These have been collected from books, other courses, and actual interview questions. Any starred question will not be asked in the exam. Unless otherwise stated, assume a graph G is simple, undirected, and unweighted. Assume the graph is represented as an adjacency list. For any question, give a clear algorithm as pseudocode, and give a running time analysis.

1. A cycle is a sequence of vertices v1, v2, . . . , vk such that for each i, (vi , vi+1) is an edge, and (vk, v0) is an edge. Determine if G has a cycle. Try using both BFS and DFS.

First begin with calling DFS(G) on any start that hasn't been visited yet

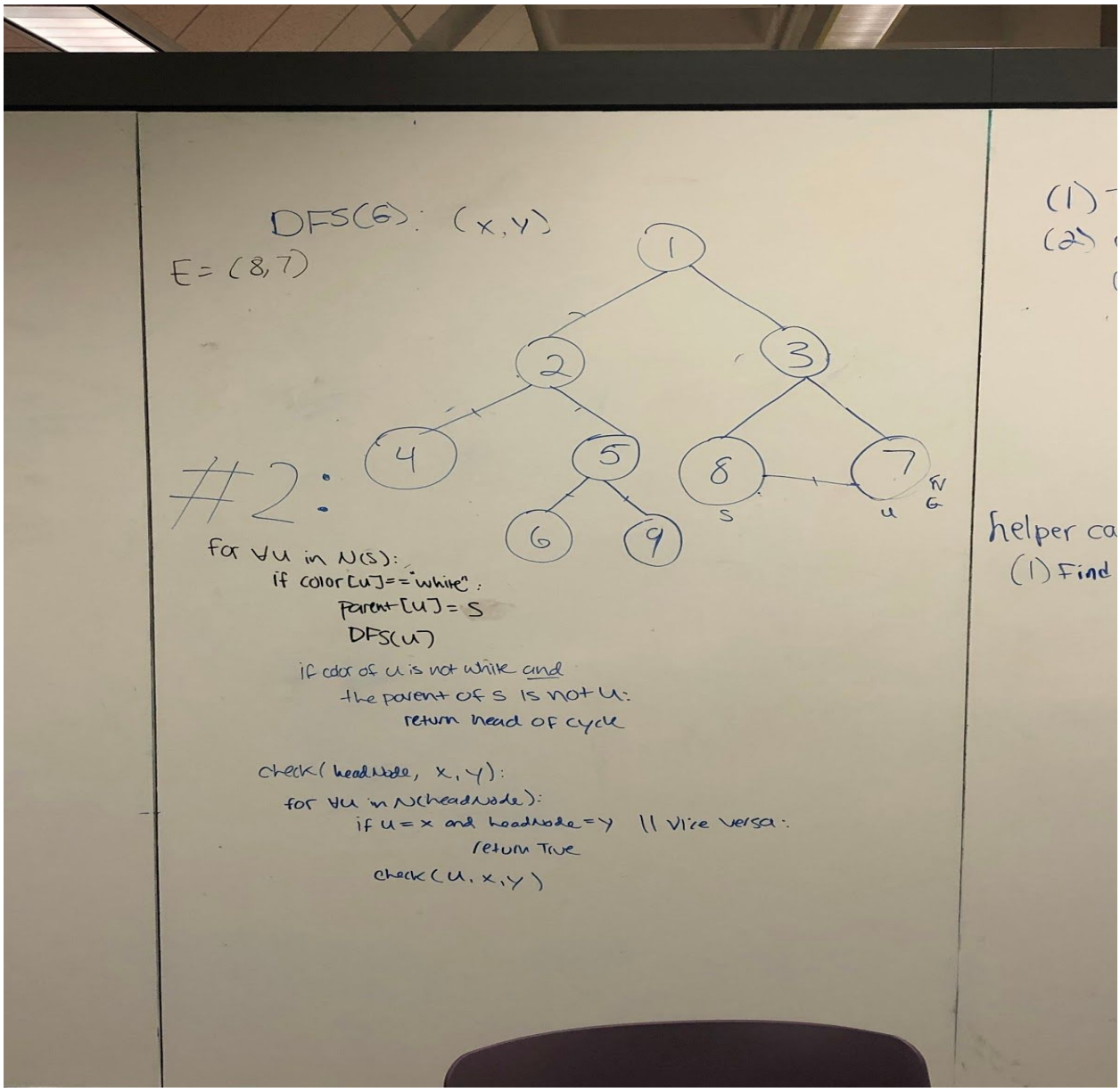
(1) Initialize arrays
(2) for all s ∈ V:
    (a) If visited[s] is false
        (i) If DFS(s) is true :
            1) Return true

