# CSC3100 Assignment 4 Report

CHEN Ang (*118010009*)

## Problem 1: Huffman Coding

Problem 1 asks for an implementation of Huffman coding. The algorithm takes in a string $S$ of $n$ characters from an alphabet $A$, and outputs its Huffman code $C$. Huffman code is the optimal symbol-by-symbol binary code in the sense that if the distribution of the alphabet follows the character frequencies of $S$, the Huffman code has the minimum expected length among all binary codes.

### Solution

We find the Huffman code of the string by building a Huffman tree and back traversing the tree from the leaves.

To build the tree, we first count the frequency of each character contained in the string $S$. Then we build a priority queue $Q$ using a binary min-heap out of the character counts which will give us the character (node) of minimum frequencies (in case of a tie, the character with the lowest ASCII code). We then keep extracting the top two characters (nodes) in the heap with minimum frequencies and merge them into internal nodes. An internal node shall have a frequency of that of its children (the two minimum nodes) combined, and shall possess a representative character identical to that of its left child (Storing representative characters and using them to break frequency ties is not necessary for the algorithm. However it is in the special regulations of the problem to ensure uniqueness of the solution). This process is repeated for $m-1$ times, where $m$ is the number of unique characters in $S$. A last extraction of the min-node gives us the root node of tree $T$. This process is made clear with the pseudocode below.

```
BUILD-TREE(S)
    /* count character frequencies */
    A = [NIL,..,NIL]                  /* alphabet of certain size */
    m = 0                             /* number of unqiue chars */
    for each character c ∈ S
        if A[c] == NIL
            A[c] = (1, c)             /* each node is (freq, char) */
            m += 1
        else
            A[c].count += 1

    /* build queue */
    Q = {a ∈ A | a != NIL}

    /* build tree */
    for i = 1..m-1
        x = Q.EXTRACT-MIN()
        y = Q.EXTRACT-MIN()
        z = MERGE(x, y)               /* merging two nodes as specified */
        Q.INSERT(z)

    return Q.EXTRACT-MIN()            /* root of the tree */
```

To encode the original string, we traverse the string again and for each character travels up from the leave nodes of the tree until root and record every turn we make.

```
ENCODE(S)
    C = ''                      /* encoding of the string */
    for each character c ∈ S
        E = ''                  /* encoding of the char */
        n = charset[c]
        p = n.parent
        while p != NIL
            if n == p.left      /* right turn */
                E += '0'
            else
                E += '1'        /* left turn */
            n = p
            p = p.parent
        C += REVERSED(E)        /* reverse the code and concatenate */

    return C
```

Notice that we must reverse the code for each character since we are encoding from the leaves up, as supposed to decoding a code from the root down.

## Complexity

The space complexity of the algorithm is asymptotically bounded by the size of the alphabet $A$. The alphabet itself takes $O(A)$ space. The size of the queue is always smaller than the $m$, which is no more than $|A|$. Also the size of the tree is sum of $m$ leave nodes and $m-1$ internal nodes, $2m-1=O(m)=O(|A|)$.

The time complexity of `BUILD-TREE` is $O(n+|A|+m\log m)$. Counting characters takes $O(n)$ time. To build the queue we must check if each character in the alphabet is present and build a size-$m$ queue out of the present characters, taking $O(|A|+m)$ time. In building of the tree, each for loop spends $O(\log m)$ on priority queue operations and $O(1)$ on merging two nodes, which is $O(\log m)$. And so the $m-1$ loops sum up to $O(m\log m)$ in total.

