Algorithm assignment 8
Nisharg Gosai

Problems

1. (25pts) Give a dynamic programming algorithm (in pseudocode) for the activity-selection problem, based on recurrence (15.2 or 16.2). Have your algorithm compute the sizes c[i, j] as defined above and also produce the maximum-size subset of mutually compatible activities. Assume the inputs have been sorted as in equation (15.1 or 16.1). Compare the running time of your solution to the running time of GREEDY-ACTIVITY-SELECTOR.

```
1.  DYNAMIC-ACTIVITY-SELECTOR(startTimes, finishTimes, numActs)
2.      let maxActs[0..numActs + 1, 0..numActs + 1] and selectedAct[0..numActs + 1, 0..numActs
+ 1] be new tables
3.      for i = 0 to numActs
4.          maxActs[i, i] = 0
5.          maxActs[i, i + 1] = 0
6.      maxActs[numActs + 1, numActs + 1] = 0
7.      for length = 2 to numActs + 1
8.          for i = 0 to numActs - length + 1
9.              j = i + length
10.             maxActs[i, j] = 0
11.             actIndex = j - 1
12.             while finishTimes[i] < finishTimes[actIndex]
13.                 if finishTimes[i] ≤ startTimes[actIndex] and finishTimes[actIndex] ≤ startTimes[j]
and maxActs[i, actIndex] + maxActs[actIndex, j] + 1 > maxActs[i, j]
14.                     maxActs[i, j] = maxActs[i, actIndex] + maxActs[actIndex, j] + 1
15.                     selectedAct[i, j] = actIndex
16.                 actIndex = actIndex - 1
17.     print "A maximum size set of mutually compatible acts has size", maxActs[0, numActs +
1]
18.     print "The set contains"
19.     PRINT-ACTS(maxActs, selectedAct, 0, numActs + 1)

20. PRINT-ACTS(maxActs, selectedAct, i, j)
21.     if maxActs[i, j] > 0
22.         selected = selectedAct[i, j]
23.         print "Activity:", selected
24.         PRINT-ACTS(maxActs, selectedAct, i, selected)
25.         PRINT-ACTS(maxActs, selectedAct, selected, j)
```

Variables:
numActs: Represents the total number of activities.
maxActs: A 2D array storing the maximum number of compatible activities for each subproblem.
selectedAct: A 2D array tracking the activities included in the optimal solution for each subproblem.
actIndex: Used as an iterator for the activities in the while loop.

The dynamic programming approach (DYNAMIC-ACTIVITY-SELECTOR) offers a complete solution to the activity-selection problem but at a higher computational cost (O(n^3)) compared to a greedy approach which is Θ(n).

2. (15pts) Not just any greedy approach to the activity-selection problem produces a maximum-size of mutually compatible activities. Give an example to show that the approach of selecting the activity of least duration from among those that are compatible with previously selected activities does not work. Do the same for the approaches of always selecting the compatible activity that overlaps the fewest other remaining activities and always selecting the compatible remaining activity with the earliest starting time.

1. Greedily Selecting the Shortest Duration:

    Activity times: {(2,10),(9,12),(11,21)}

    Using the greedy approach of selecting the shortest duration first, we would choose activity (9,12), which has the shortest duration among the three. However, this choice excludes the possibility of selecting the other two due to overlap. The optimal solution, on the other hand, would be to select activities (2,10) and (11,21), achieving two activities instead of just one.

2. Greedily Selecting the Task with the Fewest Conflicts:

    Activity times: {(−2,0),(1,4),(−1,2),(−1,2),(−1,2),(3,6),(5,8),(7,10),(7,10),(7,10),(9,11)}

    In this scenario, the greedy strategy of selecting the activity with the fewest conflicts would lead us to choose (3,6) first, as it has the fewest overlaps. However, this precludes selecting the optimal set of activities, which would be (-2,0), (1,4), (5,8), and (9,11), totaling four activities.

3. Greedily Selecting the Earliest Start Times:

    Activity times: {(1,11),(3,4),(5,6)}

    By selecting the activity with the earliest start time, we would choose (1,11). This choice, however, excludes the other two activities. The optimal solution would be to select (3,4) and (5,6), which together comprise two activities, more than the single activity selected by the greedy approach.

3. (15pts) Prove that the fractional knapsack problem has the greedy choice property.

Contradiction proof for the greedy-choice property in the fractional knapsack problem:

Initial Setup: Consider an instance *I* of the knapsack problem with *n* items, each with a value v_i and a weight w_i. The items are sorted by decreasing value-to-weight ratio $\frac{v_i}{w_i}$, and the knapsack has a capacity *W*, with $W >= w_n$

Greedy Algorithm's Approach: The greedy algorithm would first consider the item with the highest value-to-weight ratio, which is the last item *n* due to the sorting. It would then allocate $s_n = min(w_n, W)$ to this item and proceed to solve the reduced problem with the remaining items and the reduced capacity *W*-w_n.

