Algorithms

Assignment 8

Ashwin Sathiyanarayanan
002227609

Question 1:

```
DynamicOptimalActivitySelection(start, finish, n):
    let c[0..n+1][0..n+1] be a 2D array
    let optimal_activities[0..n+1][0..n+1] be a 2D array

    for i = 0 to n:
        c[i][i] = 0
        c[i][i+1] = 0
    c[n + 1, n + 1] = 0

    for length = 2 to n+1:
        for i = 0 to n-length+1:
            j = i + length
            c[i][j] = 0
            k = j - 1
            while finish[i] < finish[k]
                if finish[i] ≤ start[k] and finish[k] ≤ start[j]
                                        and c[i, k] + c[k, j] + 1 > c[i, j]:
                   c[i, j] = c[i, k] + c[k, j] + 1
                   optimal_activities[i, j] = k
                k = k - 1

    print("The maximum number of activities is: ", c[0][n+1])
    PrintOptimalActivities(c, optimal_activities, 0, n + 1)

PrintOptimalActivities(c, act, i, j)
    Let Q be an empty queue
    Q.enqueue((i, j))

    while Q is not empty:
        (i, j) = Q.dequeue()
        if c[i][j] > 0:
            k = optimal_activities[i][j]
            print(k)
            Q.enqueue((i, k))
            Q.enqueue((k, j))
```

Explanation:

1. We can make a dynamic algorithm for the activity selection problem where we use memorization.
2. First, we can declare 2 empty 2D arrays, namely c and optimal_activities. The c 2D array is the memorization table that we use to save the outputs of the subproblems, In this c table, c[i][j] represents the maximum set of mutually compatible activities starting from activity i up until activity j. The optimal_activities 2D array is used to save the activities that were utilized to get the optimal solution.
3. First, we loop through the c table and initialize the diagonal values and the values above the diagonal as 0. We do this because the diagonal elements represent the set where there is only one activity, whereas the values above the diagonal represent the set where there are only 2 activities.
4. We don't need the one activity set because as said there is only one activity. We don't need the two activity sets since the 2 activities are consecutive we can only select 1 activity from them. Therefore we set them to 0.
5. Then we set the value of c[n+1][n+1] to 0 because we don't want to include the case where there are no activities left to consider.
6. Then we initiate a main outer for loop from 2 to n+1. We start from 2 because we want to consider the segment of activities of length at least 2.
7. Then we initiate another inner loop starting from 0 to n - length + 1. This is because we want to consider the activities from 0 until the length of the activities that the outer loop is currently in.
8. Inside this inner loop, we first set j to be i + length. This will be the ending activity that we consider to stop the loop.
9. Then we set c[i][j] to 0 because we are just now considering this field. Then we set k to j - 1. This k is used to iterate through the activities that could be included in the optimal_activities when compared with the ith activity.
10. Finally, we initiate a while loop with a terminating condition that the finishing time of the starting activity(i) of the current loop should be

less than the finishing time of the current activity(k) in consideration.

11. Inside the while loop, we check three conditions:
    a. If the finishing time of i is less than or equal to the starting time of k.
    b. If the finishing time of k is less than the finishing time of j, i.e. the last activity in the segment that we are considering
    c. Selecting the current activity along with the activity being considered in the outermost loop yields a set that has a higher length than what is already stored in our memorization table for that subset.
12. If all these are true, then we update our memorization table with the more optimal solution and decrement k heading to the next iteration considering the next element.
13. Then, to print the optimal activities, we make use of a queue. We loop through all the activities and print the activity whose c[i][j] value is not zero. We then add the subset of i, k, and k, j to the queue which we will dequeue and print in the next iteration of the loop.

Comparison of Time Complexities

This whole algorithm runs in $O(n^3)$ time. This is because we have a triple-nested loop here. One to consider a particular length of activities at a time, another to iterate through the activities starting from the 0th element to the last element in the current subset, and one final while loop to consider all the pairs of activities like i and k.