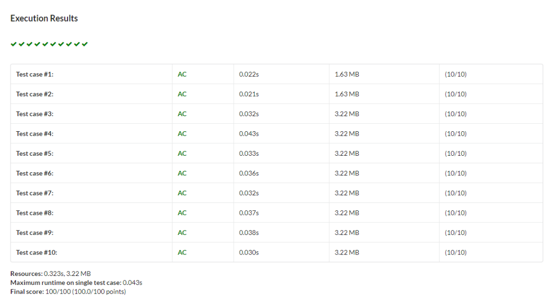
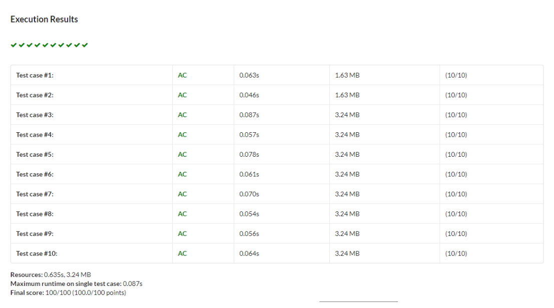
## Other Possible Solutions & Testing

Since we only wish to perform upward traversal on tree $T$. We might try to eliminate all the `left` and `right` child pointers and use only the `parent` pointer. Keeping the representative character of all internal nodes the same as their left children, we can differentiate a left and right turn in `DECODE` by comparing the representative character of the parent and its child:

```
ENCODE-CHAR-COMPARE(S)
    C = ''
    for each character c ∈ S
        E = ''
        n = charset[c]
        p = n.parent
        while p != NIL
            if n.char == p.char     /* n is left child */
                E += '0'
            else                    /* n is right child */
                E += '1'
            n = p
            p = p.parent
        C += REVERSED(E)

    return C
```

which should cut down the constant factor of the space needed by the tree by almost $2/3$. Nevertheless, when we tested the modified approach on the OJ platform, an unexpected, nearly identical space usage was obtained. Speed-wise, the method based on character comparison was almost two times slower.

| With Child Pointers (0.323s, 3.22 MB) | Without Child Pointers (0.635s, 3.24 MB) |
|---|---|
|  |  |

Execution Results

✓✓✓✓✓✓✓✓✓✓

| | | | | |
|---|---|---|---|---|
| Test case #1: | AC | 0.022s | 1.63 MB | (10/10) |
| Test case #2: | AC | 0.021s | 1.63 MB | (10/10) |
| Test case #3: | AC | 0.032s | 3.22 MB | (10/10) |
| Test case #4: | AC | 0.043s | 3.22 MB | (10/10) |
| Test case #5: | AC | 0.033s | 3.22 MB | (10/10) |
| Test case #6: | AC | 0.036s | 3.22 MB | (10/10) |
| Test case #7: | AC | 0.032s | 3.22 MB | (10/10) |
| Test case #8: | AC | 0.037s | 3.22 MB | (10/10) |
| Test case #9: | AC | 0.038s | 3.22 MB | (10/10) |
| Test case #10: | AC | 0.030s | 3.22 MB | (10/10) |

**Resources:** 0.323s, 3.22 MB
**Maximum runtime on single test case:** 0.043s
**Final score:** 100/100 (100.0/100 points)

