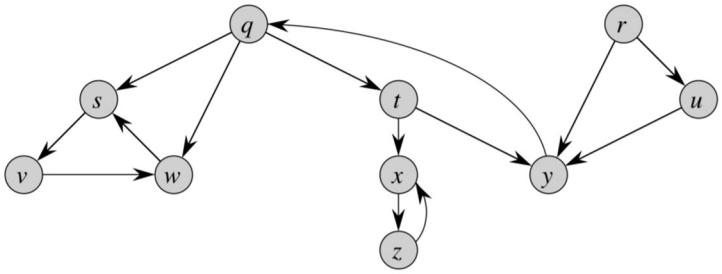
1. (10 points) Create the adjacency list for the graph above. Create the adjacency matrix for the subgraph composed of nodes v, s, w, q and t. If I had asked you to create the adjacency matrix for the entire graph, how many entries would it have?



Adjacency list

Q	S->T->W
R	->u->y
S	->V
T	->X->Y
U	->Y
V	->W
W	->S
X	->Z
Y	->Q
Z	->X

Adjacency Matrix

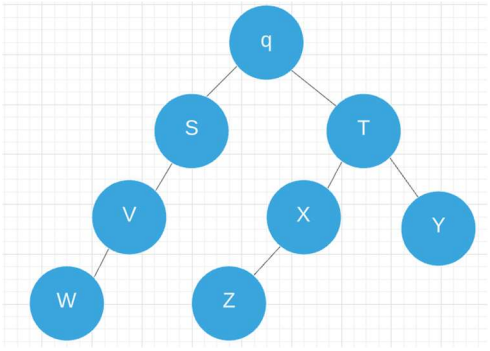
	Q	S	T	V	W
Q	0	1	1	0	1
S	0	0	0	1	0
T	0	0	0	0	0
V	0	0	0	0	1
W	0	1	0	0	0

If we made an adjacency matrix for the entire graph, the size of the entries would be $O(V^2) = O(10^2) = 100$

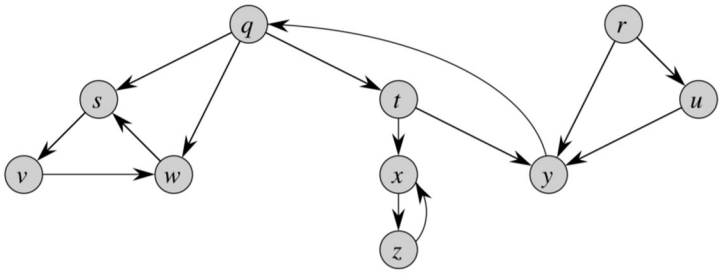
2. (10 points) Read about Breadth-first trees on the bottom of page 600. Then show the d (distance) and π (predecessor) values that result from running breadth-first search on the directed graph above when starting at vertex q.

Draw the predecessor subgraph.

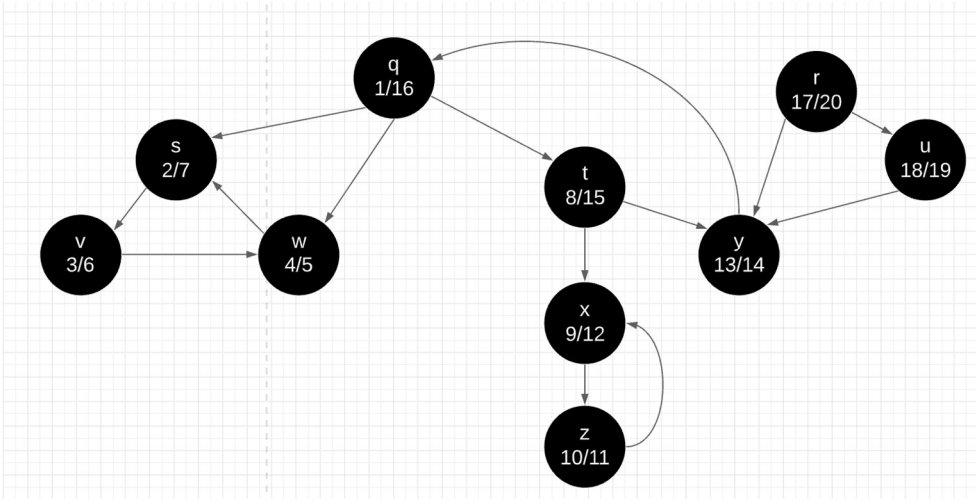
	q	r	s	t	u	V	w	x	y	z
d	0	∞	1	1	∞	2	1	2	2	3
π	NIL	NIL	q	q	NIL	s	q	t	t	x



