CS5800: Algorithms — Iraklis Tsekourakis

Homework 8 Name: Avinash R. Arutla

1. (25 points)

Solution:

```
    def ACTIVITY_SELECTOR(s, f):
    n = len(s)
    c = [[0 for _ in range(n+2)] for _ in range(n+2)]
    p = [[None for _ in range(n+2)] for _ in range(n+2)]

    for l in range(2, n+2):
        for i in range(0, n-l+2):
            j = i + l
            for k in range(i+1, j):
                if f[i] <= s[k] and f[k] <= s[j-1]:
                    q = c[i][k] + c[k][j] + 1
                    if q > c[i][j]:
                        c[i][j] = q
                        p[i][j] = k

    selected_activities = get_activities(p, 0, n+1)
    print("Maximum size of mutually compatible activities:", c[0][n+1])
    print("Selected activities (1-indexed):", selected_activities)
    return selected_activities

def get_activities(p, i, j):
    if p[i][j] is None:
        return []
    else:
        return get_activities(p, i, p[i][j]) + [p[i][j]] + get_activities(p, p[i][j], j)
```

Time Complexity : From the nested loop here, we can inference that the time complexity $O(n^3)$. This is because for each pair i,j it checks each in the range for compatibility. Here n is the number of activities.

The Greedy ACTIVITY-SELECTOR operates in $O(nlogn)$ for sorting and O(n) for activity selection. Here we can clearly understand that greedy algorithm is much more effecient than dyanmic programming for larger inputs.

2. (15 points)

Solution:

To understand that not all greedy approaches produce a maximum size set of mutually compatible activities in the activity selection problem, we can give examples for each strategy where it fails.

Selection an activity with the least duration.

Activity Number	Starting Time	Finishing Time	Duration
1	1	4	3
2	2	3	1
3	3	5	2
4	5	6	1

We always select the activites which have the least duration, from those compatible with the previously selected activities.

Here, based on the approach, we would be selecting activity 2, due to the overlapping times with activity 1 and 3, we cannot select them, so we would be selecting only activity 2 and 4. However if we didnt choose the shortest duration approach, we can choose 1,3,4.

Selection an activity which overlaps the least with other activities.

Activity Number	Starting Time	Finishing Time	Duration
1	1	3	2
2	2	5	3
3	3	4	1
4	4	6	2
5	5	7	2

With this approach, we would select activity 3, as it overlaps with only 2 and 4. Now due to this we can only select 1, 3, and 5. So, we can only select 3 pairs of activities here. However if we didnt follow this approach we can select 1, 4, and 5, which doesnt necessarily increase the number of activities, but negates this theorem of choosing better activities to work on.

Selection an activity with the earliest starting time

Activity Number	Starting Time	Finishing Time	Duration
1	1	2	1
2	2	4	2
3	3	5	2
4	5	6	1

Choosing an acitivity with the least duration would first let us pick 1, then 2, then 4, leading us to pick a total of 3 activities. However the optimal selection would be 1, 3, and 4, which proves that selecting based on the earliest duration may reach an optimal solution but is not guarenteeed.

3. (15 points)

Solution:

Lets consider the following problem to solve the question

Item	Value (v_i)	Weight (w_i)	Ration (k_i)
1	200	10	20
2	150	15	10
3	90	45	2

To prove that the fractional knapsack problem has the greedy choice property, we need to show that taking the item (or fraction of an item) with the highest value-to-weight ratio (value per unit weight) as the first choice leads to an optimal solution.

For the fractional knapsack problem, the greedy choice involves selecting items based on their value-to-weight ratio $\frac{v_i}{w_i}$, starting with the highest. This strategy is predicated on the premise that maximizing value per unit weight at each step will lead to an optimal overall solution.

Let's assume for contradiction that there exists an optimal solution S_i, that does not prioritize the item with the highest value-to-weight ratio k_i.

Case 1 (Underfilled Knapsack): If S^* does not fill the knapsack to its capacity, then we can add more of the item with the highest $\frac{v}{k}$, thereby increasing the total value of S^*. This contradicts the optimality of S^* since we found a better solution.

Case 2 (Fully Filled Knapsack): If S^* is at capacity but includes an item with a lower $\frac{v_j}{k_j}$ than the highest available $\frac{v_i}{k_i}$, replacing any amount of this lower-ratio item with an item of higher ratio $\frac{v_i}{k_i}$ (while maintaining the same total weight) will increase the total value. This contradicts the assumption that S^* was optimal.

Given a knapsack capacity W of 50 units, the greedy approach proceeds as follows:

Item 1 is selected first because it has the highest k_i, after that W becomes 30 units.
Item 2 is selected next, there is enough capacity here, so W becomes 15 units.
Item 3 is selected next, there is enough capacity here, so W becomes 13 units.

The knapsack now contains Items 1, 2,and 3, maximizing the total value within the capacity limit. This example illustrates that by making the greedy choice of selecting items based on the highest value-to-weight ratio, we indeed reach an optimal solution.