Proof by Contradiction:
Suppose there exists an optimal solution $(s_1, s_2, s_3, ..., s_n)$where $s_n < min(w_n, W)$. This means that the knapsack is not filled to its full capacity with the item having the highest ratio.

Let's identify the smallest index *i* where $s_i > 0$ in this solution. This item *i* will have a lower value-to-weight ratio than item *n*.

Now, consider adjusting this solution: reduce the amount of item *i* taken (decrease s_i) and use this freed capacity to increase the amount of item *n* (increase s_n) by the same weight.

This adjustment does not violate the knapsack's capacity constraint but increases the total value, as we are substituting a portion of a less efficient item *i* with a more efficient item *n*.

This new solution has a higher total value than the supposed optimal solution, which contradicts our assumption that $(s_1, s_2, s_3, ..., s_n)$ was optimal.

Conclusion: The assumption that a solution where $s_n < min(w_n, W)$ could be optimal leads to a contradiction. Therefore, the greedy algorithm, which maximizes the amount of the item with the highest value-to-weight ratio until the knapsack is full or the item is exhausted, must yield an optimal solution. This confirms that the fractional knapsack problem has the greedy-choice property.

4. (20pts) The transpose of a directed graph G = (V, E) is the graph GT = (V, ET), where ET ={(v, u) ∈ V x V : (u, v) ∈ E}. That is, GT is G with all its edges reversed. Give in pseudocode efficient algorithms for computing GT from G, for both the adjacency-list and adjacency-matrix representations of G. Analyze the running times of your algorithms.

Adjacency-List Representation
Pseudocode:

```
let Adj'[1..|V|] be a new adjacency list of the transposed G^T
for each vertex u ∈ G.V
        for each vertex v ∈ Adj[u]
                INSERT(Adj'[v], u)
```

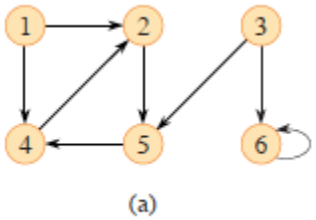
This pseudocode iterates over each vertex u in the graph G, and then for each vertex v in the adjacency list for u, it inserts u into the new adjacency list for v, effectively reversing the direction of the edge.
Time Complexity: $O(|E| + |V|)$

Adjacency-Matrix Representation
Pseudocode:

```
for i from 1 to |V|
        for j from i+1 to |V|
                swap(G[i][j], G[j][i])
```

This pseudocode transposes the matrix in place by swapping elements above the diagonal with those below the diagonal.
Time Complexity: $O(|V|^2)$

5. (10pts) Show the d and π values that result from running BFS on the directed graph of Figure 20.2a (22.2a) using vertex 3 as the source.



(a)

Using vertex 3 as the source the values d and π are:

	d	π
1	infinity	NIL
2	3	4
3	0	NIL
4	2	5
5	1	3
6	1	3

6. (15pts) Show how DFS works on the graph of Figure 20.6 (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 finish times for each vertex and show the classification of each edge.

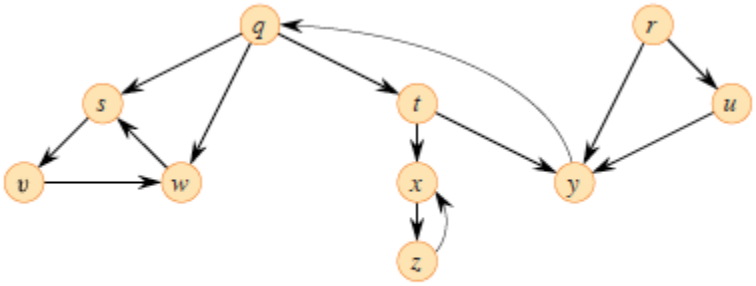


Figure 20.6 A directed graph for use in Exercises 20.3-2 and 20.5-2.

Start at vertex q:
Discover q at time 1. It's the first vertex, so DFS begins here.

Move to vertex s:
From q, move to s, discovered at time 2. Since s is the first adjacent vertex to q, we visit it next.

Move to vertex v:
From s, move to v, discovered at time 3. It is the next white vertex in the adjacency list of s.

Move to vertex w:
From v, we proceed to w, discovered at time 4. This continues the traversal deeper down this path.

Backtrack to v and s:
Since w has no further adjacent white vertices, we finish w at time 5 and backtrack to v, which finishes at time 6, and then to s, which finishes at time 7.

Backtrack to q and move to t:
After finishing s, we backtrack to q and find the next white vertex, which is t. We discover t at time 8.

Move to vertex x:
From t, move to x, discovered at time 9. The DFS continues to x as it is the next white vertex in the adjacency list of t.

Move to vertex z:
From x, move to z, discovered at time 10. z is explored next as part of the depth-first strategy, pushing deeper into the graph.

Backtrack to x, then to t and move to y:
Since z has no adjacent white vertices, we finish z at time 11 and backtrack to x, which finishes at time 12, and then to t.
We then move to y from t, discovered at time 13.

Finish y and backtrack to t:
Finish y at time 14 and backtrack to t, which finishes at time 15.