3. (10 points) Show how depth-first search works on the graph of above. Assume that the for loop of lines 5−7 of the DFS procedure considers the vertices in alphabetical order, and assume that each adjacency list is ordered alphabetically. Show the discovery and finishing times for each vertex.



- We start by checking q
    - We find s first so we move into s
- We move onto s and check its adjacencies
    - We add v to the stack
- We move into v and check
    - We find w and mark it
- We move into w and see that s has already been found.
- There is nowhere else to go for w, so we backtrack to v then to s then to q
- We return back to q
    - Q finds t next and moves into t
- We find t and check it for adjacencies
    - T finds x and moves into x
- At x, we find z and move into z
    - Z has x which we already check
    - Z has no other outbound directions, so we backtrack back to x to t
- We return to t and find y
    - Y points to q which we have already found. This leaves y with no other outbound directions.
- We backtrack to t then q again to check for adjacencies
    - At q, we find w, but we have already checked w for adjacencies, so we stop.
- However, the graph still has a few white nodes, so we check those
- We check r for adjacencies
    - We find u and go into u
- We check u for adjacencies
    - U finds y but it is black, so we finish U
- We backtrack to r and see y but y is a black node, so we finish r.
- Our stack is now empty, and we have traversed whatever we could find.
- The discovery and finish times are listed in the graph below.


q
1/16
r
17/20
s
2/7
u
18/19
t
8/15
v
3/6
w
4/5
y
13/14
x
9/12
z
10/11

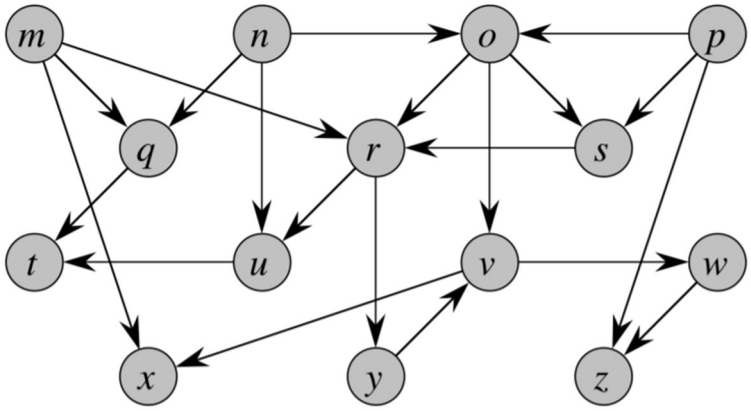
4. What is the running time of DFS if the graph is given as an adjacency matrix? Justify your running time.

- The adjacency matrix is being built as we explore more vertices in a graph.
- The adjacency matrix runs on $O(V^2)$ time because we have to check every vertex.
- DFS traversal operates in $O(|V| + |E|)$ time because it has to check every possible edge of every vertex found
- So, we would have to build the adjacency matrix each time we step into a new vertex, meaning every edge we travel will cost us $O(V^2)$ to build the adjacency matrix

In total, a DFS graph will build an adjacency matrix in $O(|V| + |V^2|)$ time.

