Problem 1 Given a directed graph G = (V, E) we define the graph G2 = (V, E2), such that (u, w) ∈ E2 if and only if for some v ∈ V, both (u, v) ∈ E and (v, w) ∈ E. That is, G2 contains an edge between u and w whenever G contains a path with exactly two edges between u and w. Describe an efficient algorithm for computing the adjacency matrix of G2 given the adjacency matrix of G. Present the pseudocode and analyze the running time in terms of |V | and |E|. (This was on a previous final exam)

"""

graph2 initialization contains two for loops to initialize the adjacency list to zero's so it's time complexity will be $O(V^2)$.

We have 3 nested for loop and every for loop is running for V = number of vertices. So time complexity of the 3 nested for loops will be $O(V^3)$ as most inner for loops contain only constant time assignments which is O(1).

So the total time complexity of the algorithm will be $O(V^3 + V^2)$. where V^3 is the bottleneck here so worst-case **time complexity will be $O(V^3)$.**

"""

```
1     def graph_g2(graph: GraphUsingMatrix):
2         graph2 = GraphUsingMatrix(graph.number_of_vertices)
3         for u in range(0, graph.number_of_vertices):
4             for w in range(0, graph.number_of_vertices):
5                 for v in range(0, graph.number_of_vertices):
6                     if graph.adjacency_matrix[u][v] == 1 and
                                graph.adjacency_matrix[v][w] == 1:
7                         graph2.add_edge(u, w)
8                         break
9         return graph2
```

Supporting Class for graph_g2 problem.

```
1     class GraphUsingMatrix:
2         def __init__(self, number_of_vertices: int):
3             self.adjacency_matrix = []
4             self.number_of_vertices = number_of_vertices
5             for index in range(0, number_of_vertices):
6                 self.adjacency_matrix.append([0 for i in range(0,
                                        number_of_vertices)])

7         def add_edge(self, v1, v2):
8             self.adjacency_matrix[v1][v2] = 1
```

Problem 2 Exercise 22.4-1 page 614 from the textbook. It has the same number (but a different page) in the second edition. Note: the DFS-based algorithm must be used, not the one from the next problem.

22.4-1 Show the ordering of vertices produced by TOPOLOGICAL-SORT when it is run on the dag of Figure 22.8, under the assumption of Exercise 22.3-2.

22.3-2 Show how depth-first search works on the graph of Figure 22.6. 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, and show the classification of each edge.

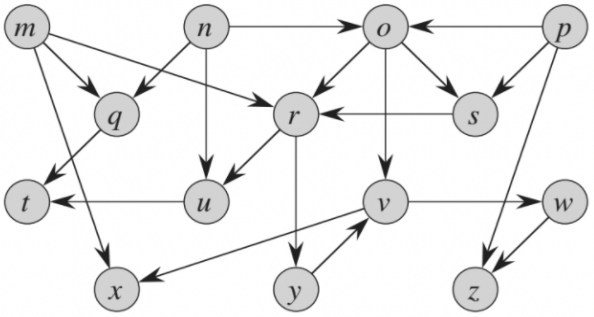


Figure 22.8 A dag for topological sorting.

Assumption according to the problem is that the for loop of lines, 5–7 of the DFS procedure considers the vertices in alphabetical order, and assumes that each adjacency list is ordered alphabetically. So according to the assumption vertices ordering and adjacency list will be as following:-

m = [q, r, x]
n = [o, q, u]
o = [r, s]
p = [o, s]
q = [t]
r = [u, y]
s = [r]
t = None
u = [t]
v = [w, x]
w = [z]
x = None
y = [v]
z = None