The greedy algorithm for the same activity selection problem runs in $\Theta(n)$ time. So, obviously, the greedy approach is the better pick for the activity selection problem.

Question 2:

There are three cases that we have to disprove using counterexample:

1. selecting the activity of the least duration from among those that are compatible with previously selected activities.
2. selecting the compatible activity that overlaps the fewest other remaining activities.
3. selecting the compatible remaining activity with the earliest starting time.

Case 1:
Counterexample:
(1, 5), (4, 6), (7, 10), (11, 15)

In this case, if we select the activity that has the shortest duration we would get only one activity that is (4, 6). But the optimal solution here would be (1, 5), (7, 10), (11, 15). So we have proved by counter example that selecting the activity with minimum duration does not yield an optimal solution

Case 2:
Counterexample:
(1, 4), (3, 9), (3, 9), (3, 9), (5, 11), (10, 11), (12, 14), (15, 18)

In this case, if we want to select the activity that overlaps the least with other activities, we would select (10, 11). But the optimal solution is (1, 4), (5, 11), (12, 14), (15, 18). So we have proved by counter example that selecting the activity that overlaps the least with other activities does not yield an optimal solution.

Case 3:
Counterexample:
(1, 3), (4, 8), (9, 10)(11, 15)
In this case, if we want to select the activity that has the earliest starting time, we would select (1, 3). But the optimal solution is (4, 8), (9, 10), (11, 15). So we have proved by counter example that selecting the activity that has the earliest starting time does not yield an optimal solution.

Question 3:

The Fractional Knapsack problem has a greedy choice property. We can prove this by taking a counter-example:

Let us consider an example problem:

We have n products in a shop, and let us say P_i it has the maximum value (i.e.) maximum V_i / W_i. Now there are two possibilities:

1. The entire P_i has already been included in the knapsack.
2. P_i has not been included in the knapsack.

Let's say that the original optimal solution is S and the new optimal solution after any changes we make is S'.

Let's see the solution for each case:

1. For case 1 if we already have the P_i in our knapsack, we already have the optimal solution, and the time complexity remains the same. So we do nothing
2. For case 2, if P_i is not included in the knapsack, we need to add it. We add it by removing the contents from the knapsack that is equal to the weight of P_i, that is w_i. Let W_o be weight of all items in the knapsack be the total weight of all products subtracted by the weight of P_i those in the knapsack, (i.e.) w_p.

   $$W_o = W - w_i$$

   Let $w_i` = w_p - w_i$. That is, let $w_i`$ be the remaining weight of P_i that is not in the knapsack.

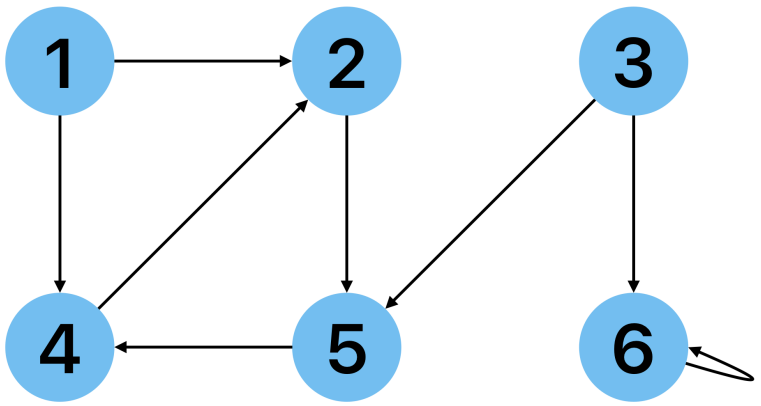
   Now we have to replace the weight of either W_o or $w_i`$, whichever is minimum, from the knapsack and fill it with that weight of P_i.

Therefore, this new optimal solution S' is either the same as S (old optimal solution) or better than S. So if we find a solution that is at least as optimal as the original optimal solution or better than that, our new optimal solution is also acceptable. In this case, our new optimal solution S' is either as good as S or better than S. Hence, the Fractional Knapsack problem has a greedy choice property.