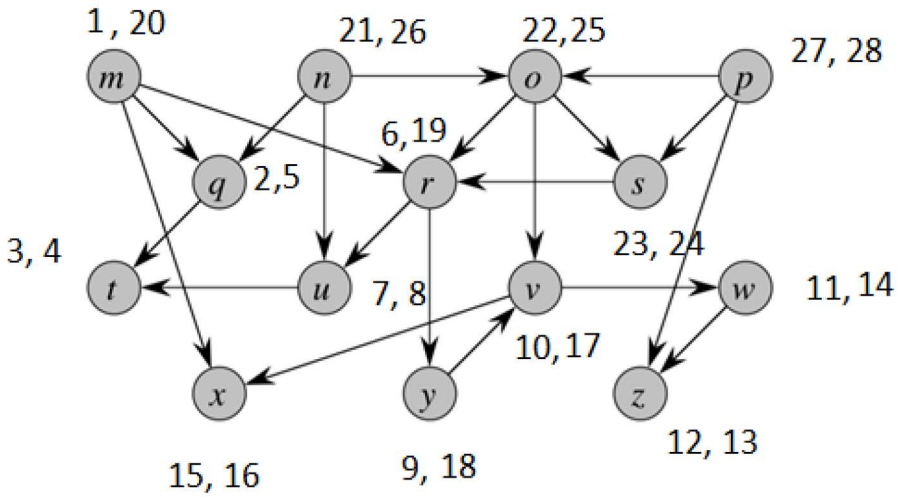
5. Show the ordering of vertices produced by TOPOLOGICAL-SORT when it is run on the DAG below. Assume that the for loop of lines 5 − 7 of the DFS procedure (page 604 in CLRS) considers the vertices in alphabetical order, and assume the adjacency list is ordered alphabetically.

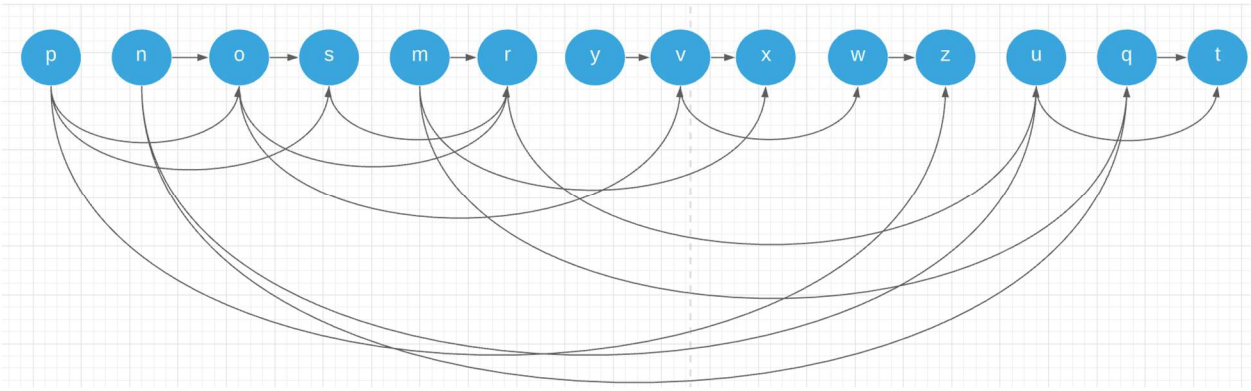
∗Many of these questions came from outside sources.



DFS start time and finish time

- M- 1, 20
- Q – 2, 5
- T – 3, 4
- R- 6, 19
- U – 7, 8
- Y – 9, 18
- V – 10, 17
- W – 11, 14
- Z – 12, 13
- X – 15, 16
- N – 21, 26
- O – 22, 25
- S – 23, 24
- P – 27, 28

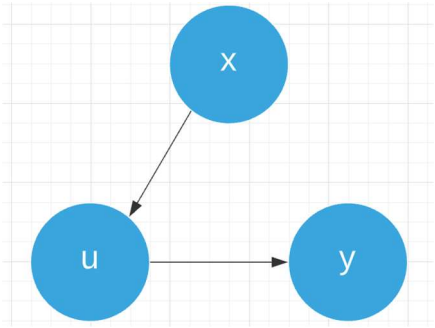

1 , 20
21, 26
22,25
27, 28
m
n
o
p
6,19
q
2,5
r
s
3, 4
23, 24
t
u
7, 8
v
w
11, 14
10,17
x
y
z
12, 13
15, 16
9, 18


p
n
o
s
m
r
y
v
x
w
z
u
q
t

6. Explain how a vertex u of a directed graph can end up in a depth-first tree containing only u, even though u has both incoming and outgoing edges in G.

