Jahidul Islam - 171001155 - Programming Assignment 4

Problem 1:

Code:
```
for (int v = 0; v < G.V(); v++)
for (int w : G.adj(v))
StdOut.println(v + "-" + w);
```

a. Undirected Graphs Adjacency Matrix Representation

For an adjacency matrix, we know it keeps a value 1/0 for every pair of nodes or in our case we can call them vertices. So the space required is n*n so the time complexity is O(V^2).

b. Undirected Graphs Adjacency List Representation

For a List on the other hand, it only contains existing edges meaning the length it has is at most the number of edges or the number of nodes. Thus the time complexity is O(V+E).

c. Directed Graphs Adjacency Matrix Representation

Similar to an undirected graph's matrix, a directed graph's adjacency matrix will have the time complexity O(V^2)

d. Directed Graphs Adjacency List Representation

Again the adjacency list representation will have the time complexity O(V+E)

Problem 2:

a)

v	0	1	2	3	4	5	6
Marked[ ]	T	T	T	T	T	T	F
edgeTo[ ]	-	4	0	2	0	0	-

b)

v	0	1	2	3	4	5	6
edgeTo[ ]	-	4	0	2	0	0	-
distTo[ ]	0	2	1	2	2	1	-

Problem 3:

a)

v	0	1	2	3	4	5	6
Marked[ ]	T	F	T	T	T	T	F
edgeTo[ ]	-	-	0	2	0	0	-

b)

v	0	1	2	3	4	5	6
edgeTo[ ]	-	-	0	2	0	0	-
distTo[ ]	0	-	1	2	1	1	-

Problem 4:

A.

The time complexity is O(V+E) because we are visiting each node once and every edge is considered twice so the time complexity is a multiple of those two always. Thus the time complexity is written as O(V+E). Below is the pseudocode.

1) First create a stack S
2) Then mark v as visited and push v onto the stack S
3) while S is not empty
    a) Proceed to peek at the top u of S <- loop
        i) if u has an unvisited neighbour we can call w then:
            1)mark w and push it onto the stack S
            2)else pop S

B.

Since the time to process a vertex is proportional to the length of its adjacency list and again we are visiting each node once the time, we use the same thought process of part A to attain the time complexity for BFS which is O(V+E). Below is the pseudocode:

1) First create a queue Q
2) Proceed to mark v as visited and put v into the queue Q
3) while Q is not empty
    a) remove the head which we can call x of Q <-loop
    b) mark and enqueue all the unvisited neighbours of x <- loop

Problem 5)

Solved with BFS

1)create a color array to store colors
2)assign all vertices with v
3)Give value of -1 to the color array meaning no color and 1 is first color and 0 is second color
4)Assign first color to source
5)Create a queue of v's and enqueue source v
6)While there are v's in queue <-loop
    a)dequeue a V from q
        i)if there is a self loop
            1)return false
        ii)for all non colored adjacent vertices <-loop
            1)if an edge from u to v exists and v is not colored
                a)assign the other color to this adjacent v of u
            2)else if v is colored the same as u
                a)return false
    b)now all adjacent vertices are the alternate color so return true

We can see in tabular form that all points have multiple connections so we have to choose the points to color where they will not have an adjacent or connected point with the same color. So we can select 0,3, amd 4 to be red as they are not adjacent but make up the points that connect to all others.

0	1
0	2
0	5

0	6
1	**3**
2	**3**
2	**4**
4	5
4	6

Problem 6)

We will use recursive function and parent to detect a cycle from vertex v

1)Set up a class graph with in v and a pointer to array with adjacent lists
2)In a function to find cycle mark the current node as visited
3)for all vertices adjacent to this selected vertex <-loop
    a)if an adjacent is not visited recur for that adjacent
        i)return true and recur
    b)else if an adjacent is visited and it is not the parent of current vertex, there is a cycle
        i)return true that there is a cycle
4)else return false for the "for"

If we take a look at the table below we can see that in order to maintain a cycle that returns to the original we can have a cycle such as 0-5-4-6-0 because it starts and ends in zero while being able to follow the given graph's restrictions.

0	1
0	2
0	5
0	6
1	3
2	3
2	4
4	5
4	6

Problem 7:

1)First, create a class graph with int V and a pointer to an array containing adjacency list

2)Create a function to detect a cycle

a)if the vertex is not visited or equals false

i)Mark the current node as visited and also make it a part of recursion stack

ii)use a for to recur all the vertices that are adjacent to this currently

b)otherwise remove the vertex from recursion stack

3)Another function that returns true if the graph confirms a cycle

a)First mark all vertices as not visited and not part of recursion stack

b)call the previous function to detect cycles and return true if cycle

c)otherwise return false

This has the same time complexity of DFS since that is what it uses which means it is O(V+E).
For the data we will separate the three circled zones to keep data easy to understand.

Red Circle(0,1,2)

v	0	1	2
Marked[ ]	T	T	T
edgeTo[ ]	2	0	1

Red Circle(3,6)

v	3	6
Marked[ ]	T	T
edgeTo[ ]	6	3

Red Circle(7,5,4)

v	4	5	7
Marked[ ]	T	T	T
edgeTo[ ]	5	4	,75