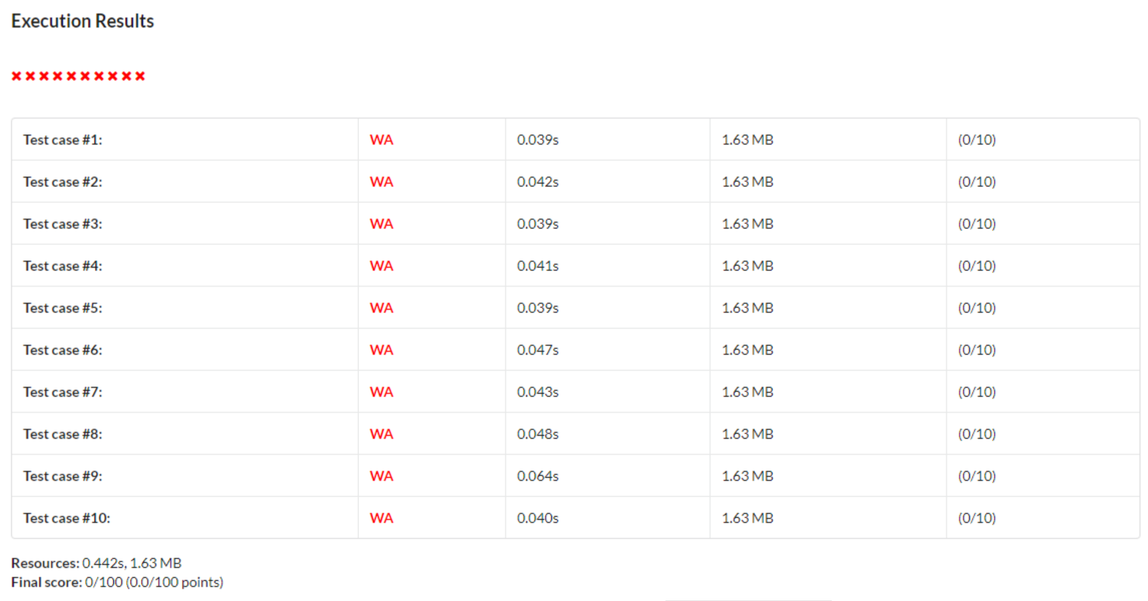
Execution Results

✓✓✓✓✓✓✓✓✓✓

| | | | | |
|---|---|---|---|---|
| Test case #1: | AC | 0.063s | 1.63 MB | (10/10) |
| Test case #2: | AC | 0.046s | 1.63 MB | (10/10) |
| Test case #3: | AC | 0.087s | 3.24 MB | (10/10) |
| Test case #4: | AC | 0.057s | 3.24 MB | (10/10) |
| Test case #5: | AC | 0.078s | 3.24 MB | (10/10) |
| Test case #6: | AC | 0.061s | 3.24 MB | (10/10) |
| Test case #7: | AC | 0.070s | 3.24 MB | (10/10) |
| Test case #8: | AC | 0.054s | 3.24 MB | (10/10) |
| Test case #9: | AC | 0.056s | 3.24 MB | (10/10) |
| Test case #10: | AC | 0.064s | 3.24 MB | (10/10) |

**Resources:** 0.635s, 3.24 MB
**Maximum runtime on single test case:** 0.087s
**Final score:** 100/100 (100.0/100 points)

After pondering numerous potential reasons for this unexpected result, the most convincing explanation seems to be that the vast of the space consumption, instead of coming from the Huffman tree, is really due to the big constant factor introduced by the `<iostream>` library. This is supported by the drastic drop of space usage after removal of `getline()` function.



Execution Results

✗✗✗✗✗✗✗✗✗✗

| | | | | |
|---|---|---|---|---|
| Test case #1: | WA | 0.039s | 1.63 MB | (0/10) |
| Test case #2: | WA | 0.042s | 1.63 MB | (0/10) |
| Test case #3: | WA | 0.039s | 1.63 MB | (0/10) |
| Test case #4: | WA | 0.041s | 1.63 MB | (0/10) |
| Test case #5: | WA | 0.039s | 1.63 MB | (0/10) |
| Test case #6: | WA | 0.047s | 1.63 MB | (0/10) |
| Test case #7: | WA | 0.043s | 1.63 MB | (0/10) |
| Test case #8: | WA | 0.048s | 1.63 MB | (0/10) |
| Test case #9: | WA | 0.064s | 1.63 MB | (0/10) |
| Test case #10: | WA | 0.040s | 1.63 MB | (0/10) |

**Resources:** 0.442s, 1.63 MB
**Final score:** 0/100 (0.0/100 points)

## Possible Improvement

We can improve the space complexity of the algorithm by replacing the the array $A$ by a hashmap keyed by characters and storing the frequencies as items.

```
BUILD-TREE-HASHMAP(S)
    /* count character frequencies */
    A = Hashmap()
    for each character c ∈ S
        if c ∈ A
            A[c].count += 1
        else
            A[c] = 1
    m = A.size                          /* number of unqiue chars */
...
```

This approach decreases the space complexity to $O(m)$ as we no longer need to have to keep the entire alphabet. This is particularly helpful when the string to encode only contains a limited portion of characters in the alphabet, namely $m$ << $|A|$.

