4/5/2017

Advance Data Structures

Programming Project
Huffman Coding

Desai,Prachi P
11921313

The Huffman Coding is done using the encoder and the decoder. A text file is encoded to a binary file and sent to the decoder to decode the binary file back to text file.

The data structures used for this are:

1. Binary Heap
2. Four-way Heap
3. Pairing Heap

- **Data Structures**

The classes of the data structures used in the project is written using the idea of separate compilation. Instead of writing all the code in one file, and compiling that one file, the files are separated as .cpp and .h files. A .cpp file consists of implementation methods while .h file contains class declaration and function prototype.

1. **bheap**: This is binary heap code which is used for inserting node, deleting the min node and for every insert and delete_min, the heap is heapified using functions percolateup and percolatedown.

2. **fheap**: This is fourway heap, the index of the nodes start from 3 to achieve cache optimization. Thus the parent and children relation is modified using formulae, i/4+2 to find the parent of ith node and (4 * (i-2) + (k-1)) as kth child of node i. Rest of the code has functions to insert element and delete the minimum node, and is heapified using functions percolateup and percolatedown

3. **pheap**: The pairing heap is different than the previous two. The insert function inserts the element in the tree but on deleting the minimum node from the tree, the subtrees are combined using combineSiblings. The compareAndLink funcions compares two subtrees and the subtree with greater root node is attached to the other subtree.

hnode.h:

It is defined as a class binary tree node which is attributed with data, frequency value, and some designated pointers as left and right to efficiently parse through the nodes of our generated Huffman Trees.

Encoder:

1. void ***FillFreqMap***- Fills frequency map from sample input file.

2. void ***CreateHnodeVec(std::unordered_map<int,unsigned int> const &freq_map, std::vector<hnode const*> &vec)***-
   Creates Huffman-Tree vector with the help of frequency map.

3. hnode ***const* BuildTreeFheap(std::unordered_map<int,unsigned int> const &freq_map)***-
   Builds Huffman tree by utilizing a 4-way cache optimized heap

4. hnode ***const* BuildTreeBheap(std::unordered_map<int,unsigned int> const &freq_map)***-
   Builds Huffman tree by utilizing a Binary heap

5. hnode ***const* BuildTreePheap(std::unordered_map<int,unsigned int> const &freq_map)***-
   Builds Huffman tree by utilizing a Pairing heap

6. void ***printCodes(hnode const *root, std::string str, std::unordered_map<int,std::string> &code_table_map)***-
   Traverses through Huffman tree to generate key value pairs for generating code table.

7. void ***compress(std::string &value,std::vector<char> &result)-***
   8-bit compression process.

8. void ***binfilecreate(std::unordered_map<int,std::string> &code_table_map, char *filename, char const *out)***-
   Creates final encoded binary by using hash map and code table generated intermediately.

Decoder:

1. struct ***Node {struct Node *right = NULL; struct Node *left = NULL; int value; bool isLeaf = false;};-*** Defining the node of the Huffman tree which is intended to be decoded.

2. Node* ***getRoot()***- Gets the root of the Huffman tree.

3. void ***addLeafat(int value, string binaryString)-***
   Constructs the decode-huffman tree after parsing code_table.txt

4. int ***getAt(string binaryString)-***
   Traverses through decode tree to parse the binary string and returns valid value stored in leaf node.

5. The primary function of decoding the entire **encoded.bin** file is executed by the main of **decoderHuffman.cpp**.

April 5, 2017

PERFORMANCE ANALYSIS RESULTS AND EXPLANATION

The average time required to create Huffman Tree 10 times for all three heaps is measured as:

- Binary heaps: 2.15 s
- Four way heaps: 1.71 s
- Pairing Heaps: 2.16 s

Overall, using Four way heaps, encoder takes 30 seconds on an average, decoding takes 28 seconds on an average.

Explanation:

First, Huffman Tree is generated using all three data structures and the most efficient data structure is used, which turns out to be Four-way heap. The four way is fastest because it has smaller height compared to the other two. Hence, when the input is very large, the height of the tree matters. Cache optimization is more in Four-way because the index of the parents starts from 3, the children will thus be at indices 4, 5, 6, 7. Thus the siblings will be at consecutive indices i.e. same cache line as shaded below.

Prachi Prakash Desai 11921313

April 5, 2017

DECODING ALGORITHM AND COMPLEXITY

1) Open the files encoded.bin and code_table.txt
2) Consider the code table which has the leaf node value and its code separated by a space and construct a Huffman Tree.
   a) Using the code, create nodes in the tree, if 0 then create a new node as left child else if 1 then create a new node as right child.
   b) Once the code length is over, insert the value to that node and make it a leaf node.
   c) Repeat till entire Tree is constructed.
3) Then traverse through Huffman tree using binary string till you reach the leaf and return the leaf node value.
   a) If binary string is at 0, traverse to the left child else if 1, then traverse to the right child of the current node.
   b) If the node is a leaf node, it will have the value stored in it, which will be returned.
4) Write the leaf node value in the file called decoded.txt

The complexity of the code will be height times order of input.

⇨ O(n.h) time

Prachi Prakash Desai 11921313