Question 4:

We can get the Transpose of a directed graph G, that is G^T, from the existing adjacency matrix or adjacency list of the existing graph easily. Let us see this with an example for each case:

Main graph:



Adjacency Matrix:

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Transpose of Adjaceny Matrix:

	1	2	3	4	5	6
1	0	0	0	0	0	0
2	1	0	0	1	0	0
3	0	0	0	0	0	0
4	1	0	0	0	1	0
5	0	1	1	0	0	0
6	0	0	1	0	0	1

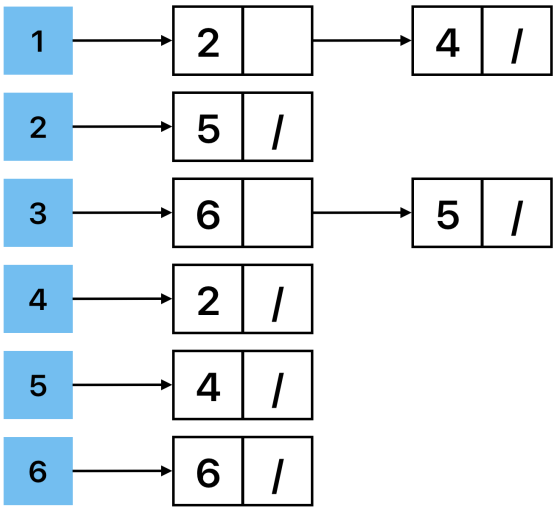
From the example above, we can see that the transpose of the adjacency matrix can be just found by transposing the 1 values of a row, column pair with the column, row pair. That is, if the value (1,2) is 1, it means that a directed edge exists from 1 to 2. So we change that value to 0 and set the value of (2,1) to 1. So now we have changed the direction of the edge from 1 to 2 to now new direction from 2 to 1. In this way, we change the exchange values from the top of the main diagonal to the bottom of the main diagonal to get the matrix of the transposed graph. This whole algorithm takes $O(V^2)$ because we use a nested loop that runs the square of the number of vertices.

Algorithm:

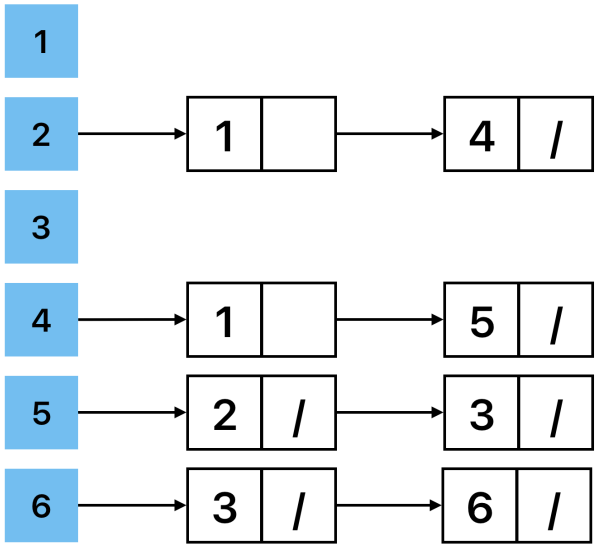
```
GraphTranspose-Matrix(AdjacencyMatrix G, AdjacencyMatrix GT):
    for i = 1 to G.rows
        for j = 1 to G.columns
            GT[i][j] = 0;

    for i = 1 to G.rows
        for j = 1 to G.columns
            GT[j][i] = G[i][j];
```

Adjacency List:



Transpose of Adjacency list:



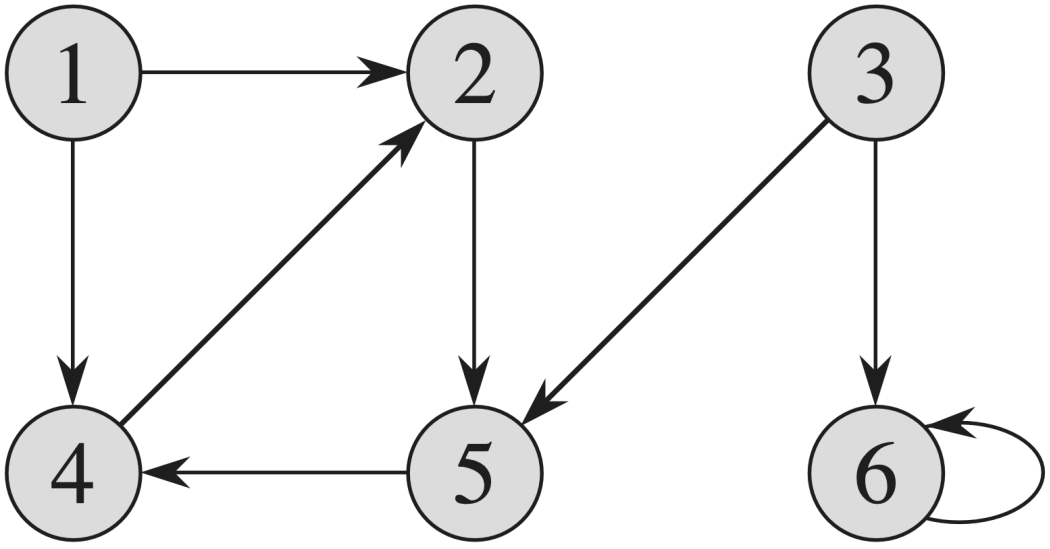
In order to create a transpose of the adjacency list we can use the following strategy. First, we can loop through the array of linked lists, and for each linked list we iterate through them. If we are in the linked list for vertex 1 and encounter a vertex, say vertex 2, we add a new entry into the linked list of vertex 2 to add an edge to vertex 1. We can do this similarly for all the vertices. This whole algorithm takes $O(V + E)$ because we loop through all the vertices and an internal loop iterates through all the edges.

```
GraphTranspose-list(AdjacencyList G, V, E):
    AdjacencyList GT = new AdjacencyList()
    for each vertex v in G:
        for each vertex u in G[v]:
            GT[u].add(v)
```

Question 5:

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2       u.color = WHITE
 3       u.d = ∞
 4       u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11       u = DEQUEUE(Q)
12       for each v ∈ G.Adj[u]
13            if v.color == WHITE
14                 v.color = GRAY
15                 v.d = u.d + 1
16                 v.π = u
17                 ENQUEUE(Q, v)
18       u.color = BLACK
```

Using the BFS method as above we can calculate the d and π values from the graph as shown below.



Initial Tracing:

1. We have selected our source vertex as 3, so the value of s will be node 3.
2. First, we will loop through all the vertices except the s node and set the color of each node to white, the distance of each node from s, d, to ∞ , and the π of each node to NIL.
3. Then we initialize the color, d, and π values of the S node to Gray, 0, and Nil respectively.
4. Then we create an empty queue and push the S node into the queue.
5. Then we initiate a while loop until the queue is empty. In each iteration, we dequeue an element from the queue and set it as U. Then for each adjacent vertex of U we check if we have visited that node earlier using the color and if we have not visited, we update the color, distance, and π of the V node.
6. Then we enqueue the vertex V into the queue so that we can visit its adjacent vertices in the next iteration.
7. In this way, we can calculate the d and π values of each vertices.

Final d and π values:

vertex	1	2	3	4	5	6
d	∞	3	0	2	1	1
π	NIL	4	NIL	5	3	3

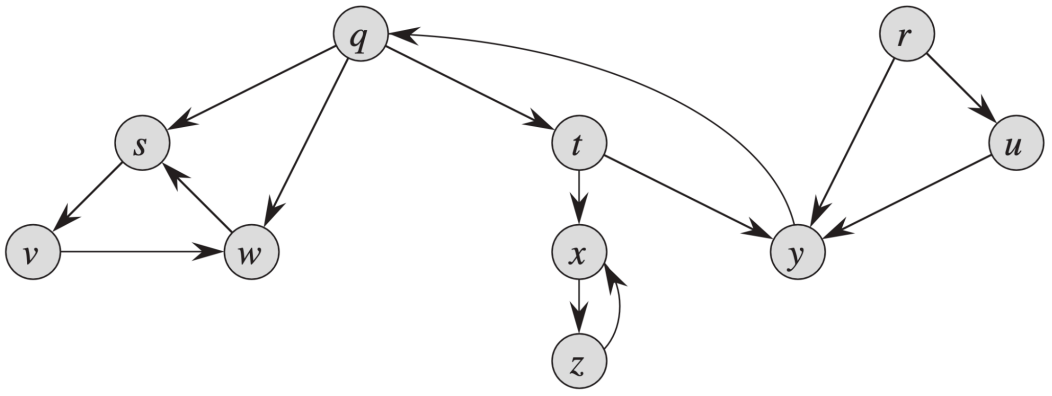
Question 6:

In order to implement the DFS in a graph, we can use the DFS method and DFS-VISIT method. The DFS method is used to iterate through all the unvisited vertices whereas the the DFS-VISIT method is used to visit all the adjacent vertices of the current vertex we are in.