In addition, the $O(|A|)$ term in the time complexity can also be reduced to $O(m)$ since we no longer need to scan the entire alphabet when building the queue. The total time complexity thus becomes $O(n + m + m \log m) = O(n + m \log m)$.

## Problem 2: Dijkstra

Problem 2 asks for an implementation of Dijkstra's algorithm, which finds, in a weighted directed graph $G = (V[1..N], E[1..M])$, all single-source shortest paths starting from some vertex $V[i], 1 \leq i \leq N$.

### Solution

We represent the graph $G$ using an array $\text{adj}$ of $N$ adjacency lists. The $i$-th list $L_i$ stores all edges starting from vertex $V[i]$ to some $V[j]$. Since the starting vertex is determined by index $i$, we can represent an edge $(V[i], V[j], w)$ with only the end vertex ID and the weight of the edge:

$$(j, w) \in L_i \iff (V[i], V[j], w) \in E$$

Code-wise, this is implemented as a `Vertex` struct.

```
/* Vertex struct for adjacency list */
struct Vertex {
   Vertex* next;
   int     vnum;
   u32     dist;

   Vertex(Vertex* next, int vnum, u32 dist) {
       this->next = next;
       this->vnum = vnum;
       this->dist = dist;
   }
};
```

We also implement a priority queue $Q$ using a binary min-heap to allow quick access to the closest vertex. At any moment, $Q$ contains all vertices yet to be examined. The queue is keyed by the vertices' distance to the start, so that the top of the queue always indicates the currently closest vertex:

$$Q.\text{extractMin}() = \arg\min_i \, d(V[i])$$

In addition to regular priority queue operations, we also maintain a `loc` array which stores the vertices' index within the min-heap. This way we can access a vertex in the heap directly by its vertex ID without having to search for that ID in the heap.

The main algorithm is consistent with the textbook implementation, illustrated by the pseudocode below

```
DIJKSTRA(G, s)
1   Q = G.V
2   d = [∞,..,∞] of size N
3   d[s] = 0
4   while Q != ∅
5       u = Q.EXTRACT-MIN()
6       for each vertex v ∈ G.adj[u]
7           RELAX(u, v)
```

## Complexity

The space complexity of the algorithm is $O(N+M)$. The $O(N)$ part comes from the priority queue storing $N$ vertices, the distance array of size $N$ as well as the `adj` array of size $N$. The $O(M)$ part comes from the $M$ `Vertex` structs representing the $M$ edges in total stored in the adjacency lists. The time complexity of the algorithm is $O(N \log N + M \log N)$. Initialization of the priority queue and distance array on lines 1 to 3 takes $O(N)$ time. Line 5 takes $O(\log N)$ time with min-heap implementation, and is executed exactly $N$ times, taking $O(N \log N)$. Line 7 calls the `DECREASE-KEY` method of $O(\log N)$ complexity. In each for-loop method is executed at most $M_i$ times, where $M_i$ is the number of edges starting from $E[i]$. In total Line 6 and 7 takes $\sum_{i=1}^{n} O(\log N) O(M_i) = O(M \log N)$ time. Therefore the total time complexity is $O(N \log N + M \log N)$.

