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)

```
public abstract class Heaps<T> {
    abstract  void offer(T data);
    abstract T poll();
    abstract T peek();
    abstract int size();
    abstract boolean isEmpty();
}
```

Types of Heaps :

Pairing Heaps:

```
PairingHeap<T extends Comparable<T>> extends Heaps<T>

Pairing Heap Internally uses a Node data Structure in order to implement pairing heap
operations.

Uses Inernally a Node: Node<T extends Comparable<T>> implements Comparable<Node>
```

Node Operations:

- `void insert(Node node); // if the child node is null then the incoming node becomes the child node. Else the incoming child is added to the left most child and pointers are changed accordingly.`

```
Pairing Heap Operations: (Heaps Operations & Custom operations which act as
utilities for operations mentioned in heaps)

Node meld(Node list): does a pairwise combine using the best strategy discussed in
lecture (2 pass scheme) and returns the root.

public void offer(T data) : // creates a new node and based on the compare Method
inserts into the pairing heap.
public T poll(): // if root is null throw a NullPointerException other wise return the
root and do a meld operations on the root.child
```

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

```
class DaryHeap<T extends Comparable<T>> extends Heaps<T>
```

Constructors:

```
public DaryHeap(int d)
public DaryHeap(int d, int kbasedindex)
public DaryHeap(int d, Comparator<? super T> c)
public DaryHeap(int d, int kbasedindex, Comparator<? super T> c)
```

Parameters :

- d :: is binary / 4way heaps.. d is the kind of the heap that you want
- kbasedIndex : for cache optimized version we use this as 3 while initializing the Dary heap. Formulae will be computed accordingly (if not given takes kbasedIndex as 0).
- Comparator : used to specify custom Comparater.. by default it will take the min heap.

Custom Operations:

```
private void heapifyUp(int i): iterates to the top of the tree if heap property is not
satisfied between currnode and parent node.
private void heapifyDown(int index): once all the nodes are offered a position in the
tree we run the heapifyDown from n/2 to 0th position in the array Inorder to satisfy
the heap property.
```

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).
    a. 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)

```
Iterates till the heap contains only one element. Picks the top 2 elements with the
minimum frequency and push them back to the heap. The one with left is considered as 0
and one on right is considered as 1 while generating the code_table.txt

while(!heap.isEmpty() && heap.size() != 1){
    left = heap.poll();
    right = heap.poll();
    temp = new TreeNode(left.data + right.data);
    temp.left = left;
    temp.right = right;
    heap.offer(temp);
}
return heap.peek();
```

3. 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

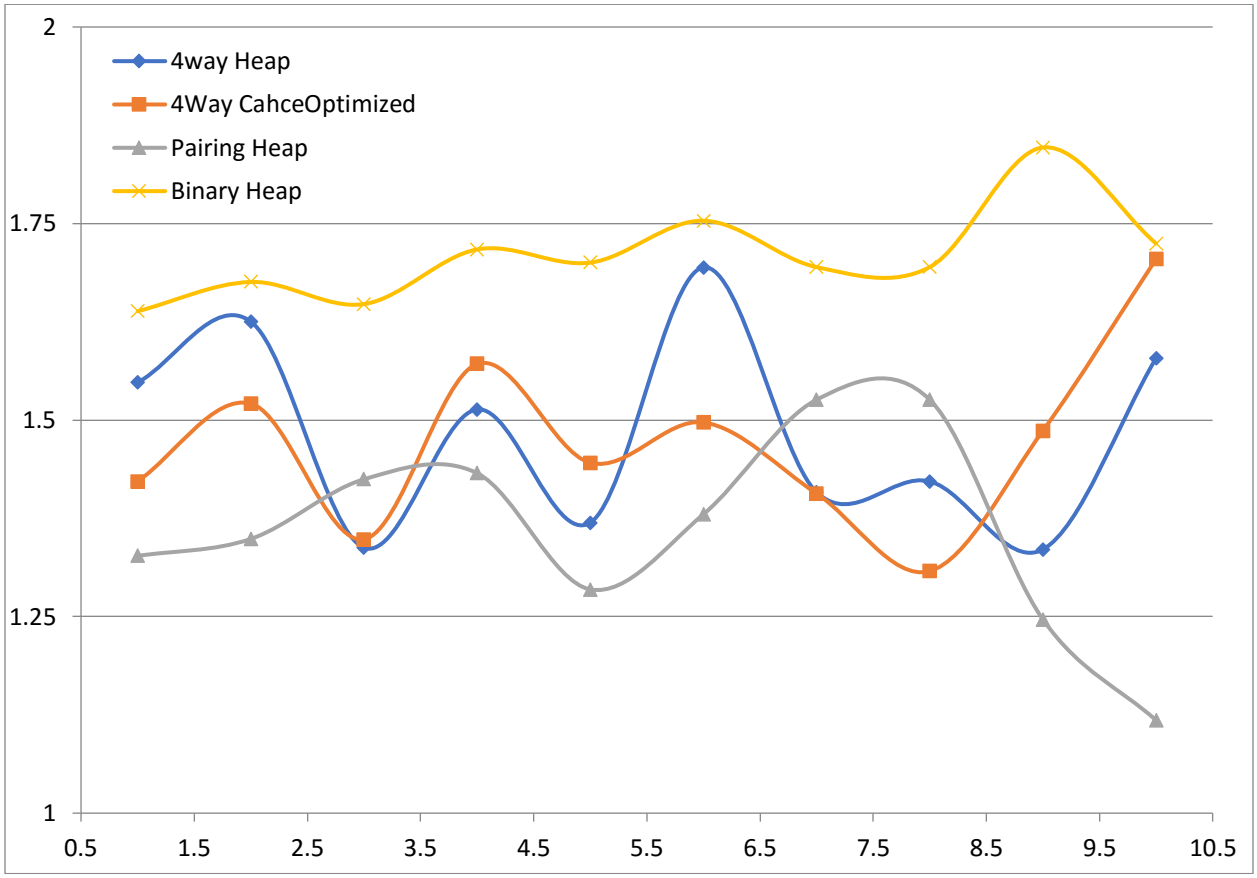
Steps:

1. Construct Decode Tree
   `void constructDecodeTree(TreeNode root, HashMap<String, Integer> encodingMap) // encoding map is the content of code_table.txt`
   Takes in an empty Root and populates its value with the correct value.
   Iterates over the encodingMap.. for each key that is present in the encoding map if the char at the i^{th} position in the key is 1 traverse left of the tree else traverse right of the tree. If at any point of the time if the next node(left / right) is null then create a TreeNode. If you have reached the position at the end of the key create a LeafNode (with the value as value of the encodingmap for this key).
   Performance Analysis : Let h be the height of the tree. (in worst case). For each key that is present in the encoding map we traverse the tree and create nodes if necessary accordingly. So for n items in the encoding map the algorithmic complexity of the tree would be O(nh).
2. 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:

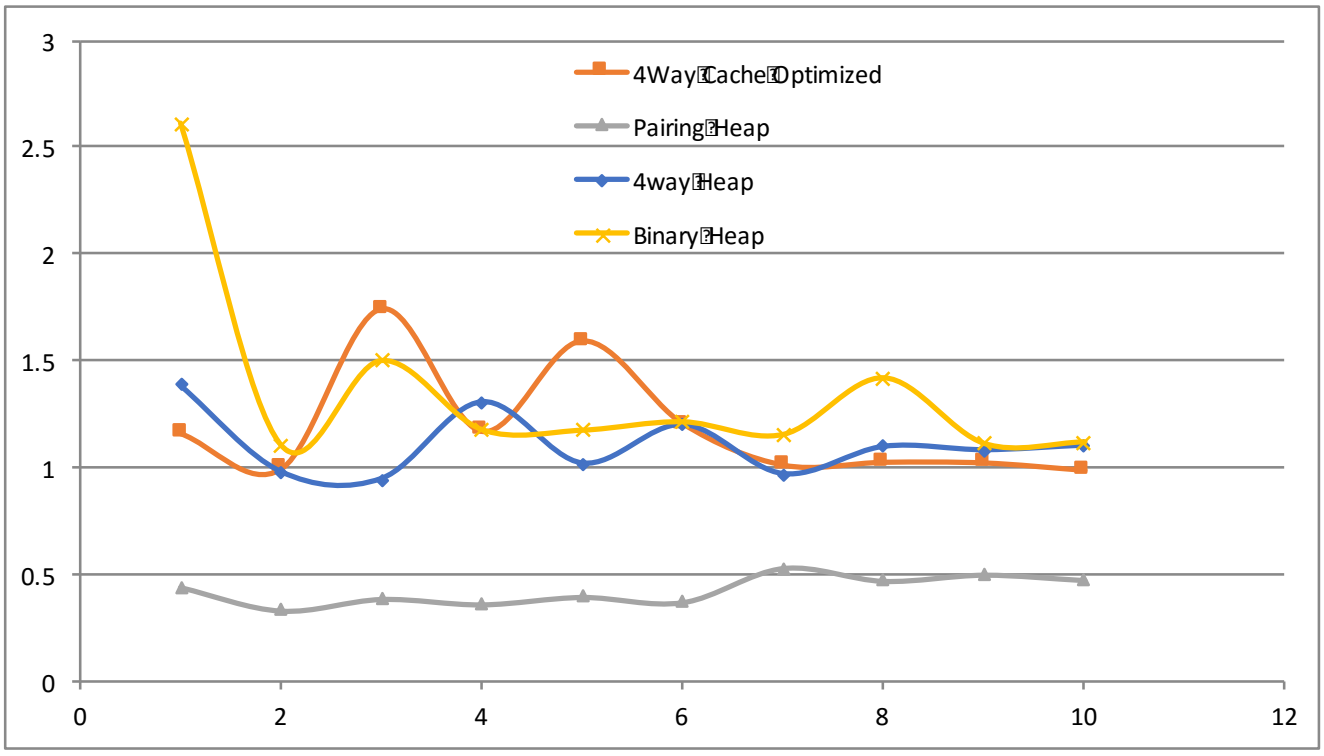
After performing careful analysis on the best kind of heap which would result in opmitized encoder performance, I came up with 4-way cache optimized as the best way (compared to pairing heap & binary heap). Below table gives a brief overview of the performance measurement over 10M lines of integer values 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).