NAMES: **Suchita Patel,bingcong li, Dakota Krogmeier**

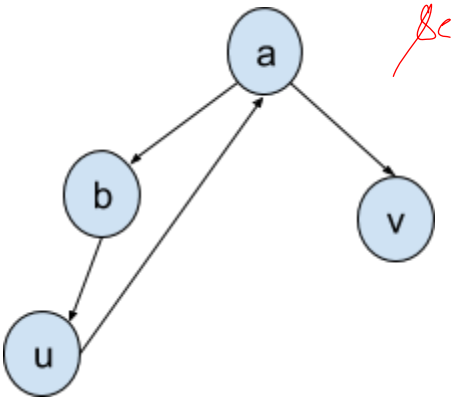
Worksheet 19: Work with the other students in your Zoom breakout room to complete this worksheet. You should only submit one paper for the group.

1. Either prove or give a counter-example (by drawing a relevant directed graph and explaining why your graph is relevant by clearly showing that the hypothesis is met but the conclusion fails to hold): if there is a path from u to v in a **directed** graph G, and if d[u] < d[v] in a **breadth-first search** of G, then v is a descendant of u in the breadth-first forest produced. (2 pts)

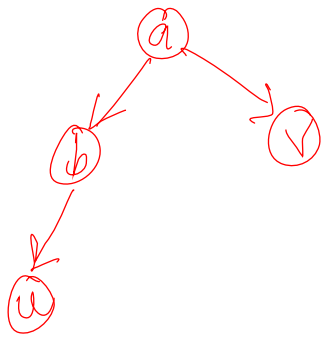
Consider the following graph. There is a path from u to v. If we run BFS from vertex a. Consider b as the first neighbor of a during BFS, and v as a neighbor of a. So, we would have start u.start =5 , u.finish =6 , v.start =3 , and v.finish =4 .

[start, finish] for each vertex ⇒ a [1,8], b[2,7], v[3, 4], u[5,6]

Here d[u] = 5 is not less than d[v] = 3. Hence, v is not a descendant of u in the produced breadth first forest.



See comment below.



Provide BFS tree

Explain with respect to distance of u/v from root node

(-1)

d[u] = distance of u from root node (See algorithm)

2. Describe an algorithm that determines if an **undirected** graph G = (V, E) contains a cycle. Your algorithm should take, as input, a graph (you should specify how the graph is represented), and returns True if the graph contains a cycle, and False if it does not. (4 points)

KEY IDEA:

The goal is to use a helper function that will use recursion to go over all the nodes and if one node is visited twice then it says its a cycle. This helper function will be used in the main function to test the cycle where it will return true or false depending on the helper functions findings. We start by marking all nodes as unvisited, then after choosing a starting vertex, we'll mark that node as being visited. After that, we will do recursive checks to all the adjacent nodes, if a node has not been visited, we will repeat the same recursive checks to all nodes that are adjacent. If we come across a node that has already been visited and isn't the parent node, then we can mark it as a cycle. The main function will then iterate through cycleFound to return true or false based on if there is a cycle.

Algorithm HasCycle(G):

Input: Undirected graph G of vertices V and edges E, G = (V, E)

Output: True if graph G has a cycle, false otherwise

PROCESS:

```
HasCycle(G):
    visited <- []
    cycleFound <- []
    for i <- 0 to V:
        if helper(i, visited, cycleFound) = true:
```

```
                return true
            else
                return false
        return false


helper(i, visited, cycleFound):
        if cycleFound[i] = true:
            return true
        if visited[i] = true:
            return false
        visited[i] <- true
        cycleFound[i] <- true
        children <- adjacent[i]  #all adjacent vertices of i

        for c in children:
            if helper(c, visited, cycleFound)  = true:
                return true
            else
                return false
        cycleFound[i] <- false
        return false
```

if c is not visited:

elif check if c != parent[i]

(-0.5) Keep track of parent also