DFS start and end time

vertex	*start*	*end*
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

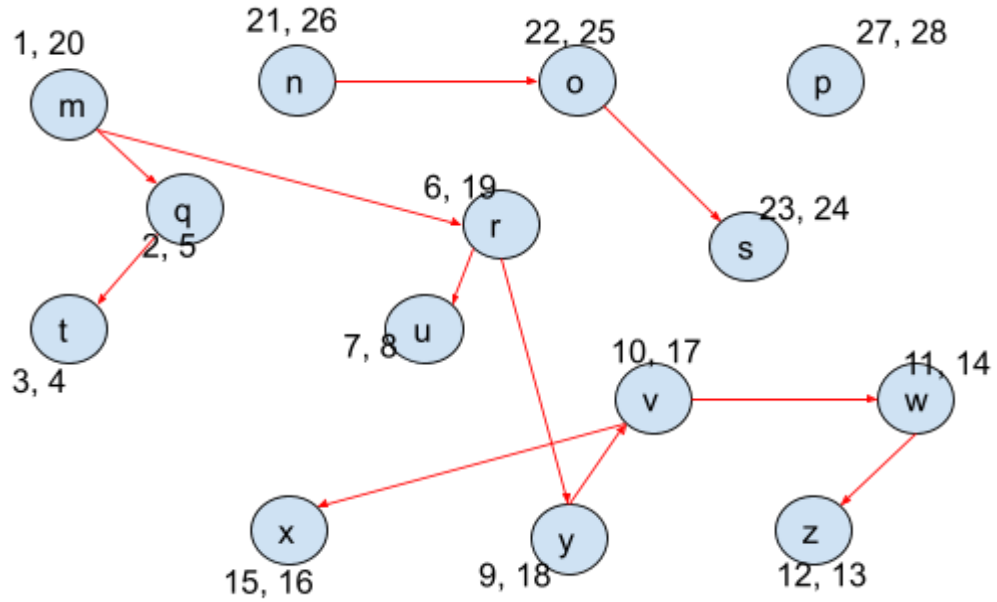
Topological sort on the basis of DFS ending time

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

So Topological sort ordering vertices will be as following:-

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

The DFS Traversal of the above graph will be as following:-



```
TOPOLOGICAL-SORT(G)
1  call DFS(G) to compute finishing times v.f for each vertex v
2  as each vertex is finished, insert it onto the front of a linked list
3  return the linked list of vertices
```

```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
 1  time = time + 1              // white vertex u has just been discovered
 2  u.d = time
 3  u.color = GRAY
 4  for each v ∈ G.Adj[u]        // explore edge (u, v)
 5      if v.color == WHITE
 6          v.π = u
 7          DFS-VISIT(G, v)
 8  u.color = BLACK              // blacken u; it is finished
 9  time = time + 1
10  u.f = time
```

Problem 3 There is another O(|V |+|E|) method for topological sort. Repeatedly find a vertex of in-degree 0, print it, and ”remove it from the graph” by adjusting the in-degree of the other nodes as if this node was removed. Give pseudocode and analyze the running time. Argue correctness. Discuss what happens if the input graph is cyclic.

