477 | 1.422 | 1.3272 | 1.6384 |
| 2 | 1.6253 | 1.521 | 1.3488 | 1.6757 |
| 3 | 1.3377 | 1.3483 | 1.4247 | 1.6475 |
| 4 | 1.5136 | 1.5714 | 1.4325 | 1.7171 |
| 5 | 1.3695 | 1.4457 | 1.2846 | 1.7003 |
| 6 | 1.6943 | 1.497 | 1.3802 | 1.7536 |
| 7 | 1.4083 | 1.4066 | 1.5259 | 1.6946 |
| 8 | 1.422 | 1.3079 | 1.5259 | 1.6946 |
| 9 | 1.3355 | 1.4865 | 1.2464 | 1.8468 |
| 10 | 1.5788 | 1.7048 | 1.1182 | 1.7246 |

Another run is performed on 100M lines of random input satisfying the input conditions and the statistics of the varying heaps is as follows :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Iteration | 4way Heap | 4Way Cache Optimized | Pairing Heap | Binary Heap |
| 1 | 1.384 | 1.162 | 0.441 | 2.606 |
| 2 | 0.977 | 0.996 | 0.332 | 1.102 |
| 3 | 0.945 | 1.743 | 0.386 | 1.497 |
| 4 | 1.308 | 1.172 | 0.361 | 1.174 |
| 5 | 1.018 | 1.594 | 0.395 | 1.172 |
| 6 | 1.2 | 1.208 | 0.371 | 1.211 |
| 7 | 0.968 | 1.015 | 0.529 | 1.152 |
| 8 | 1.097 | 1.028 | 0.47 | 1.416 |
| 9 | 1.081 | 1.025 | 0.498 | 1.11 |
| 10 | 1.103 | 0.991 | 0.473 | 1.117 |



Hence I conclude that Pairing Heap optimized is best in constructing the Huffman Tree when compared to Binary / 4ary Heaps. From both the graphs we can safely conclude that Pairing Heap >> 4way CacheOptimized / 4way >> Binary heap. Reason that we are not seeing much optimization between 4-ary and 4-ary cache optimized is because of the programming language. Since I’m using java as my programming language and the memory model in java is stack based and not strictly dependent on the machine (machine independent) as opposed to c/c++ which are low level programming languages which has direct access to system resources (machine dependent optimazations can be performed).