Backtrack to q and finish it:
Now, backtrack all the way to q and finish it at time 16.

Move to the next white vertex r:
All vertices connected to q are finished, so we look for the next white vertex, which is r, discovered at time 17.

Move to vertex u:
From r, move to u, discovered at time 18.

Finish u and backtrack to r:
Since u has no further white adjacent vertices, finish u at time 19 and then backtrack to r.

Finish r:
Finally, finish r at time 20, completing the depth-first search.

The following table gives the discovery time and finish time for each vertex in the graph.

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

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

7. (10pts) Show the ordering of vertices produced by TOPOLOGICAL-SORT when it is run on the dag of Figure 20.8 (22.8). 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.

To find the ordering of vertices we will see the start and finish times, the table for which is given below

Vertex	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

Writing the vertices in the increasing order of their finish time,
t,q,u,z,w,x,v,y,r,m,s,o,n,p

8. (10pts) Show how the procedure STRONGLY-CONNECTED-COMPONENTS works on the graph of the figure 20.6 (22.6). Specifically, show the finish times computed in line 1 and the forest produced in line 3. 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.

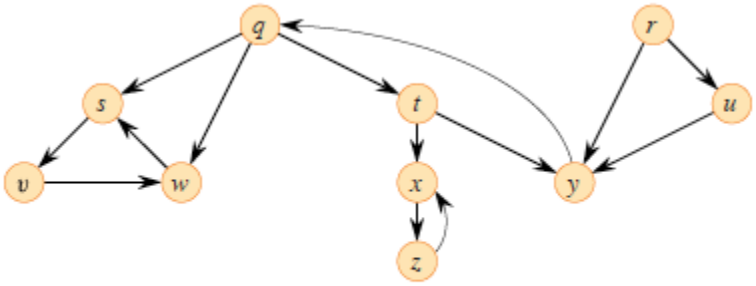


Figure 20.6 A directed graph for use in Exercises 20.3-2 and 20.5-2.

STRONGLY-CONNECTED-COMPONENTS(G)
1 call DFS(G) to compute finish times $u.f$ for each vertex u
2 create G^T
3 call DFS(G^T), but in the main loop of DFS, consider the vertices in order of decreasing $u.f$ (as computed in line 1)
4 output the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component

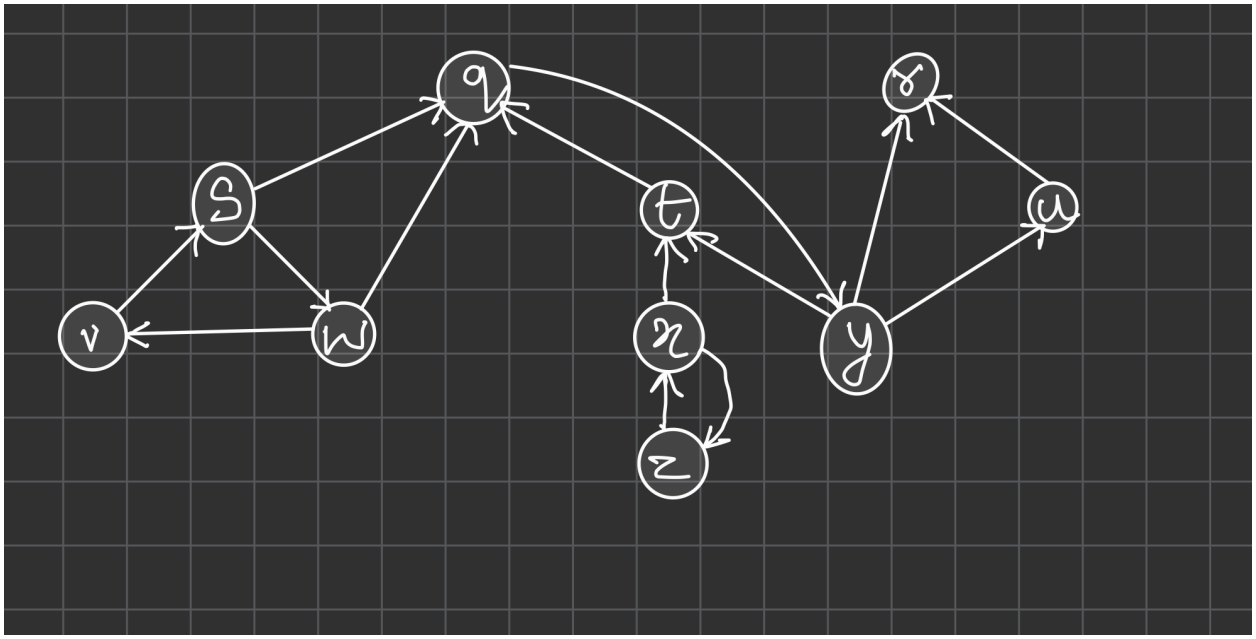
Line 1

The following table gives the discovery time and finish time for each vertex in the graph.

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

Line 2



Line 3

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

Gives us the following components: {r} -> {u} -> {q, y, t} -> {x, z} -> {s, w, v}

The forest produced,