```
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
```

We can implement these two methods for the graph shown below:



Initial Tracing:

1. In DFS, we have to visit every node in the graph unlike BFS, where we have to search only the reachable vertices from the source vertex.
2. So, we iterate through the list of vertices, in this case alphabetically, and send each vertex to the DFS-VISIT method.
3. In each function calling of a vertex in the DFS-VISIT method, we first increment the global time variable and update that time to be the discovery time of the current vertex we are in.
4. Then for each adjacent vertex of the current vertex U, if the vertex is unvisited, we change the π of that node to U and call the DFS-VISIT method recursively for the adjacent vertex V.
5. This recursive call is the main reason that the Depth-first search works accurately as this recursive call enables us to visit the deepest vertex from the current vertex and then return the ancestors after that. Thus the name Depth-first search.
6. Finally, we change the color of the current node U to black to indicate that this has been visited. Then increment the time variable and update that time to be the finishing time for the current vertex. Then we move on to the next unvisited node from the DFS method.

The discovery and finishing time of the vertices in the graph above are:

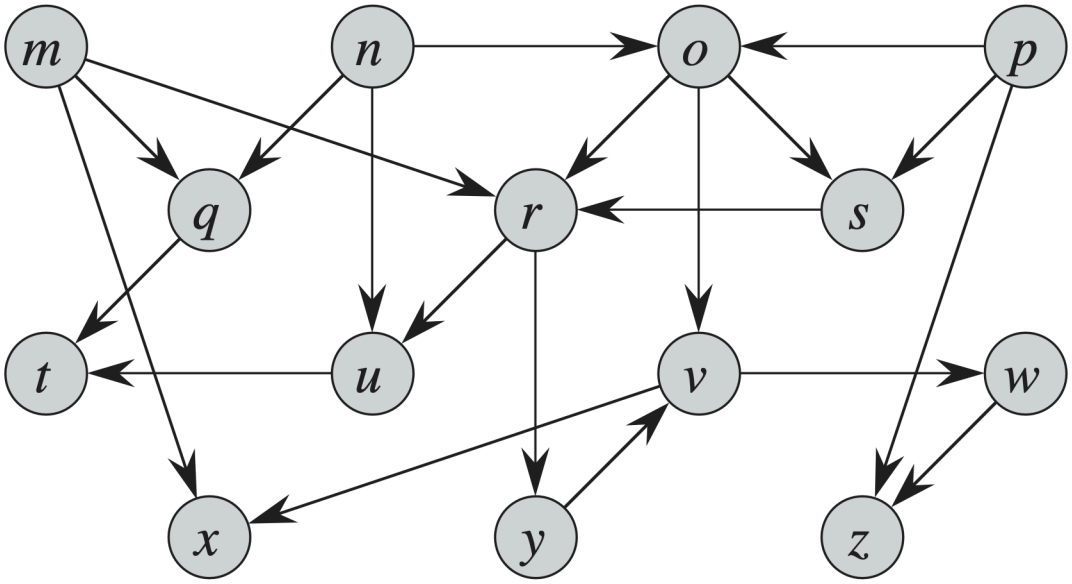
v	q	r	s	t	u	v	w	x	y	z
dt	1	17	2	8	18	3	4	9	13	10
ft	16	20	7	15	19	6	5	12	14	11

Types of edges:

Tree edges: (q, s), (s, v), (v, w), (q, t), (t, x), (x, z), (t, y), (r, u)
Back Edges: (z, x), (y, q), (w, s)
Forward Edges: (q, w)
Cross Edges: (r, y), (u, y)

Question 7:

In order to find the topological ordering of the vertices in a graph, we can use the same DFS and DFS-VISIT methods to first find the discovery and finishing time of each vertex. But in addition to it, we have to add each vertex to the end of a linked list as soon as its finishing time is set. This linked list will the topological ordering of the vertices of the graph.



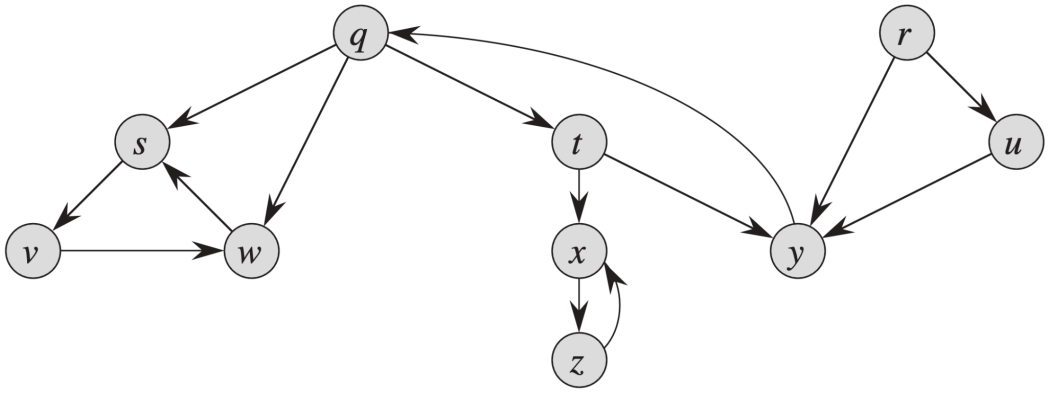
Let's see the topological ordering of vertices of this graph.

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

This is the discovery time and finishing time for each vertex in the graph according to the DFS algorithm.

The topological ordering will be in the descending order of the finishing time of the vertices. The topological order is
p, n, o, s, m, r, y, v, x, w, z, u, q, t

Question 8:



STRONGLY-CONNECTED-COMPONENTS(G):

call DFS(G) to compute finishing times u.f for each vertex u

compute the transpose of G, G^T