Solution

```
"""
     Supporting class for the algorithm
"""

class Graph:
   def __init__(self, vertices):
       self.adjacent_list = defaultdict(list)
       self.vertices = vertices

   def add_edge(self, u, v):
       self.adjacent_list[u].append(v)

   def print_graph(self):
       for i in range(self.vertices):
           print(i, ' ', self.adjacent_list[i])

"""
   1. Here first for loop run for the number of vertices = V so
      time complexity to initialize in-degree will be O(V).
   2. Second loop runs for the number of vertices then for each
      vertex runs for every edge so the inner loop runs as
      following E1 + E2 + .... + EV where EN represents the number
      of adjacent edges to Nth Vertex So Total of this will be
      Equals to E = number of edges. So the time complexity of the
      second for loop will be O(E).
   3. The remaining time complexity is O(c1) where c1 is constant
      time complexity.

     So Total Time complexity to compute in degrees is O(V + E +
c1) which equals to O(V + E).

"""

1    def calculate_in_degree(graph: Graph):
2       in_degrees = []
3       for vertex in range(graph.vertices):
4           in_degrees.append(0)

5       for vertex in range(graph.vertices):
6           for adjacent in graph.adjacent_list[vertex]:
7               in_degrees[adjacent] += 1
```

```
8       return in_degrees


"""
   1. calculate_in_degree algorithm time complexity is O(V + E) as
      explained above.
   2. The 4 line for loop is running for V = number of vertices so
      time complexity of for loop is O(V).
   3. On the 7 line we are managing visited status for every vertex
      this will lead to time complexity of O(V).
   4. while loop will run for V = number of vertices and inner for
      including outer while loop will run for E = number of total
      edges. So time complexity of this while loop will be O(V * c1
      + E). As there are some constant time assignments in the
      while loop. So the total time complexity of the while loop
      will be O(V + E).
   5. The remaining code will be constant time complexity c2.

    So Total Time Complexity of the Algorithm will be O(V + E) +
O(V) + O(V) + O(V + E) + O(c2), which will be O(V + E) as we can
remove the constant in the equation.

"""


1    def topological_sort(graph: Graph):
2       in_degrees = calculate_in_degree(graph)

3       queue = []
4       for vertex in range(0, graph.vertices):
5           if in_degrees[vertex] == 0:
6               queue.append(vertex)

7       visited = [False for vertex in range(0, graph.vertices)]

8       topological_sort = []
9       while len(queue) != 0:
10          vertex = queue.pop(0)
11          visited[vertex] = True
12          topological_sort.append(vertex)

13          for adjacent in graph.adjacent_list[vertex]:
14              in_degrees[adjacent] -= 1
15              if visited[adjacent] is False and
                                    in_degrees[adjacent] == 0:
16                  queue.append(adjacent)

17      return topological_sort
```

Correctness:-

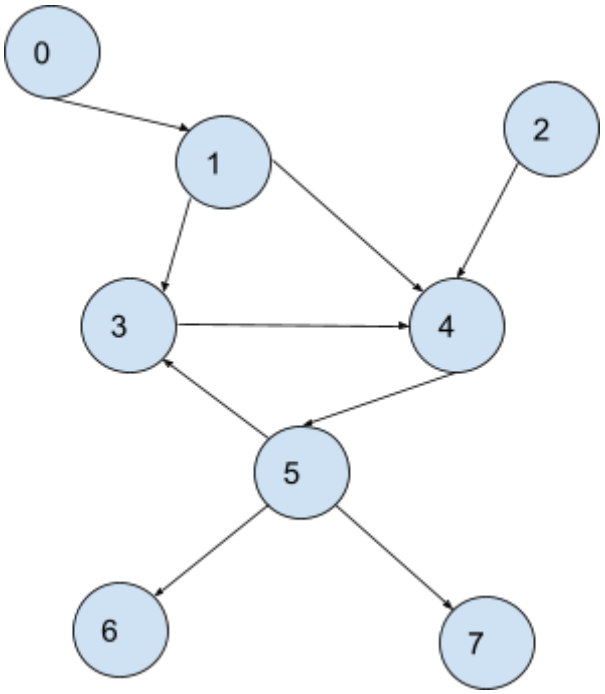
Algorithm Repeatedly find a vertex of in-degree 0, print it, and ”remove it from the graph” by adjusting the in-degree of the other nodes as if this node was removed. So when we remove vertex from graph then the in-degree of adjacent nodes will decrease which will lead to more vertices with in-degree 0 in Directed Acyclic Graph. This will continue till when there are no more vertices to print with in-degree of 0.

The algorithm is finding vertices with in-degree 0. Algorithm goes on until we find all the vertices with in-degree of 0. Which gives us a topological sort.

Cycle in the Graph:-

When there is a cycle in the graph we will never be able to find any vertex with 0 in-degree in the cycle. As cycle means vertices have dependencies on each other. So the above algorithm will give us only 0 in-degree vertices which it was able to find out. Rest vertices will have dependency on each other as there is a cycle so we will not be able to find 0 in-degree vertices anymore.

eg.