- In a DFS tree, u will not include arrows that it does not point to
    - If a node points to u but u has no path to that node, then that node will not appear in u's DFS tree.
- In DFS, we build the FS tree by finding a node that's white then traversing all outward paths.
- When we reach the end of that DFS sub-graph and the node's finish time is found, the DFS tree is complete
- Afterward, we create a separate DFS tree and build it using the next white node found.

- We can create 3 nodes, x, u, y
    - X can point to U and U can point to Y which gives u a set of incoming and outgoing edges.
- When building a DFS tree, we can start with Y which will be a DFS tree containing only Y
    - This marks Y as black and we visit the next white node which will be U
- We will build a DFS tree with U.
    - U will be created as a separate DFS tree from Y
    - When checking the paths of U, Y is marked as black and already visited, so U cannot add anymore edges. U will be marked as black.
    - Only U be in its own DFS tree
- We backtrack to X and check its paths
    - It sees U as black and already visited, so X will be in its own DFS tree
    - X gets marked as black and visited.
- If we built the DFS forest in the order of Y, U, X, each node will be in its own DFS tree because their paths have already been visited.



7. Write a method that takes any two nodes u and v in a tree[1] T, and quickly determines if the node u in the tree is a descendant or ancestor of node v.

You may spend O(n) time preprocessing the tree, where n is the number of nodes in the tree.

Give the running time of your method and justify your running time.

- White path theorem-v is a descendant of u if and only if at time u is discovered, there is a path from u to v consisting of only white vertices
- U is an ancestor of v if u has a simple path to vertex v
- To check which node is the ancestor or descendent, we would need to use Depth-First-Search and find the simple path.
    - If we find a tree edge, then u is a parent of v
    - Where u.d < v.d < v.f < u.f
- Forward edge connects u to a descendent v
    - U is an ancestor of v if u.d < v.d < v.f < u.f
- Back edges connect u to an ancestor v
    - Where v.d < u.d < u.f < v.f

To check for ancestors or descendants, we can do preprocessing where we run DFS visit on vertex u until we discover and finish visiting both u and v. This would provide us with u.d, u.f, v.d, and v.f which we can use to identify what type of edge or path they share.

We can then go through the tree to find either u or v. Once we find both nodes, we compare based on their discover and finish times.

This would use O(n) time to build the predecessor subgraph.

CHECK-TREE(T, u, v)

Run DFS and write the u.d and u.f time into each node

A = T.root

//Trace the DFS graph and find node u and node v

for each children i in A

    Check-Children(i, u, v)

Endfor

//If we have reached this point, it means u and v are not on the same path

Print("U and V are not on the same path")

```
Check-Children(I, u, v)

        for each children j in i

                if j == u or j == v

                        if (u was found but not jv)

                                u.d = j.d

                                u.f = j.f

                                Check-Children(j, u, v)

                        Else if v was found but not u

                                v.d = j.d

                                v.f = j.f

                        Else

                                If(u.d < v.d < v.f < u.f)
                                        Print (u is a parent of v)
                                Else if (v.d < u.d < u.f < v.f)
                                        Print (u is an ancestor to v)
                                Else
                                        Print (v is the ancestor to u)


                Else Check-Children(j, u, v)
```

8. (10 points) Bob loves foreign languages and wants to plan his course schedule to take the following nine language courses: LA15, LA16, LA22, LA31, LA32, LA126, LA127, LA141, and LA169.