After this follow the steps that go with DFS(s)

(1) visited[s] = true
(2) for all u ∈ N(s) :
    (a) If visited[u] is false :
        (i) pred[u] = s
        (ii) if DFS(u) is true:
            1) return true
    (b) else if u is not equal to pred[s] :
        (i) return true
(3) return false

We use a DFS(G) to solve this problem. The difference in this case is that we do not need time, discovered, or finished. Also this function will be returning a boolean; true is there is a cycle and returns false if not. The traversal of the graph is the same as a normal DFS although we check if an already visited vertex is equal to the parent of the start. If a graph does not have a cycle the already visited vertex will be equal to the start's parent. Otherwise if it does not equal the parent then there is a cycle in the graph. Given that this is a normal DFS with boolean statements it is possible we will search every vertex and edge if there is no cycle. Also it is possible that there is a cycle at the end of the graph. Given this information the given operation has a runtime of O(V+E)/ O(m + n).

2. Given edge e, determine if G has a cycle involving e.



"White" = not checked yet

For all that's in u in n(s), we basically want to check to see if the neighbors have been visited (hence the color reference)

If the color [u] == "white", that means it hasn't been visited yet in the graph

Once a node has been visited, it turns "black"

(you can come up with different references that might help)

As the graph shows, DFS is being used to find a cycle

The cycle in the graph is 3, 7, 8

If the parent node has a child and that child has a neighbor, then it is considered cycle

3. . Determine if G contains any cycle. (Clearly, you can use the solution for the previous problem, but there's a more efficient method.)

First begin with calling DFS(G) on any start that hasn't been visited yet

(3) Initialize arrays
(4) for all s ∈ V:
    (a) If visited[s] is false
        (i) If DFS(s) is true :
            1) Return true

After this follow the steps that go with DFS(s)

(4) visited[s] = true
(5) for all u ∈ N(s) :
    (a) If visited[u] is false :
        (i) pred[u] = s
        (ii) if DFS(u) is true:
            1) return true
    (b) else if u is not equal to pred[s] :
        (i) return true
(6) return false

We use a DFS(G) to solve this problem. The difference in this case is that we do not need time, discovered, or finished. Also this function will be returning a boolean; true is there is a cycle and returns false if not. The traversal of the graph is the same as a normal DFS although we check if an already visited vertex is equal to the parent of the start. If a graph does not have a cycle the already visited vertex will be equal to the start's parent. Otherwise if it does not equal the parent then there is a cycle in the graph. Given that this is a normal DFS with boolean statements it is possible we will search every vertex and edge if there is no cycle. Also it is possible that there is a cycle at the end of the graph. Given this information the given operation has a runtime of O(V+E).

4. A directed graph is called a DAG if it contains no directed cycles. Determine if a graph is a DAG.

It is a graph that is directed and without cycles connecting the other edges.

It's not possible to traverse the entire graph starting at one edge and the edges of the directed graph only go one way.

A graph is a series of vertices connected by edges. Within a directed graph the edges are connected in a way each edge only goes one way.

A directed acyclic graph is not cyclic, or it is not possible to start at one point in the graph and traverse the entire graph.

Each edge is directed from an earlier edge to a later edge. This is also known as a topological ordering of a graph.

When a graph has at least one loop is called cyclic, and if it doesn't have is called acyclic. Acyclic directed graphs are also called DAGs.

When a graph is acyclic it must have at least one node with no targets called a leaf.

DFS(G)

1. Initialize boolean arrays to false. An array named visited and the other named backEdge
2. for all s ∈ V:
    a. If DFS(s) is false:
        i. Return false
3. Return true

DFS(s):

1. If visited[s] is false :
    a. visited[s] = ture
    b. backEdge[s] = true
    c. for all v ∈ N(s) :
        i. If (visited[v] is false and DFS(v) is false) :
            1. return false
        ii. else if (backEdge[v] is true) :
            1. return false
2. backEdge[s] = false
3. Return true

A graph is a DAG when it is directed and Acyclic. The way to figure out if a graph has this form is to check if any back edges exist in the directed graph. A back edge exists if a vertex is still within the stack from a DFS operation. We can have a makeshift stack by having a backEdge boolean array to see if it is still there. Given that this is a DFS implementation the running time is O(m + n) where m is the number of vertices and n is the number of edges.

