HW 7 Algorithms – Dakota Krogmeier

1. The maximal length of an alphabet of n characters is n-1, because the Huffman tree needs to have the remaining leaves each be on its own level. The height of a Huffman tree is n-1, which coincides with the maximal length of a codeword.
2. Key Idea: The key idea is to input two separate arrays, iterating through each one to find the two smallest elements within the first two elements of in each array. Add the sum of the smallest two and add it to the second array, deleting the elements in the process.

Algorithm: LinearHuffman(G,F);

Input: Two arrays, G that holds the weights of each new tree, and F that holds the given frequencies in non decreasing order.

Output: Total Cost of Huffman tree

Process:

Initialize empty list G;

Cost <- 0;

While |G|+|F|>2;

Let u, w be the smallest elements from the first two elements of G and F;

Remove u,w from F, insert into G;

Cost <- Cost + (u+w);

Return Cost;

1. Key Idea: The key idea is to add minimum weights into a priority queue, adding the sum of the minimum weights to the priority queue. The runtime of this algorithm takes the computation of deleting the smallest element as taking more time at 2(n-1). The time efficiency for this implementation will be O(n^2).

Algorithm: HuffmanTree(W[0..n-1])

Input: An Array W of weights

Output: A full Huffman tree

Process:

#Initialize the priority Q with one node trees

#Initialize Empty tree T

#Tleft<-the minimum weight tree in the Q

#Delete the minimum weight tree in the Q

#Tright<-the minimum weight tree in the Q

#Delete the minimum weight tree in the Q

#Create a new tree by combining Tleft and Tright parts, //weight of new tree is equal to sum of the weights

#Insert the new tree into the priority Q

#Return T

4. Key Idea: Set each vertex to be the same weight, then traverse the set based on which vertex you come across. Prims algorithm is a greedy algorithm, thus we will move the first vertex we see into the set of visited vertices.

Algorithm: MSTUnweighted(G, v)

Input: Undirected connected graph G = (V,E), with a source vertex v.

Output: A spanning tree p[1…V]

Process:

Set the weight of all vertices to 1;

Set key <-infinity;

Set the color of all vertices to white;

Q<-empty list;

Insert(Q,s);

While Q != empty;

Assign min value in Q to u;

For edge next to u;

If edge is white

Color edge black and insert into Q;

Key(v) <- (u,v);

p(v) <- u;

Else;

Then if key(v) > (u,v);

Then key(v) <-(u,v);

p(v)<-u

color(v) <-Black;

return p;

5. Key Idea: Iterate through an adjacency matrix looking for values that prove a path exists.

Algorithm: PathExists(A, i, j)

Input: Adjacency matrix and two vertices(where the vertex numbers correspond to the rows/columns of the adj matrix.

Output: A bool

Process:

Set bool = False;

For [i][j] in A;

Iterate through adj list;

If value returns 1;

Bool = true;

Else;

Bool = false;

Euler Path: If the walk travels along every edge exactly once.

Euler Circuit: If the walk travels along every edge once and the starting/ending vertices are the same. A graph must be connected for either to be possible.

Euler Path Characteristic: If a graph G has a Euler Path, then it must have exactly two odd vertices.

Euler Circuit Characteristic: If a graph G has a circuit then all of its vertices must be even.

6. Algorithm EulerPath(G=(V,E)):

Input: A graph G = (V,E) represented as an array list.

Output: A Boolean True if G admits an Euler Path, and False if G does not admit an Euler Path.

Process:

Bool = false;

Start with any vertex that’s non-zero;

Color[v] = white; // set vertex to white

For u in in adj[v]: //choose any edge

Erase the edge v-u;

Repeat with new vertex;

Color[v] = black; //set vertex to black

For v in V;

Check if all black;

If true;

Bool = True;

Else;

Return bool;