The course prerequisites are:

(a) LA15: (none)

(b) LA16: LA15

(c) LA22: (none)

(d) LA31: LA15

(e) LA32: LA16, LA31

(f) LA126: LA22, LA32

(g) LA127: LA16

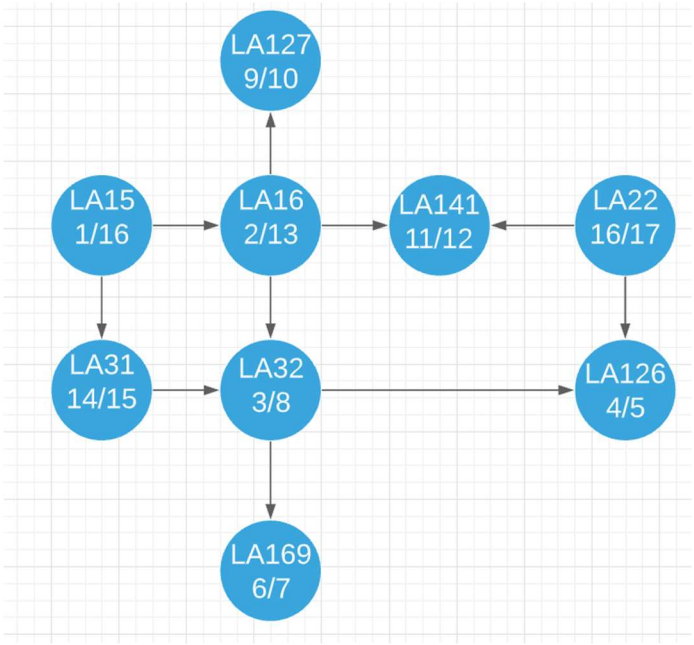
(h) LA141: LA22, LA16

(i) LA169: LA32.

Find a sequence of courses that allows Bob to satisfy all the prerequisites. He will take only one course at a time.[2]

[1]You know who is the parent of any non-root node.

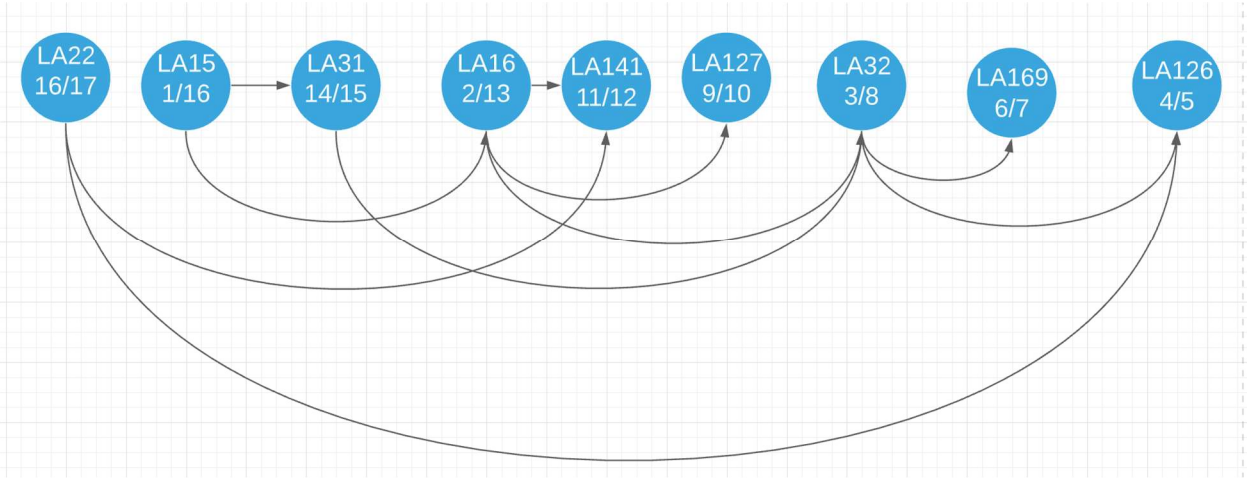
[2]Question from Goodrich, Michael T.; Tamassia, Roberto. Algorithm Design and Applications



- The graph above creates a graph where (LA15, LA16) means LA16 has LA15 as a prerequisite.

- DFS traversal began with LA15 and sorted based on numerical order.