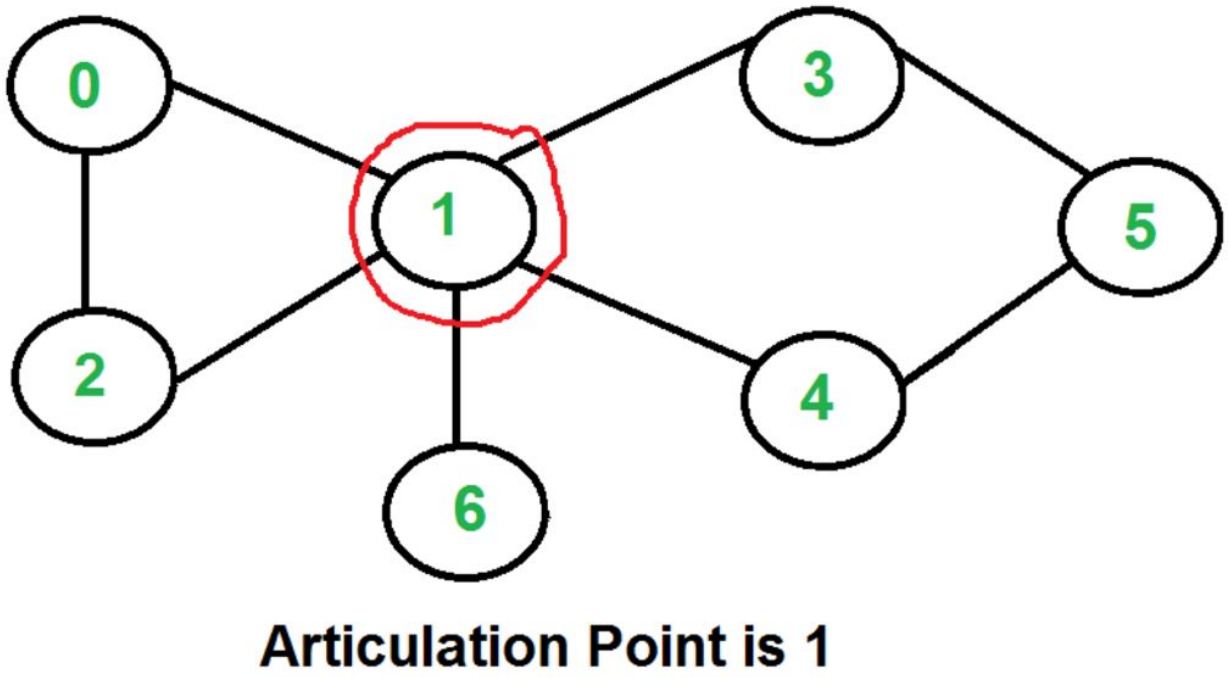
5. ∗ Prove that a graph G is bipartite iff it has no cycles of odd length. (Hint: follow the progress of BFS on G)

6. A vertex v is a cut or articulation vertex if its removal disconnects two other vertices. Equivalently, all paths (in G, before removal) from u to u 0 pass through v. Determine if v is a cut vertex. If so, find some pair (u, u0) that got disconnected.

Cut Vertex in an Undirected Graph:- A vertex in an undirected connected graph is an **articulation point** (or **cut vertex**) iff removing it (and edges through it) disconnects the graph. It can be thought of as a single point of failure. Any graph that contains an articulation point is inherently fragile because deleting that single vertex causes a loss of connectivity between other nodes. If a graph does not have an articulation point, then it is said to be biconnected.

So, in case of an undirected Graph, if the removal of a vertex disconnects two other vertices and also disrupts all paths in G before removal from u to u' via v, then v is said to be the Cut Vetex / Articulation Point in the Graph G.

In the below attached example Vertex = 1 is the Cut Vertex because it disconnects Vertex 0 and Vertex 4 from the Graph. So (u,u') = (0, 4) and v = 1 [Cut Vertex].



