CS 38, PS 5

Codename : Jambul
Late submission : No

May 19, 2023

1 Problem 1

(a) The table below maps the characters in the alphabet to their codewords in an optimal Huffman code for the alphabet.

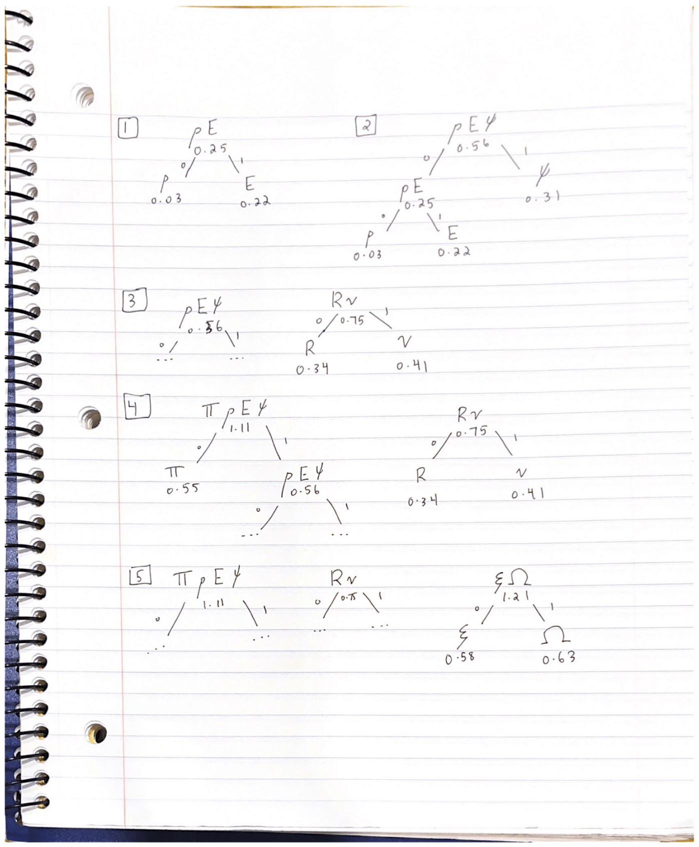
Character	Codeword
κ	010
A	1101
B	0001
γ	1110
ζ	1100
K	1010
λ	0011
M	0110
μ	1000
Γ	01111
η	10111
Θ	11111
θ	10011
Λ	01110
N	00001
α	100101
β	101101

Character	Codeword
Z	101100
H	001010
Ξ	000000
Π	001001
σ	111100
ϕ	001011
ω	111101
ϵ	1001001
ξ	0010000
π	0000010
Ω	0010001
ν	10010001
R	10010000
Ψ	00000111
E	000001101
ρ	000001100

We will now describe the first five steps we employed to produce these results. We first loaded all the characters into a min heap with keys equal to their frequencies. In each of the first five steps, we popped the two characters $i, j, \text{key}(i) \leq \text{key}(j)$ with the lowest frequencies, merged them as the leaves of a binary subtree with root ij, where $\text{left}(ij) = i, \text{right}(ij) = j$ and inserted ij back into the heap with key equal to the sum of i and j's.

The characters popped off and added back on at each step are shown in the table below. A picture of the Huffman tree corresponding to each step is shown after that.

Step	i	j	ij	key(ij)
1	ρ	E	$\rho\ E$	0.25
2	ρE	ψ	$\rho E\ \psi$	0.56
3	R	ν	$R\nu$	0.75
4	π	$\rho E\ \psi$	$\pi\rho E\psi$	1.11
5	ξ	Ω	$\xi\Omega$	1.21



(b) Consider the alphabet $\Sigma = \{a, b, c, d\}$ of characters with associated frequencies $p_a = 25, p_b = 15, p_c = 35$, and $p_d = 30$. Set the costs of encoding "0" and "1" to be $w_0 = 2, w_1 = 4$, respectively.

Then the described algorithm will produce a Huffman tree corresponding to the encoding

$$a \to 00$$

$$b \to 01$$

$$c \to 10$$

$$d \to 11.$$

This encoding will have cost

$$2w_0p_a + (w_0 + w_1)p_b + (w_0 + w_1)p_c + 2w_1p_d = 640.$$

However, observe that if we applied the encoding

$$a \to 10$$

$$b \to 01$$

$$c \to 00$$

$$d \to 11,$$

then this cost would be

$$2w_0p_c + (w_0 + w_1)(p_b + p_a) + 2w_1p_d = 620 < 640,$$

so indeed the algorithm presented does not always produce an optimal encoding.

2 Problem 2

Algorithm 1. (Cycle-Detecting DFS.) This algorithm takes as input a graph $G = (V, E)$, a dictionary m that maps from vertices in V to boolean values, and a vertex $v \in V$. It will be used as a subroutine in the solution given below.

```
1. Create a stack $s$ initially holding the 2-tuple $(v, 0)$.
2. While $s$ is not empty:
   (a) Pop the top element $(c, l)$ off $s$.
   (b) If $l = 3$ and $v$ is a neighbor of $c$: Return True.
   (c) Else if $l = 3$ and $v$ is not a neighbor of $c$: Set $m[c]$ = False and continue.
   (d) Set $m[c]$ = True.
   (e) For $w$ a neighbor of $c$:
       i. If $m[w]$ = False:
          A. Append $(w, l + 1)$ to $s$.
3. Return False.
```

Algorithm 2. (Cycle Detection in Undirected Graph.) We will use a depth-first search to find all paths of length 3 from each vertex in the graph. If any such path ends at the vertex it started at, then there exists a cycle of length 4 in the graph. Let our input graph be $G = (V, E)$.