4. (20 points)

Solution:

Transpose of a Directed Graph in Adjacency-List Representation.

```
Transpose-Graph-Adj-List(G)
    Initialize Adj_T[1..|V|] as new empty lists for each vertex in G
```

```
        For each vertex u in V do
            For each vertex v in the adjacency list of u (Adj[u]) do
                Append u to the adjacency list of v in Adj_T (Adj_T[v].append(u))
Return Adj_T
```

The algorithm iterates through each vertex u and then through each edge (u, v) from u. Since every edge in the graph is considered exactly once, the running time of this algorithm is $O(V + E)$, where V is the number of vertices, and E is the number of edges in the graph. This makes the algorithm perfect for sparse graphs

Transpose of a Directed Graph in Adjacency-Matrix Representation

```
Transpose-Graph-Adj-Matrix(A)
    Let T be a new matrix of size |V|x|V|, initialized to zeros
    For i = 1 to |V| do
        For j = 1 to |V| do
            Set T[j][i] = A[i][j]
Return T
```

The algorithm requires a nested loop that iterates over each cell of the $|V|x|V|$, adjacency matrix A where each cell corresponds to a potential edge in the graph. Since the operation inside the loop has a constant time complexity, and the loop iterates V^2. The time complexity here is $O(V^2)$. This reflects that the algorithm's efficiency is independent of the graph's sparsity, making it particularly suited for dense graphs

5. (10 points)

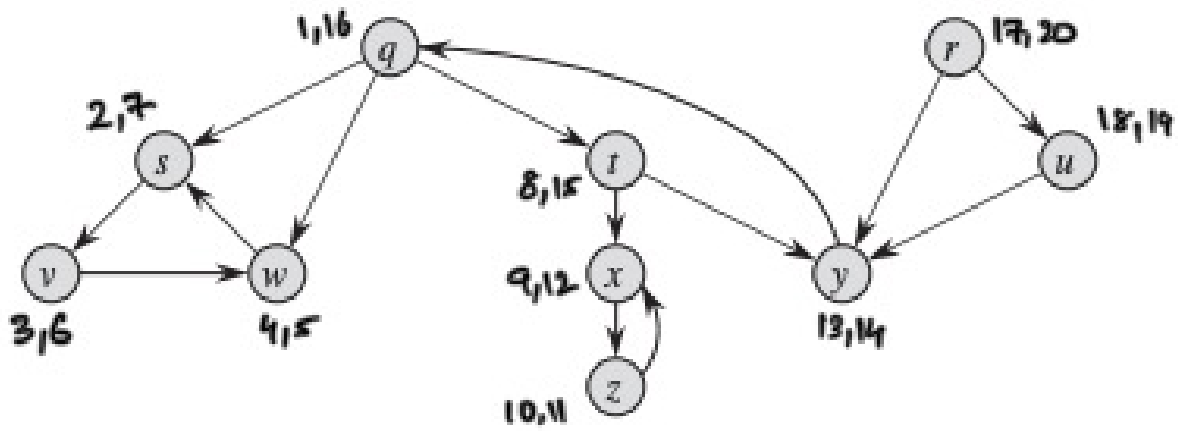
Solution:

Vertex	distance (d)	π
1	3	5
2	1	3
3	0	NIL
4	1	3
5	2	4

This table represents the shortest path distances from vertex 3 and their immediate predecessors in the BFS tree of the given graph. The distances and predecessors adhere to the paths uncovered in a breadth-first traversal starting from vertex 3.

6. (15 points)

Solution:



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

The tree edges here are (q,s), (r,u), (s,v), (q,t), (t,x), (x,z), (t,y), (v,w) The back edges here are (y,q), (w,s), (z,x) The forward edges here are (q,w) The cross edges here are (r,y), (u,y)

7. (10 points)

Solution:

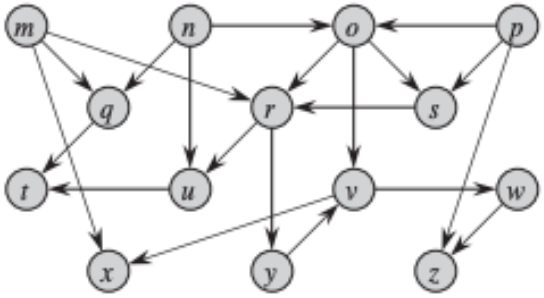


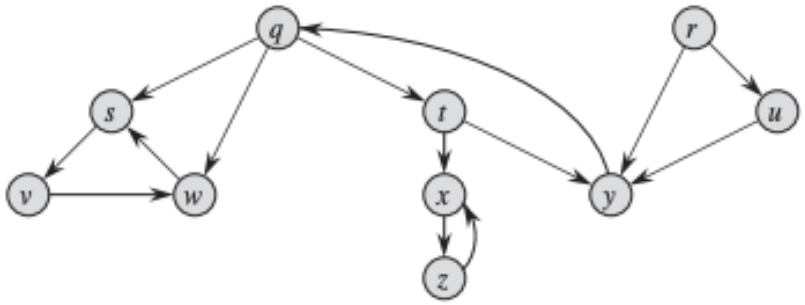
Figure 22.8 A dag for topological sorting.

label	Discovered	Finished
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

The topological result is p, n, o, s, m, r, y, v, x, w, z, u, q, t. This is generated by sorting the results in descending order of their finishing times.

8. (10 points)

Solution:



The finishing times for each node are as follows

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