Articulation Point is 1

Let us now describe the Algorithm for Finding out the Cut Vertex in a given Grapg G:-

The algorithm utilizes the properties of the DFS tree. In a DFS tree, a vertex u is parent of another vertex v if v is discovered by u. A vertex u is articulation point iff one of the following two conditions is true:

- u is the root of the DFS tree and has at least two children
- u is not the root and no vertex in the subtree rooted at one of the children of u has a back edge to an ancestor of u

Let **disc_time[u]** be the time at which a vertex u was discovered/explored during the dfs traversal. Let**low[u]** be the earliest discovery time of any vertex in the subtree rooted at u or connected to a vertex in that subtree by a back edge. Then

- If some child x of u has low[x] ? disc_time[u], then u is an articulation point.
- low[u] = min({low[v] | v is a child of u} ? {disc_time[x] | (u, x) is a back edge from u})

The Pseudocode is as follows:-

```
Let disc_time[v] = -1 and explored[v] = false forall v

dfscounter = 0;   // Intialize dfscounter to 0

DFS(v):     // The DFS Algorithm starts from here
        explored[v] = true
        disc_time[v] = low[v] = ++dfscounter
        foreach edge (v,x):
                if !explored[x]:
                        DFS(x)
                        low[v] = min(low[v], low[x])
                        if low[x] >= disc_time[v]:
                                print "v is articulation point separating x"
                elif x is not v's parent:
                        low[v] = min(low[v], disc_time[x]);

        end foor
end DFS.
```

7. The proof of Dijkstra's algorithm fails when there are negative edges. Why?

Infinite loop when we do this. Because each time we go back and check for less than it will always have a less than with negative.

8. We have three containers, whose sizes are 10 pints, 7 pints, and 4 pints. The 7-pint and 4-pint containers are full of water, while the 10-pint container is empty. We are only allowed one operation: pouring the contents on one container into another, stopping only when the source container is empty, or the destination container is empty. Determine if there is a sequence of pourings that leave exactly 2 pints in the 7- or 4- pint container. Model this as a graph problem. Trying solving the general version where there are three containers of integer capacities of a maximum of n. The starting configuration is provided, and we have some desired "end state". Determine if the "end state" can be achieved. Work out the running time.

Let G = (V,E) be a directed graph.

It will model the set of nodes as triples of numbers (a_0, a_1, a_2) where the following relationships hold:

Let $S_0 = 10, S_1 = 7, S_2 = 4$ be the sizes of the corresponding containers.
a_i will correspond at the actual contents of i^{th} container. It must hold $0 \leq a_i \leq S_i \; for \; i = 0, 1, 2$ and at any given node (a_0, a_1, a_2) and (b_0, b_1, b_2) exists if two of the following are satisfied:

Two nodes differ in exactly two coordinates (and the 3rd one is the same in both).
If i, j are coordinates they differ in, then either a_i = 0 or $a_j = 0$ or $a_i = S_i$ or $a_j = S_j$.

Are there paths between the nodes (0,7,4) and (*, 2,*) or (*,*,2) where * stands for any (allowed) value of the corresponding coordinate?

DFS algorithm on that graph should be applied, starting from node (0,7,4) with an extra line of code that halts and answers 'YES' if one of the desired nodes is reached 'NO' if all the connected component of the starting node is exhausted and no desired vertex is reached.

After a few steps of the algorithm (depth 6 on dfs tree) the node (2,7,2) is reached, so the answer is 'YES'.

9. Given a directed (unweighted) graph G, the reverse is obtained by simply reversing all edges. Assume G is represented as an adjacency list of out neighbors. Construct the reverse of G.

We are given a directed graph G represented as an adjacency list. Given this we know an adjacency list can be represented in the form of a matrix. For example, say this is graph G:
Ⓐ→Ⓑ
↓ ↑
ⓒ→Ⓓ

Given this directed graph the matrix would look like:

	A	B	C	D
A	0	1	1	0
B	0	0	0	0
C	0	0	0	1
D	0	1	0	0

To get the reverse of the matrix (or adjacency list) we would find the transpose of the matrix. The matrix would now be:

	A	B	C	D
A	0	0	0	0
B	1	0	0	1
C	1	0	0	0
D	0	0	1	0

Given the above matrix we now can see what the new graph looks like

Ⓐ←Ⓑ
↑ ↓
ⓒ←Ⓓ

This is the reverse of the graph because all edges have been reversed. Matching with the transpose of the matrix we see this finds the reverse of the graph G.

10. ∗ An Euler tour is a cycle that can pass through each vertex multiple times, but must use each edge exactly once. Prove that an undirected graph has an Euler tour iff all vertices have even degree.