Creating a topological sort from the DFS traversal:



The classes can be taken in the order of:

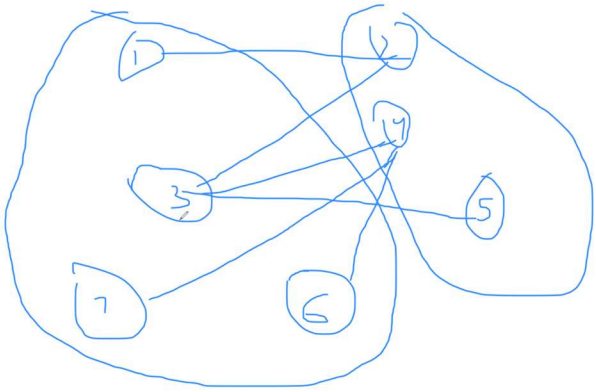
LA22→LA15→LA31→LA16→LA141→LA127→LA32→LA169→LA12

9. You are working for the local animal shelter, and part of your job is to let the dogs play with each other.

There are two large fenced play areas at the shelter, unfortunately, not all dogs get along with each other. During your training, the shelter has told you which dogs do not get along with each other.

Design an efficient algorithm to determine if/how you can separate the dogs into two groups so that in each group the dogs do not fight with each other.

State your running time.



- This is similar to a bipartite graph where dogs must be placed into two groups
    - Must also have an odd number of cycles
- This bipartite graph will be undirected as the condition is that both dogs must get along with each other to be put into the same area
- This will also be represented as (u, v) where an edge means the two dogs do not get along with each other
- For our algorithm, we can start by having all dogs in play area 1 by default
- Then, we can run BFS on dog u
    - If we find another dog during our BFS visit, we put that dog in play area 2 because it means those two dogs do not get along with each other.
- We're running BFS on every node for every edge we find, so the run time would be O(|V| + |E|)

```
//Set up the graph and run BFS search on any unvisited nodes

PLAY-PEN(G)

Cycles = 0

for each vertex u ∈ V - {s}

        u.pen = 1          //By default, start in pen 1 and move to pen 2 if necessary

        u.d = ∞

s.d = 0

Q = Ø
```

```
ENQUEUE(Q, s)
while Q ≠ ∅
   u = DEQUEUE(Q)
  cycle++
      for each v ∈G.Adj[u]
          if v.d == ∞
//If two dogs are adjacent, they do not get along, place them in two different pens.
//Otherwise, let them stay in the same pen
                    if (u.pen == 1)
                              v.pen = 2
                    else if (u.pen == 2)
                              v.pen = 1
                    v.d = u.d + 1
                    ENQUEUE(Q, v)
If there is an even number of cycles
      Print ("Not enough play areas for the dogs")
     Return
Else
          Return the u.pen and v.pen
```

10. (This question might be slightly update on Wednesday November 10th) You are working at the local Dude(ette) Ranch, Rancho Costa Plenty, leading horseback riding adventures. Part of your job is to ensure all your customers get back safely from their time on the trail. To do this, you must carefully consider the riders and horses personalities.

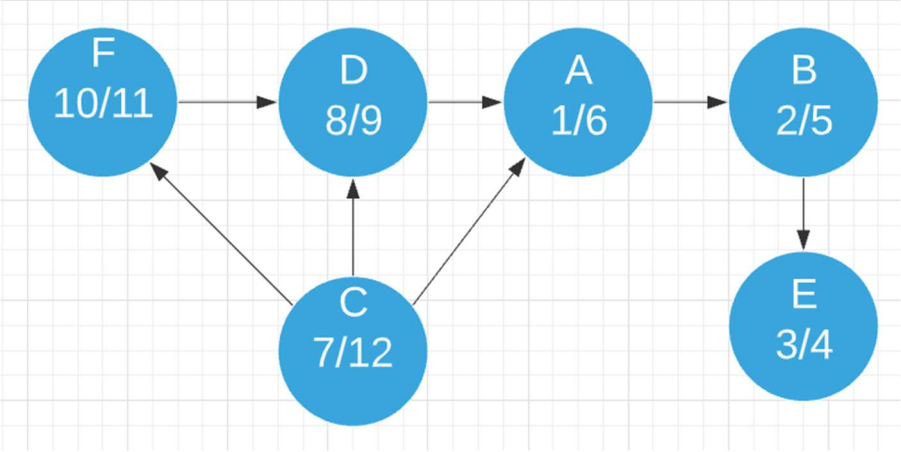
The owner told you before leaving on a vacation to Hawaii that some horses could not be behind other horses on the trail. You were surprised to hear that between any pair of horses on this ranch, there was always at least one horse who could be in front of the other horse. The owner gave you the list of which horses could not be behind which other horses.