## Other Possible Solutions & Comparison

Variations on Dijkstra's algorithm boil down to how one chooses to represent the graph $G$ as well as how `EXTRACT-MIN` is performed. As a first variation, we might choose to represent $G$ not by adjacency lists but an adjacency matrix $M$ of size $N \times N$, in which every entry

$$M_{i,j} = \begin{cases} \infty & \text{if there is no edge from } V[i] \text{ to } V[j] \\ \min_{V[i] \xrightarrow{e} V[j]} w(e) & \text{otherwise} \end{cases}$$

This representation brings the space complexity of $G$ up to $\Theta(N^2)$. And it does not help with the speed complexity because we need to traverse every edge anyhow. The for-loop on line 6-7 now takes $\Theta(N)$ as supposed to $\Theta(m_i)$ since every entry on row $i$ of the matrix needs to be checked and edges potentially relaxed.

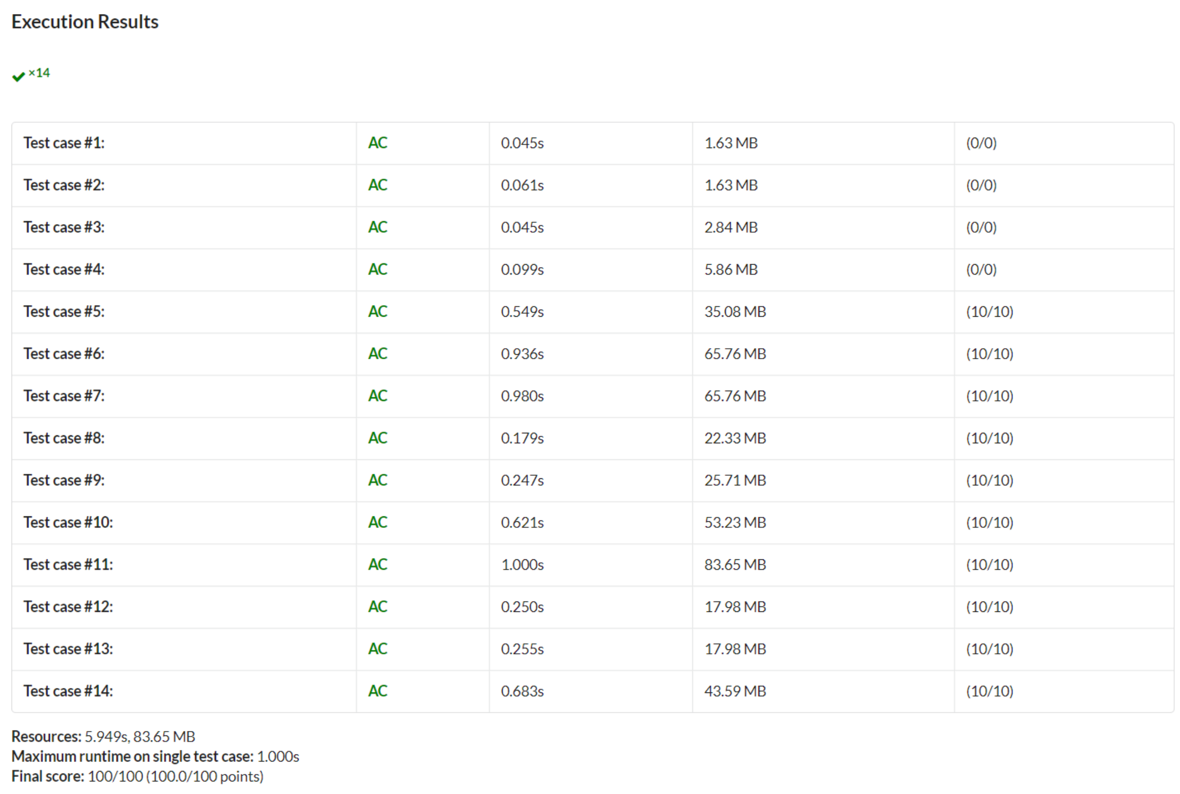
A naive alternative to perform `EXTRACT-MIN` is to simply maintain all unexamined vertices in an reversely-ordered linked list (by distance). The initialization steps still takes $O(N)$. `EXTRACT-MIN` can be achieved by deleting the head node from the list and returning the vertex ID in $O(1)$. However the `RELAX` routine on line 7 slows down to $O(N)$ for finding the correct position to insert the vertex. And so the time complexity increases to $O(N + MN) = O(MN)$.

