4041 Homework 6

Fletcher Gornick

November 20, 2021

16.3

16.3-2

Prove that a binary tree that is not full cannot correspond to an optimal prefix code.

First, we should note that in order for a binary tree to be full, it must be the case that evry non-leaf node has exactly 2 children. Now we will show that an optimal solution cannot be obtained from anything other than a full binary tree. We will do this by taking two separate cases.

Assume, to the contrary, that we can obtain an optimal solution with a non-leaf node containing just one child node. In this first case, let's assume this leaf is at the furthest depth of the tree. This leaf node has an n-length bit sequence to traverse from the root to the leaf. But if we simply replace the leaf's parent with the leaf itself, it doesn't change the optimality of the rest of the leaves, but this one particular leaf now has an $n-1$-length bit sequence, thus our solution has been further optimized, contradicting our original claim.

For the second case, let's assume we have an optimal solution with a non-leaf node only containing one child leaf x, and this child is somewhere above the lowest depth. Now if we take the two nodes of greatest depth, call them y and z, both of length m, then we can just take node y and move it to be the neighbor of x, so y now has a new bit sequence length less than or equal to m. We can also perform the same operation we did in the last paragraph to z to give it a new bit sequence length of $m-1$, thus our new solution is more optimal, which is a contradiction.

Assuming in the previous case that x isn't a leaf node, then we could just replace the parent with x, making it so all the children of x contain one less bit, also making the solution more optimal, and contradicting our original claim.

16.3-6

Suppose we have an optimal prefix code on a set $\mathbf{C} = \{\mathbf{0}, \mathbf{1}, \ldots, \mathbf{n-1}\}$ of characters and we wish to transmit this code using as few bits as possible. Show how to represent any optimal prefix code on C using only $\mathbf{2n - 1 + n \lceil \lg n \rceil}$ bits. (*Hint:* Use $\mathbf{2n-1}$ bits to specify the structure of the tree, as discovered by a walk of the tree).

As we proved in the previous question, we need a full binary tree representation of our characters for an optimal solution. We also know that a tree with n leaves has exactly $2n - 1$ nodes from previous chapters. So if we treat a binary tree as an array of bits, we can assign 0 to non-leaf nodes and 1 to leaf-nodes (or the opposite, doesn't really matter). This is where our $2n-1$ bits come from.

From *homework 1 problem 2*, we know an n-element heap has a height of $\lfloor \lg n \rfloor$, so if we take a $2n - 1$ element heap, we get a heap height of $\lfloor \lg(2n-1) \rfloor \leq \lfloor \lg 2n \rfloor = \lfloor 1 + \lg n \rfloor = \lceil \lg n \rceil$. This means that every character can be represented with at most $\lceil \lg n \rceil$ bits, and since there are exactly n characters, we need at most $n \lceil \lg n \rceil$ bits to represent all the characters.

So in order to properly transmit our huffman code, we need $2n - 1$ bits to encode the binary tree structure. We also need another $n \lceil \lg n \rceil$ bits for the bit-representation of our n characters. So in conclusion, we need $2n - 1 + n \lceil \lg n \rceil$ bits to represent our optimal prefix code.

22.1

22.1-1

Given an adjacency-list representation of a directed graph, how long does it take to compute the out-degree of every vertex? How long does it take to compute the in-degrees?

Out-Degree: The out-degree of a vertex is the number of other vertices it points to. In an adjacency-list, to find the out-degree of a vertex x, we simply jump to position x on the list, and loop through the linked list of nodes representing the connected vertices of x, incrementing at each element in x's corresponding linked-list.

To calculate the out-degree for every vertex, we must loop through every vertex on the adjacency-list and loop through all their corresponding edges. And since we're talking about a directed graph, we must check every edge one time. This gives us a running time of $\Theta(|V| + |E|)$, because we must loop through each vertex, and at each vertex, we must loop through each representative edge.

In-Degree: The In-Degree of a vertex is the number of other vertices that point to it, so if we wish to find the in-degree of a vertex, we must check the vertices that every other node points to, incrementing every time we land on a vertex matching our target vertex.

There's a fast way to do this though. If we create a new array of integers of length $|V|$, where $|V|$ is the number of vertices, then we can keep track of the out-degrees of every node as we go. assuming our verticies are numbered $1, \dots, n$, we can loop through all the vertices of our adjacency-list, then go through all their connected vertices. So if vertex i in the adjacency-list points to vertex j $(1 \leq i, j \leq n)$, we can increment the value at position j of out out-degree array by 1. Doing this gives us the same time complexity of $\Theta(|V| + |E|)$.

