Juan Lopez

4702042

COP4534

Dr. Ning

**Homework 4**

**1.**

Several algorithms can sort a list of 0/1-valued records. We can use counting sort to sort these records in O(n) time. However, counting sort is not in-place. We could also count the number of 0s (x = number of 1s) and 1s (y = number of 1s) in the list and simply replace the first x elements with 0 and the rest of the elements with 1. This algorithm, however, would not be stable. In addition, we could use insertion sort, which is both in-place and stable. However, it is not very efficient (O(n^2) running time). The following algorithm is in-place, stable, and has a running time complexity of O(n):

Input: An unsorted array of size n containing the integers 0 or 1.

Output: A sorted array

Algorithm:

Sort(int[] array)

{  
                        int zeroCount = 2;  
                        int oneCount = 0;

                        for (int i = 0; i < array.Length;  i++)  
                        {  
                                if(array[i] == 0)  
                                {  
                                        array[i] = zeroCount;  
                                        zeroCount++;  
                                }  
                        }

                        oneCount = zeroCount;  
                        for (int i = 0; i < array.Length; i++)  
                        {  
                                if (array[i] == 1)  
                                {  
                                        array[i] = oneCount;  
                                        oneCount++;  
                                }  
                        }

                        for (int i = 0; i < array.Length; i++)  
                        {  
                                while ((array[i] - 2) != i)  
                                {  
                                        //swap  
                                        int temp = array[i];  
                                        array[i] = array[array[i] - 2];  
                                        array[temp - 2] = temp;  
  
                                }  
                        }  
  
                        for (int i = 0; i < zeroCount-2; i++)  
                        {  
                                array[i] = 0;  
  
                        }  
  
                        for (int i = zeroCount-2; i < oneCount-2; i++)  
                        {  
                                array[i] = 1;  
  
                        }  
  
}

This algorithm first iterates through the list, replacing occurrences of 0 with zeroCount. Each time we replace a 0, we increment zeroCount. The count begins at 2 so that it is distinct from the occurrences of 1. Then, we need to replace the occurrences of 1 with oneCount. Now, we have an array containing integers, which we can use to sort the array. Each integer (minus 2) corresponds to the correct index in the sorted array. We iterate through the list, moving each integer to its correct index in the array by swapping. We only need at most n swaps, so this sort is accomplished in O(n) time. We now have a sorted list of integers. We now need to replace the first zeroCount integers with 0 and the rest of the integers with 1. This algorithm is therefore in-place because we do not need an auxiliary array, and it is in place because the relative position of records with the same value is preserved.

**2.**

If we use an adjacency list, a BFS requires O (n + m) time, however if we use an adjacency matrix, the time required is O(). As shown in class, the critical part of the BFS algorithm involves enqueing adjacent unexplored vertices. With an adjacency list, the list of adjacent vertices is immediately available. The summation of all adjacent vertices in the graph is O(m) because 2m = sum of the degrees of all vertices. With the initialization, the BFS algorithm runs in O(n + m) time using an adjacency list.

However, with an adjacency matrix, to determine the adjacent vertices of vertex x, we must scan the entire row in the adjacency matrix corresponding to x. Each row is of size n, so determining the adjacent vertices is O(n). We must examine n vertices, checking the adjacent vertices of each vertex, so the total running time must be O().

**3.**

To determine if an undirected graph contains a cycle, we can use the DFS algorithm with minor modifications:

Input: An undirected graph G

Output: “Yes” - if there is a cycle in G. “No”- otherwise.

Algorithm:

DFS-cycleFind(graph G)

{

bool cycle = false

For each u

u.color = white

u.parent = Nil

For each u

If (u.color == white)

If (DFS-visit (G, u) == -1)

cycle = true

break

if(cycle)

Print “Yes, there is a cycle”

Else

Print “No cycle”

}

int DFS-visit(G, u)

{

u.color = gray

For each v

If(v.color == gray && v != u.parent)

return -1

If (v.color == white)

v.parent = u

if (DFS-visit(G,v) == -1)

return -1

u.color = black

return 0

}

For a vertex u, if an adjacent vertice v is visited (gray) and v is not the parent of u, then we have found a cycle. A tree (an acyclic graph) has at n-1 edges where n = number of vertices. Therefore, we do not need to examine all edges as if we were doing normal DFS. We only need to examine n edges to determine if there is a cycle. Consequently, the running time for this algorithm in O(n). An alternate algorithm is to simply count the number of edges using the adjacency list and compare the total number of edges to the number of vertices. If we have more than n-1 edges, then the graph will have this cycle. We would need to count at most edges, so this algorithm would have O(n) complexity.

**4.**

We can determine the number of paths between two arbitrary vertices s and t in DAG using topological sort. As shown in class, we can use the DFS algorithm to determine a topological ordering by examining the finish time of each vertice and ordering the vertices from greatest finish time to least finish time. Now, we have list of vertices in topological order that we can use to find the number of paths:

Input: A list of vertices of a DAG in topological order L, a DAG G where each vertice has a numOfPaths field

Output: Number of paths between vertices s and t

Algorithm:

int numPaths(L, G, s, v)

{

For each v

v.numOfPaths = 0

tIndex = L.find(t) //Search L for t vertice

L[tIndex].numOfPaths = 1

Scan L in reverse order starting from L[tIndex-1] until L[i] = s

for each descendent of L[i]

L[i].numOfPaths += x.numOfPaths

return L[s].numOfPaths after calculating numOfPaths for s

}

The number of paths between vertices s and t is the sum of the paths from direct descendants of s to t. In this algorithm, we implement this idea using the numOfPaths field. Because we process the vertices in reverse topological order, we are ensuring that the numOfPath field is calculated for all descendants before processing the parent. For example, if s has two descendants a and b, the reverse topological ordering will force us to examine a and b, before s. Once we examine s, the numOfPath field will already be computed for both a and b. We can then calculate the numOfPaths for s by adding a.numOfPaths + b.numOfPaths.. In this algorithm, we need to initialize all n vertices in the topological list which is O(n). We also need to calculate the numOfPaths of each vertice between s and t. In the worst case, we need to iterate over all edges so the running time is O(m). Therefore, the total running time is O(n+m).

**5.**

The following algorithm gives a topological sort of a DAG:

Input: A DAG G

Output: Linked List containing

Algorithm:

TopologicalSort(G)

{

Linked list L = empty

Queue M = empty

Calculate in-degree for each vertice v

Enqueue all vertices with in-degree 0

While(V(G) != empty)

{

x = M.dequeue

L.addToEnd(x)

For each edge (x,j) , j

E(G) = E(G) – (x,j)

j.indegree—

if(j.degree == 0)

M.enqueue(j)

V(G) = V(G) – x

}

Output L

}

First we must calculate the in-degrees of all vertices, which is O(n+m). Then, we need to initialize the queue by looking for vertices with in-degree 0. This step is O(n). The loop can be in O(n+m) because each vertice will be dequeued and examing the edges of all vertices is the summation of degree which is O(m). Therefore, this algorithm is runs in O(n+m) time.