Design an efficient algorithm to determine the order of the horses on the mountaineering adventure.

State the running time of your algorithm.

Prove your algorithm works correctly.

- Because we're given a list of what horses could not be behind other horses, we will use the list of horses that can't be behind one another and create the graph that depicts the inverse
    - We create a graph where (u, v) means horse u can be behind horse v
    - If there's no adjacency, that means those two horses cannot be behind each other
    - For example: F->C->D->A->B->E means the order is Horse F (Back of the line), Horse C, Horse D, Horse A, Horse B, Horse E (in front of the line)
- To create an inverse graph, we create a graph G where all nodes are initially connected to every other node
    - Then, we go through the list of horses and remove the adjacency that represents a horse that cannot be behind another horse
    - We will end up with an inverse graph showing what horses can be behind what other horses
- If there is always at least one horse which could be in front of another horse, then that means there is a Hamilton path where we can start with one horse node and travel to every other horse node
- Because there is a Hamilton path, we will run DFS traversal on our graph and look for the largest possible path we can take
    - We can find the longest possible path by running topological sort
    - Because we are guaranteed to have a path for all horses, topological sort will return the longest possible path.
    - In Topological sort, we can't go back during our DFS visit, we can only go forward, adding each horse to the order
- Because our s tarting node may not be the source node for the path, we have to check each path for the longest possible path
- This would cost us O(|V| + |E|) time since we're running DFS on every node and exploring each adjacency only once
    - Topological sort doesn't affect the run time as it runs alongside DFS.



- The graph above is an example where it depicts the relationship of what horses are allowed to be behind what other adjacent horses are available.
    - This means that Horse C can be behind Horse F, D, and A
    - With proper order being C->F->D->A->B->E
    - Horse C is at the back while Horse E is at the front
- We can obtain the line up for horses by running topological sort DFS on the graph starting at A.
    - This gives us the line up of A->B>E
- We start at C
    - This provides C->F->D -> A->B->E

DFS(G, LIST)

//Where LIST represents the list of horses that aren't allowed to be behind

//Where G is initially a graph where all nodes are connected to all nodes, representing all horses that can be behind another

Create PATH as a linked list depicting the path we'll be taking

for each u ∈ G.V

    //Remove any adjacency found in LIST where one horse is not allowed to be behind another

    for every v where (u, v) exists ∈ in LIST)

        Remove v ∈ G.adj(u)

    u.color = WHITE

    u.π = NIL

time = 0

for each u ∈ G.V

```
    if u.color == WHITE
        DFS-VISIT(G, PATH,u)
Return PATH


DFS-VISIT (G, PATH, u)
//Travel through the graph and perform topological sort by adding our vertex when it has run out of adjacencies to travel through
u.color = GRAY
//Adjacency (u, v) means horse u can be behind horse v
for each v ∈G.Adj[u]
    if (v.color == WHITE and v.d has the highest discovery time out of the other adjacent nodes)
    //Visit the nodes with the highest discovery time
        v.π = u
        DFS-VISIT (G, PATH, v)
Insert u to the head of linked list PATH        //insert to the horse to the path
u.color = BLACK
time = time + 1
u.f = time
```

11. (3 bonus points) Think of a good exam/homework question for the lecture 8.

Design an algorithm that can count the total number of cross edges in a directed graph, if any exist. Tree edges do not count towards the count.