As we can see in the above graph algorithm will find out in-degrees of the vertices as follows:-

vertex	0	1	2	3	4	5	6	7
in-degree	0	1	0	2	3	1	1	1

Now algorithm will start topological sort as follows:-

queue = [0, 2]

vertex	0	1	2	3	4	5	6	7
in-degree	0	1	0	2	3	1	1	1

topological sort = []

queue = [2, 1]

vertex	0	1	2	3	4	5	6	7
in-degree	0	0	0	2	3	1	1	1

topological sort = [0]

queue = [1]

vertex	0	1	2	3	4	5	6	7
in-degree	0	0	0	2	2	1	1	1

topological sort = [0, 2]

queue = []

vertex	0	1	2	3	4	5	6	7
in-degree	0	0	0	1	1	1	1	1

topological sort = [0, 2, 1]

So the above algorithm will give a topological sort till 3 elements are able to find 0-indegree by removing 0-indegree vertex. But due to the cycle algorithm will not be able to find 0-indegree vertices anymore and the topological sort list contains less items. This means there is a cycle in the graph or vice versa.

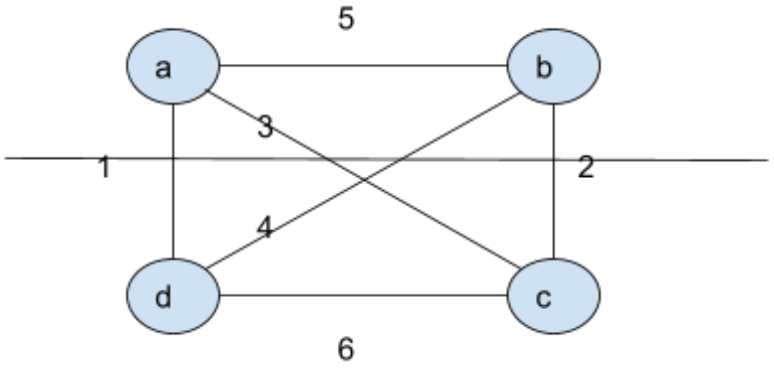
Problem 4 Consider the following divide-and-conquer algorithm for computing minimum spanning trees. Given a **complete graph** G = (V, E), partition the set V of vertices into two sets V1 and V2 such that |V1| and |V2| differ by at most 1. Let E1 be the set of edges that are incident only on vertices in V1, and let E2 be the set of edges that are incident only on vertices in V2. Recursively solve a minimum spanning tree problem on each of the two subgraphs G1 = (V1, E1) and G2 = (V2, E2). Finally, select the minimum-weight edge in e that crosses the cut (V1, V2), and use this edge to unite the resulting two minimum spanning trees into a single spanning tree.

Either argue the algorithm correctly computes a minimum spanning tree of G (regardless of how the partition is done), or provide an example for which the algorithm fails, showing the run of the algorithm (where you can choose the partition) and a better spanning tree. (This was on a previous final exam)

Solution

Above algorithm of divide-and-conquer will fail as we can produce an example of failure.

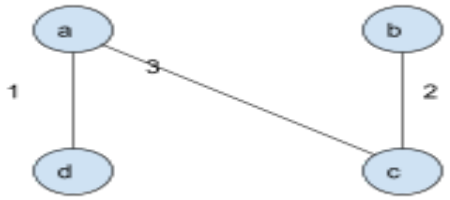
Let's say we have the following complete graph with four vertices a, b, c, d and partitioned(specially to fail the above algorithm) as below:-



V1 = {a, b}, E1 = {5} only one edge so mst is already there with edge weight 5
V2 = {c, d}, E2 = {6} only one edge so mst is already there with edge weight 6
minimum - weight edge in E that crosses the cut (V1, V2) is 1 among {1, 2, 3, 4}.

So according to the algorithm above the total weight of mst of G(V, E)
= weight of mst of G1(V1, E1) + weight of mst of G2(V2, E2) + minimum - weight edge in E that crosses the cut (V1, V2)
= 5 + 6 + 1 = 12

Above complete graph minimum spanning tree should be as following:-



As we can see above graph mst weight should be 1 + 2 + 3 = 6, but with the above algorithm we are getting mst weight of 5 + 6 + 1 = 12 which is not the minimum weight, **therefore the above algorithm is not producing a minimum spanning tree.**