1. Create a dictionary m that maps each vertex $i \in V$ to `False`.
2. For $i \in V$:
    (a) Call "Cycle-Detecting DFS" on $G = G, m = m, v = i$, and denote the result o.
    (b) If $o =$ `True`: Return `True`.
    (c) Set $m[i] =$ `True`.
3. Return `False`.

Proof of correctness. We iterate over all vertices in the graph. At each of these iterations, we will either stop and return that we have found a cycle or we will continue. If we continue, we will never need to check that node again, as we know that it is not part of any 4-cycle.

To determine whether to stop and return that we have found a cycle, we can apply a modified version of the standard depth-first search algorithm. In this modification, we wish to search for all paths of length 4 from a given starting node and check whether the end of any of these paths is the starting vertex itself. If one is, then we have found a cycle; if not, we mark the vertex as not being a member of a 4-cycle and continue.

Furthermore, when running our modified depth-first search algorithm, we may ignore all paths that include nodes previously visited within that specific iteration in addition to all nodes that have been marked as non-members of a 4-cycle from previous iterations. This will not affect our algorithm's correctness, as we know after visiting and processing all paths from a node that it cannot be a part of a 4-cycle if the algorithm chooses to continue. (Otherwise, we would have stopped and returned that we found a 4-cycle.)

Complexity. The algorithm loops over $\mathcal{O}(|V|)$ vertices, and for each vertex, it runs a modified depth-first search on all surrounding vertices.

Importantly, the modified DFS ignores vertices in the graph that have already been searched and are known to not be a part of a 4-cycle. Due to this, the algorithm only iterates over the edges of the graph a fixed number of times, resulting in a runtime $\mathcal{O}(|E|) = \mathcal{O}(|V|^2)$, as desired.

3 Problem 3

Algorithm 3. (Minimum Bottleneck Distance.) We will modify Dijkstra's algorithm slightly to solve the problem. We assume we are given two vertices u, v in a directed, weighted graph $G = (V, E, w)$ as input, with $|V| = n, |E| = m$, and we wish to find the bottleneck distance from u to v.

```
1. Create an empty set $S$.
2. Create a dictionary $d$ that maps $d(w) = \infty$ for every $w \in V, w \neq u$. Set
   $d(u) = -\infty$.
3. Create a min heap $q$ that contains the 2-tuple $(-\infty, u)$ and is ordered
   based on the first entry in each 2-tuple inserted into it.
4. While $|S| < n$:
   (a) Pop the minimum element $(\_, z)$ off $q$.
   (b) If $z \in S$: Continue.
   (c) Else:
       i. Add $z$ to $S$.
       ii. For $w \in \{$the neighbors of $z\}$:
           A. Set $d(w) = \min\{d(w), \max\{d(z), w(z, w)\}\}$.
           B. Push $(d(w), w)$ onto $q$.
5. Return $d(v)$.
```

Proof of correctness. We will prove that the distance function

$$d(u, y) = \min_{z \in P(y)} \{d(u, y), \max\{d(u, z), w(z, y)\}\},$$

for $P(y)$ the predecessors of y, is the correct one for the problem (i.e., $d(u, y) = g(u, y), y \in V$). We will prove this claim by induction. We set key$(u) = -\infty$ and never change this, so $d(u, u) = -\infty = g(u, u)$.

Suppose then that $d(u, y) \neq g(u, y)$ for some $y \in V$. Let $u \to y$ be the shortest bottleneck path from u to y. Then let U be the set of vertices that are visited on $u \to y$ prior to y. Suppose that the distance scores of all these vertices are correct (this is our induction hypothesis), and let t be the last vertex on $u \to y$ in U. Finally, let $u \to t$ be the minimum bottleneck path from u to t.

Either the weight of the edge ending at y is the bottleneck distance of $u \to y$, or it is less than the bottleneck distance. In the latter case, we know that the minimum bottleneck distance will be the maximum weight $d(u, t)$ of an edge in $u \to t$, as we have assumed $u \to y$ is the shortest bottleneck path from u to y. In the former case, we note that $u \to t \to y$ is a minimum bottleneck path from u to y, as $w(t, y)$ will be the bottleneck weight of that path.

Then, since either $w(t, y) > d(u, t)$ or $w(t, y) \leq d(u, t)$, the bottleneck distance of the path $u \to t \to y$ will be $d^* = \max\{d(u, t), w(t, y)\}$. But this is in fact the result of $d(u, y)$, as we have taken $u \to t$ to be the shortest bottleneck path from u to t, and $d(u, y)$ takes the minimum of d^* over all paths from u to y. Thus indeed $d(u, y) = g(u, y)$ for all $y \in V$, and in particular, $d(u, v) = g(u, v)$.

Complexity. Algorithm 1 has the same runtime as Dijkstra's algorithm. It differs only in the way in which it chooses to update the scores of each vertex, which is done in constant time. It therefore can achieve a runtime of $\mathcal{O}(m + n \log n)$ when using a Fibonacci heap.