22.1-3

The *transpose* of a directed graph $\mathbf{G = (V, E)}$ is the graph $\mathbf{G^T = (V, E^T)}$, where $\mathbf{E^T = \{(v, u) \in V \times V : (u, v) \in E\}}$. Thus, $\mathbf{G^T}$ is $\mathbf{G}$ with all its edges reversed. Describe efficient algorithms for computing $\mathbf{G^T}$ from $\mathbf{G}$, for both the adjacency-list and adjacency-matrix representations of $\mathbf{G}$. Analyze the running times of your algorithms.

For the algorithm, we will let A denote the adjacency-list / adjacency-matrix for the following algorithms. Our algorithm will take in A and return the corresponding transposed list / matrix.

```
transpose_adjacency_list(A)
    A^T = new |V|-length list
    for each vertex u ∈ A
        for each vertex v ∈ A[u]
            A^T[v].append(u)
    return A^T
```

This algorithm requires us to loop through every vertex of our graph, then every corresponding connected vertex, representing an edge. Therefore, this algorithm runs in $\Theta(|V| + |E|)$ time.

```
transpose_adjacency_matrix(A)
    A^T = new $|V| \times |V|$ matrix
    for i in A.rows
        for j in A.cols
            A^T[j,i] = A[i,j]
    return A^T
```

This algorithm must loop through every element in a $|V| \times |V|$ matrix, meaning the running time is $\Theta(|V|^2)$.

22.1-5

The *square* of a directed graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ is the graph $\mathbf{G^2} = (\mathbf{V}, \mathbf{E^2})$ such that $(\mathbf{u}, \mathbf{v}) \in \mathbf{E^2}$ if and only if G contains a path with at most two edges between u and v. Describe efficient algorithms for computing $\mathbf{G^2}$ from G for both the adjacency-list and adjacency-matrix representations of G. Analyze the running times of your algorithms.

For our adjacency-list implementation, to avoid duplicates, we can first make a $|V| \times |V|$ 2-dimensional array of booleans to depict whether or not a vertex will contain another vertex in it's adjacency-list for G^2. Then we can go through that 2D array, adding vertices if their corresponding boolean is true.

```
$G^2$AdjacencyList(G)
    Adj$^2$ = new $|V|$-length array
    A = new $|V| \times |V|$ false boolean matrix
    for each vertex $u \in$ G.V
        for each vertex $v \in$ Adj[u]
            A[u,v] = true
            for each vertex $w \in$ Adj[v]
                A[u,w] = true

    for each vertex $u \in$ G.V
        for each vertex $v \in$ G.V
            if A[u,v] == true && u $\neq$ v
                Adj$^2$[u].add(v)
    return Adj$^2$
```

For each edge in G, we can go through all of the vertices connected to a vertex in $\Theta(|V|)$ time, thus we get a running time of $\Theta(|V||E|)$ to calculate our boolean matrix. Now, to assign the values to our new adjacency-list, we must go through our entire $|V| \times |V|$ matrix, this gives us a running time of $\Theta(|V|^2)$, so our overall running time is $\Theta(|V||E|)$.

Now, say we have an adjacency-matrix A that's of size $|V| \times |V|$. If there's an edge from u to v, then a_{uv} is true, otherwise it's false. So if we were to take A^2, the new truth values would represent all the edges of length exactly 2. If we use 1 to represent true and 0 to represent false, then taking A^2 will give us a new matrix, but it could contain numbers greater than 1, if there are multiple paths of length 2 from one vertex to another, so we can just set all nonzero elements to 1 again.

Now that we have our new adjacency-matrix depicting all 2-length edges, we can now add it to our original matrix, giving us a new matrix depicting edges of length 1 and 2 between vertices. Our algorithm looks something like this...

```
$G^2$AdjacencyMatrix(G)
    $A^2$ = G.Adj * G.Adj
    $A^2$ += G.Adj
    for i in $A^2$ rows
        for j in $A^2$ cols
            if $A^2$[i,j] > 1 then $A^2$[i,j] = 1
    return $A^2$
```

The most expensive operation was the matrix multiplication done in the first line. This is done in $\Theta(n^3)$ time, and since the matrix is of size $|V| \times |V|$, it takes $\Theta(|V|^3)$ time. This could technically be reduced to $\Theta(|V|^{2.3728596})$ with the Coppersmith-Winograd algorithm, but I won't go into that. As for the rest of the algorithm, the addition is a $\Theta(|V|^2)$ operation, along with the reassignment of all the matrix elements. This gives us an overall running time of $\Theta(|V|^3)$.

Also worth noting, we don't neeeed to set all the nonzero values to one in the adjacency-matrix, information about how many edges connect one vertex to another could be useful. So if we wanted to keep that information, then we could just omit the double for-loop part of the algorithm.