Another similar approach is with an unordered array $A$ where each entry $A[i] = d(V[i])$. Initialization remains at $O(N)$. The relaxation routine becomes $O(1)$, but `EXTRACT-MIN` slows down to $O(N)$ since we must scan through all entries. The total complexity in this case is $O(N^2 + M) = O(N^2)$.

A more sophisticated approach is to arrange the vertices in a self-balancing BST (also need to maintain a map from vertex ID to pointers to nodes in BST). Initializing BST still takes $O(N)$ (inserting non-source vertices level by level and insert the source at last). Each `EXTRACT-MIN` takes $O(\log N)$ (finding minimum node in $O(\log N)$ and deleting in $O(1)$). Decreasing distance now takes two steps: deleting the vertex and then inserting it with the new distance, taking $O(\log N)$ time. So the total time complexity is still $O(N \log N + M \log N)$.

## Testing

We tested the program on the OJ platform and obtained the following results.



**Execution Results**

✓ ×14

| | | | | |
|---|---|---|---|---|
| Test case #1: | AC | 0.045s | 1.63 MB | (0/0) |
| Test case #2: | AC | 0.061s | 1.63 MB | (0/0) |
| Test case #3: | AC | 0.045s | 2.84 MB | (0/0) |
| Test case #4: | AC | 0.099s | 5.86 MB | (0/0) |
| Test case #5: | AC | 0.549s | 35.08 MB | (10/10) |
| Test case #6: | AC | 0.936s | 65.76 MB | (10/10) |
| Test case #7: | AC | 0.980s | 65.76 MB | (10/10) |
| Test case #8: | AC | 0.179s | 22.33 MB | (10/10) |
| Test case #9: | AC | 0.247s | 25.71 MB | (10/10) |
| Test case #10: | AC | 0.621s | 53.23 MB | (10/10) |
| Test case #11: | AC | 1.000s | 83.65 MB | (10/10) |
| Test case #12: | AC | 0.250s | 17.98 MB | (10/10) |
| Test case #13: | AC | 0.255s | 17.98 MB | (10/10) |
| Test case #14: | AC | 0.683s | 43.59 MB | (10/10) |

**Resources:** 5.949s, 83.65 MB
**Maximum runtime on single test case:** 1.000s
**Final score:** 100/100 (100.0/100 points)

## Possible Improvements

One possible way of decreasing the time complexity is by exploiting the fact that a vertex cannot be further relaxed once it has been visited. We may maintain an Boolean array `visited` (or bitmap if we wish to save more space) to record whether a vertex has already been visited. Then when we attempt to relax an edge, we skip those with an endpoint already visited.

```
DIJKSTRA-PLUS(G, s)
    Q = G.V
    visited = [False,..,False] of size N
    d = [∞,..,∞] of size N
    d[s] = 0
    while Q != ∅
        u = Q.EXTRACT-MIN()
        for each vertex v in G.adj[u]
            if visited[v] == False
                RELAX(u, v)
        visited[u] = True
```

Another more significant improvement of the algorithm is to replace the binary heap with a Fibonacci heap for implementation of the priority queue.

```
DIJKSTRA-FIB(G, s)
fibQ = G.V
d = [∞,..,∞] of size N
d[s] = 0
while fibQ != ∅
    u = fibQ.EXTRACT-MIN()
    for each vertex v in G.adj[u]
        RELAX(u, v)
```

With Fibonacci heap `DEACREASE-KEY` of the `RELAX` procedure can be achieved in constant amortized time, while `EXTRACT-MIN` remains $O(\log N)$ amortized. Thus the algorithm speeds up to $O(M + N \log N)$ in amortized time.