11. Using the previous theorem, give an algorithm that finds an Euler tour (if one exists).

eulerianCheck() :

1. for all s ∈ V : -------> O(m)
    a. If N(s) != 0 :
        i. set visited array to false
        ii. DFS(s) ----------> O(m + n)
        iii. For all s ∈ V: ----------> O(m)
            1. If (visited[s] is true and N(s) is odd) :
                a. continue
        iv. print(“This graph contains a euler tour”)
        v. return

In this code we check to see if a Euler tour exists within a graph. In line 1 we loop through every vertex of the array. If the degree is not 0 DFS(s) is called. Before this the visited array is set to be all false so we only see true for the given neighborhood. After DFS(s) is done and the visited array is true for the given neighborhood we check if each visited vertex has an odd degree (odd number of neighbors). If so then this is not a euler cycle and we go back up to check another vertex neighborhood. However if the loop goes through and all the visited vertices are even then we have found a Euler cycle within the graph. Given that the first loop runs the amount of

vertices we will call m. After this we call DFS(s) which has the possibility of being connected to all edges and vertices we will call the edges n. Inside the first loop is (m + n) + (m) given (m + n) is larger it absorbs the runtime of (m) and the total runtime becomes O(m(m + n)).

12. . A common recommendation feature used in social networking websites is the number of common friends. In an undirected graph G, find a pair (u, v) with the largest number of common neighbors.

An undirected graph G.

To find: A pair of vertices (u, v) in G with largest number of common adjacent vertices(neighbors).

Algorithm:

- Store the vertices of the graph G in the variable n.
- Initialize the variable max_count, i, j and k to 0.
- Take the one-dimensional array named visited having size n and let another array be adj [n][n] which is 2-dimensional.
- For each of the vertices till n, mark each entry of the array visited to be 0.
- Create a nested loop, which means for each value of i from 0 to n and for each value of j from 0 to n, if there exist a neighboring vertex which is not visited that is, adj[i][j] = 1 and visited[j]=0, then increment the variable count.
- Again, loop for each of the value of the vertices and increment the count.
- If the value of count is greater than max_count then assign the value of the iterator i to the vertex u and the value of the iterator j to vertex v.
- Set the variable count to 0 and visited[j] to 1.
- The value of the vertices (u, v) is obtained through the iterators.

13. ∗ A directed graph G is singly connected if for any pair u, v of vertices that are connected (from u to v), there exists at most one simple path from u to v. Determine if a directed graph is singly connected. (Hint: DFS)

14. Consider a game of snakes and ladders. Determine the minimum number of throws required to win the game.

BFS(s) :

1. s.position = 0, s.rolls = 0
2. Q is a queue initialized with s
3. visited is an array initialized to false
4. move array where index is the bottom of ladder or mouth of the snake and it's value is the vertex the character will end up. Value is -1 if there is no ladder or snake
5. visited[s.position] = true
6. while Q is not empty :
    a. u = top(Q)
    b. If (u.position = size of move - 1) :
        i. return u.rolls
    c. dequeue(Q)
    d. for all v ∈ N(u) :
        i. If (visited[v.position] is false):
            1. v.rolls = u.rolls + 1
            2. visited[v.position] = true
            3. If (move[v.position] != -1) :
                a. v.position = move[v.position]
            4. enqueue(v) into Q

Assuming the graph's vertices and move array are the same size as the number of squares on the board. Each vertex has a pair of values, one is their position on the board, the other is the amount of rolls it took to get to its location. Assuming vertex's edges are connected to the next 6 vertices over to simulate a dice roll. This will be a BFS(s) operation because we know the entire board is connected. If the vertex is at the bottom of a ladder or mouth of a snake. The vertex will be moved to the vertex of the move's destination. This is the vertex that will then be enqueued. The first vertex to reach the end will have the shortest path, and we return the number of rolls it took to get there. Given that this is a BFS every vertex and edge can be visited in the worst case. The runtime is O(V + E) where V is the number of vertices and E is the number of edges.

15. Biologists often construct a food network of species in an ecosystem. The vertices represent species, and a directed edge (u, v) means species u eats species v. An apex species is one that is not eaten by another species. Suppose you are give a list of all possible "eating" relationships (so a list of "u eats v"). Determine all apex species.

apexFinder(G):

1. Initialize an array in with size of vertices to 0
2. for all u ∈ V :
    a. for all edges(u, v) ∈ E :
        i. in[v] = in[v] + 1;

3. for all u ∈ V:
    a. If in[u] is 0:
        i. print(u " is an apex predator")

In this code we create an array that keeps track of how many times a given vertex is pointed at. Given that this is a directed graph we know that the vertex receiving an in-neighbor is being eaten and it's out neighbor is who the vertex is eating. If the vertex has any in-neighbors it is not being eaten. Therefore after all vertices are checked we run a loop of all vertices and anything with 0 in-neighbors is an apex predator.