call DFS(G^T) to compute finishing times u.f for each vertex u

in order of decreasing u.f of first DFS

output the vertices of each tree in the depth-first forest formed

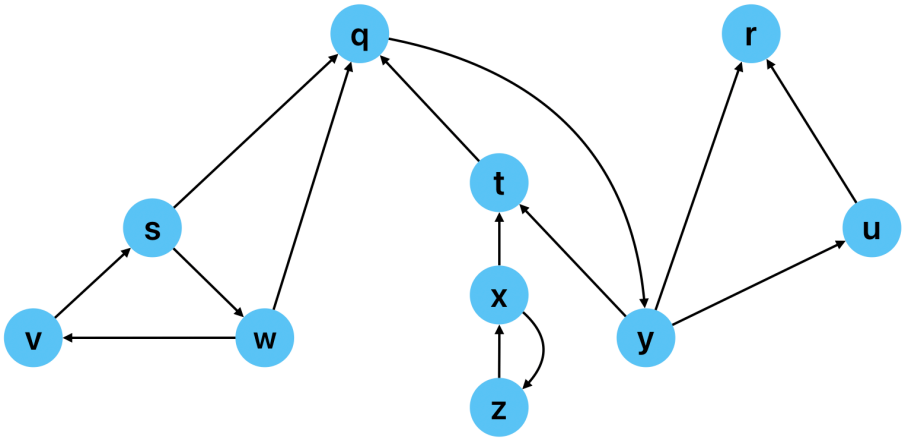
Explanation:

1. In order to find the strongly connected components in a Graph, first we have to call the DFS method for that graph and compute the starting and finishing time of the graph.
2. Then we have to calculate the transpose of the graph using the graph transpose method that we proposed in the previous question (for both adjacency matrix and adjacency list).
3. Then we have to call the DFS method for this transposed graph but this time, the order of calling the vertices should be in the decreasing order of the finishing time that we have calculated for the vertices in our previous DFS method.
4. Now we can just output the vertices of each tree in the depth-first forest thus formed.

1. The is the output of the first DFS method

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

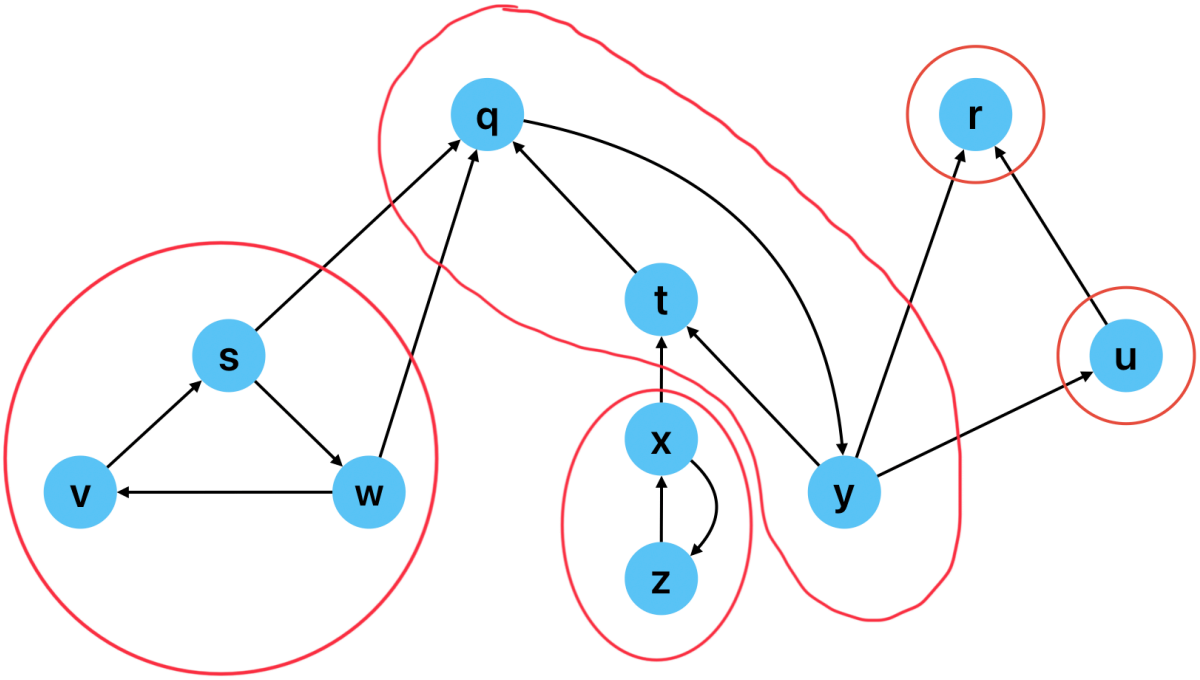
2. Transposed Graph



3. The output of the second DFS method called for G^T

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

4. The strongly connected component forests in this graph are
    a. r
    b. u
    c. q ➡ y ➡ t
    d. x ➡ z
    e. s ➡ w ➡ v


q
r
t
s
u
x
v
w
y
z