Data Structures
Theoretical Homework 5

Abraham Murciano

January 18, 2020

1. (a) i. The graph shown in figure 1 is connected and bipartite.

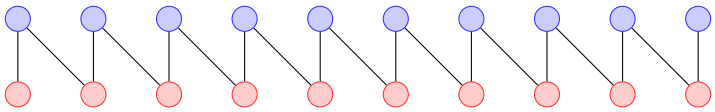

Figure 1: Connected bipartite graph

ii. The graph in figure 2 is disconnected as well as bipartite. In fact any graph with no edges is bipartite.

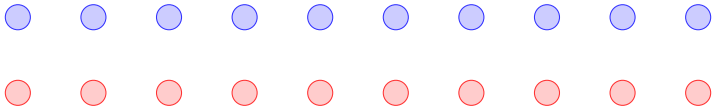

Figure 2: Disconnected bipartite graph

(b) This is an algorithm which will tell us if the graph $G = (V, E)$ is bipartite or not. It attempts to colour each connected component with only two colours, and if it fails, then G is not bipartite.

```
1:  function BIPARTITE(graph G(V, E))
2:      for all v ∈ V do                                   ▷ Mark each vertex as not visited
3:          v.colour = NULL
4:      for all v ∈ V do
5:          if v.colour == NULL then                       ▷ v is part of an unvisited connected component
6:              if ColourConnectedComponent(v) == false then
7:                  return false                           ▷ We were unable to colour this component with two colours
8:      return true
9:
10: function COLOURCONNECTEDCOMPONENT(vertex v, colour c = BLUE)
11:     v.colour = c
12:     if c == BLUE then
13:         OppositeColour = RED
14:     else
15:         OppositeColour = BLUE
16:     for all u adjacent to v do
17:         if u.colour == v.colour then                   ▷ If a vertex is connected to another of the same colour,
18:             return false                               ▷ then it is not bipartite
19:         else if u.colour == NULL then
20:             ColourConnectedComponent(u, OppositeColour)   ▷ Recursively colour all v's neigbours
21:     return true
```

2. (a) In order to aid the upcoming definitions, we will define a **path** from v to u as a sequence of edges P whose first edge is of the form $\{v, x\}$ and its last edge is of the form $\{y, u\}$, and each other edge in P shares one of its vertices with the previous edge in the sequence and shares its other vertex with the next edge in the sequence.

i. A **directed graph** $G(V, E)$ is a graph whose set of edges E is a set of *ordered* pairs of vertices.

ii. A **connected graph** $G(V, E)$ is a graph such that for all two vertices $v, u \in V$, there exists a path from v to u.

iii. An **acyclic graph** $G(V, E)$ is a graph such that for all adjacent vertices $v, u \in V$, all paths from v to u contain the edge $\{v, u\}$.

(b)
```
1: function MaximalOutDegree(graph G(V, E))
2:     HighestOutDegree = 0
3:     for i = 0 to |V| − 1 do
4:         OutDegree = 0
5:         for j = 0 to |V| − 1 do
6:             OutDegree += Adjacent[i][j]      ▷ If the i^th and j^th vertices are adjacent, increment OutDegree
7:         if OutDegree > HighestOutDegree then
8:             HighestOutDegree = OutDegree
9:     return HighestOutDegree
```

The complexity of this function is $\Theta(|V|^2)$, since we must check every single possible edge in the adjacency matrix, and there are $|V|^2$ elements in the matrix.

3. In figure 3 below, if a Depth First Search (DFS) algorithm was used to find cycles, it may occur that the algorithm would begin at node 0, then then reach node 1, and at that point it may choose to either continue to node 2 or to node 6. If the algorithm decided to take the path to node 2, it would continue onwards until reaching node 5, and only afterwards backtrack to node 1 in order to continue on the path to 6, after which it would again reach node 0 and proclaim the graph cyclic.

   However if a Breadth First Search (BFS) algorithm was employed, and it too began at node 0, the first two nodes it would find would be nodes 1 and 6. Then at the very next step it would find node 2 as well as the already visited nodes 1 and 6, therefore immediately concluding that the graph is cyclic.

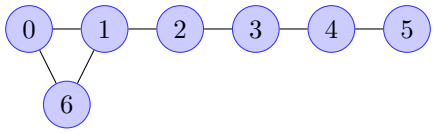

Figure 3: graph where BFS discovers a cycle before DFS

4. The Following algorithm will return the number of shortest paths from a source node s to a destination node d. It extends the Breadth First Search algorithm and achieves a run time complexity of $\Theta(|V| + |E|)$.

```
1: function CountShortestPaths(graph G(V, E), vertex s, vertex d)
2:     for all v ∈ V do
3:         v.distance = ∞
4:         v.count = 0
5:     s.distance = 0
6:     s.count = 1
7:     Queue = φ
8:     Queue ← s
9:     while Queue ≠ φ do
10:        v ← Queue
11:        if v.distance ≥ d.distance then
12:            break
13:        for all u adjacent to v do
14:            if u.distance == ∞ then
15:                u.distance = v.distance + 1
16:                u.count = v.count
17:                Queue ← u
18:            else if u.distance == v.distance + 1 then
19:                u.count += v.count
20:    return d.count
```

5. Given a set of N students and a set of pairs, where a pair (i, j) means that student j copied the homework solution from student i:

(a) To find the set of students who did not copy from anyone, a graph can be built using an adjacency list. Each student would be represented by a vertex, and each edge a, b would indicate that student a copied from student b. Once this graph is built, the students who did noy copy can be found as follows. The complexity of this algorithm is $\Theta(|N|)$.

```
1: set S                                          ▷ S will store all the students who did not copy
2: for all v ∈ V do
3:     if edges[v] is empty then
4:         Insert v into S
```

(b) To find the set of students whom nobody copied from, a graph can be built similarly to part (a), with one difference. Each edge a, b would indicate that student b copied from student a. The algorithm and complexity would be exactly the same.

(c) To find the set of students who copied from a specific student, we can build a graph using an adjacency list, where each edge (a, b) represents that student b copied from student a, we can use the following algorithm. The complexity is $\Theta(|N|)$.

```
1: function StudentsWhoCopiedFrom(student s)
2:     mark s as visited
3:     for all v adjacent to s do
4:         if v was not visited then
5:             Result.insert(v)
6:             Result.union(StudentsWhoCopiedFrom(v))
```

(d) The question is not very clear. If my understanding of it is correct, then we would have to find the largest strongly connected component, complexity of which is $\Theta(N + |E|)$

6. (a) Run the Strongly Connected Component algorithm on the graph to find am acyclic component graph, then go through each vertex of the acyclic component graph, and check the in degree of that vertex. If there is exactly one vertex with an in degree of 0, then return any vertex from within that vertex. Otherwise, there is no root.

(b) If there is exactly one vertex in the acyclic component graph with an in degree of zero, then all the other vertices can be reached. Since the graph is acyclic, then they can all be reached from the vertex with the in degree of zero. Therefore any vertex within the strongly connected component of that vertex is a valid root.

However if there is more than one vertex with an in degree of zero, then each of those vertices is unreachable from the others, hence none of them can be a root.

7.

Vertex	Step 1		Step 2		Step 3	
	d	π	d	π	d	π
A	0	NULL	0	NULL		
B	∞	NULL	7	A		
C	∞	NULL	8	A		
D	∞	NULL	0	NULL	12	B
E	∞	NULL	0	NULL	9	B
F	∞	NULL	11	A		