16. In a food network (described above), determine the effect of the extinction of a species. If a species v goes extinct, and there is some other species u that only eats v, then u will become extinct. This can cascade through the food network. Given a food network and a species v, determine all other species that will become extinct if v goes extinct.

```
Find_Ext(species v){

Danger_list +={All species eating species v}

foreach(Danger_list)

If(Selected species eats other species than species v)

{

remove from Danger_list

}

foreach(Danger_list)

{

Find(Selected species, food network) //recursion

}

}
```

The above pseudocode populates a list with Species that are directly or indirectly dependent on species v.

The running time will be n*n where n is total no of species in food network.

17. A triangle is a set of edges {(u, v),(v, w),(w, u)}. (Equivalently, a triangle is a cycle of length 3.) Given a vertex s, we wish to count the number of distinct triangles in G that involve s. Get a solution with O(n) space.

To-do: Count number of three vertex pairs which loop.
Probably use a combo of BFS and DFS? BFS to get one out neighbor on same level, then use DFS with some kind of counter which returns false if it doesn't loop in 2 steps (if vertex doesn't form a triangle). ?? I'm not sure, but i feel like that's a correct approach

18. . You are given a directed, unweighted graph G representing the power grid, as an adjacency list. Each vertex is a power station, and an edge (u, v) means that power goes from u to v. A source vertex denotes a power generator. A sink vertex denotes a power supplier, typically to a neighborhood. Your job is to understand the robustness of the power grid, under failure of a power station s. (Note that this may not be a source, it could be some internal vertex.) If s fails, it stops transmitting power to any of the out neighbors. This could potentially result in some sink node t not receiving power at all, representing a power failure in a neighborhood. Thus, a sink node t will not receive power if every path from every source to t passes through the power station s. Design an algorithm that given G and a vertex s, determines all the sink vertices t (if any) that stop receiving power when s fails.

This problem can be solved with the help of BFS traversal. The steps are as follows:

1. Remove all the edges from 's' that are in the adjacency list.

2. Perform BFS

3. Now all those nodes which are unvisited are affected i.e. they will stop receiving power.

BFS(source):

Set Discovered [source] = true and Discovered [v] = false for all other v

Initialize L[0] to consist of the single element source

Set the layer counter i = 0

Set the current BFS tree T = ∅

While L[i] is not empty

Initialize an empty list L[i + 1]

For each node u ∈ L[i] except 's'

Consider each edge (u, v) incident to u

If Discovered [v] = false then

Set Discovered [v] = true

Add edge (u, v) to the tree

Add v to the list L[i + 1]

Endif

Endfor

Increment the layer counter i by one

Endwhile

for every a ∈ Discovered[i]

if a is unvisited

add a to affected[ ]

print a is affected

19. A directed graph G is semiconnected if for all pairs of vertices u, v, there is either a path from u to v, or from v to u (but not both). Give an algorithm to determine if G is semiconnected.

Trivial 0(v^3) solution could be to use all-to-all shortest path, but that's

20. The diameter of a graph G is the largest shortest path distance in G (meaning, it is $\max_{u,v \in V} dist(u, v)$, where dist(u, v) is the shortest path distance). Give an algorithm to compute the diameter of a undirected graph G. Give a more efficient algorithm to determine the diameter of an undirected tree T.

BFS(s):

1. distance array size of all vertices set to -1
2. Initialize a queue Q for vertices
3. enqueue(Q) with s
4. Distance[s] = 0
5. while Q is not empty:
    a. u = dequeue(Q)
    b. for all v ∈ N(u) :
        i. if (distance[v] == -1) :
            1. Enqueue Q with v
            2. Distance[v] = distance[u] + 1
6. Initialize a maxDistance to 0
7. for all s ∈ V :
    a. if (distance[s] > maxDistance) :
        i. maxDistance = distance[s]
        ii. toReturn = s
8. return as a pair (toReturn, maxDistance)

getDiameter():

1. Initialize two pairs of ints p1 and p2 where first is the vertex and second is the distance
2. p1 = BFS(any first vertex)
3. p2 = BFS(p1.first)
4. print("Diameter is", p2.second)

To find the diameter of a graph first we will find the longest path using BFS starting from any given vertex. After this is done and the vertex that represents the longest path is found we then call a BFS from this vertex. After this is done the next longest path from the start is found, return a pair with the distance size, this second call will contain the value of the diameter. Given that this is a BFS call each vertex and edge will be checked. The runtime is O(V + E)