COMP 271 --- Spring 1999

FINAL EXAMINATION - May 26, 1999

Name: **Student ID**:

Tutorial Section (please circle): 1A 1B 2A 2B

All questions should be answered within the space provided after each problem .

1 - 20%] Greedy

a-8% There exist n classes, each represented by an interval [s[i], f[i]) where s[i] denotes the starting time of class i and f[i] its finishing time ($1 \leq i \leq n$). Write the pseudo-code for a *greedy* algorithm to assign as many non-overlapping classes as possible in a single lecture hall and explain briefly its complexity:

```
a] Sort classes by their finishing time f[i]
b]      H={1}
        i=1
        FOR (j=2; j++; j<=n)
                IF (s[j] >= f[i]) {
                        add j to H
                        i=j
                        }
Complexity: nlogn
```

b-12% Convert you algorithm for the case that there exist n available lecture halls (equal to the number of classes) and we want to use the smallest number of halls in order to accommodate all classes. Use an array H[1..n] to store the hall used for each class, that is, H[i]=h if class i is assigned to lecture hall h. Explain briefly how the algorithm works and state its complexity.

```
a] Sort classes by their finishing time
b]      FOR (i=1; i++; i<=n) {// initialization
                H[i]=0 // class i is not assigned to any hall
                LC[i]=0 // finishing time of last class assigned to hall i
                }
        FOR  (h=1; h++; i<=n)   // for each hall
                FOR (i=1; i+; i=n) // for each class
                        IF(H[i]=0 AND s[i]>=LC[h]) {
                        H[i]=h
                        LC[h]=f[i]
                        }
```

Complexity: n^2

2-30%] Dynamic Programming - Backtracking - Branch and Bound

Assume the following version of the 0-1 knapsack problem: there exist n objects and each object i has a weight w[i] and a value v[i]. The goal is to take the objects with the maximum value in a knapsack that can carry weight C. Notice that it is not the same version as the project since now there exists only one object of each type.

a-10% Write a dynamic programming algorithm that returns the optimal value that can be carried in the knapsack (without the objects to be included). Use the notation of the lecture notes, that is, V(k,i) is the optimal value in a knapsack with weight k using only the i first objects and W(k,i) is the corresponding weight.

```
FOR (k=1; k++; k<=C)
        FOR (i=1; i++; i<=n)
                IF (k-w[i]<0)
                        V(k,i)= V(k,i-1)
                ELSE
                        V(k,i)=max{ V(k,i-1), V(k-w[i],i-1)+ v[i]}
RETURN V(C,n)
```

b-10% Write a backtracking algorithm that solves the above problem.

```
Knapsack-BT(n, C)
BT(1,C)

BT(i,C)
b=0
FOR (k=i; k++; k<=n)
        IF w[i]<= C
                b=max{b,v[k]+BT(k+1, C-w[k])}
RETURN b
```

c-10% Write a branch and bound algorithm for the same problem

```
Knapsack-BB(n,C)
Sort all objects according to v[i]/w[i] in decreasing order
target = 0
BB(1,C)


BB(i,c)
b=0
FOR (k=i; k++; k<=n)
        IF w[i]<= C AND (v[i]+(C-w[i])*v[i+1]/w[i+1])> target
                b=max{b,v[k]+BT(k+1, C-w[k])}
                IF b>target
                        target=b
RETURN b
```

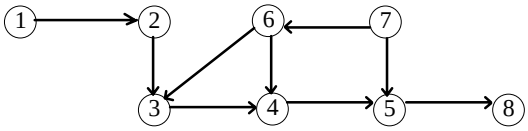
3-22%] Depth First Search - Topological Sort

a-8% Write the pseudo-code for depth first search starting from a node u (use the terminology of the lecture notes where the recursive procedure is called DFSVisit)

```
DFS(u)
for each vertex v
        flag[v]=unvisited
        pred[v]=null
DFSVisit(u)

DFSVisit(v)
flag[v]=visited
for each vertex w adjacent to v
        if (flag[w]=unvisited)
                pred[w]=v
                DFSVisit(w)
```

b-2% Write the calls of the DFSVisit for the following graph starting from node 1 (notice that the algorithm is the same for directed and undirected graphs).



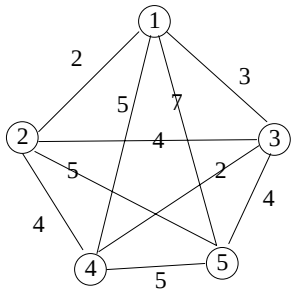
DFSVisit(1), DFSVisit(2), DFSVisit(3), DFSVisit(4), DFSVisit(5), DFSVisit(8)

c-12% Write the pseudo-code for a modified DFS algorithm that produces a topological sort of the nodes in a directed graph, that is, a printout of the nodes so that if node u is before v then it means that there is no path from v to u. For instance, a valid order in the above graph is 7,6,1,2,3,4,5,8; on the other hand 1,2,3,7,6,4,5,8 is invalid because 3 should be after 7 and 6. (Notice: do not use the algorithm for topological sort in the lecture notes because it is based on breadth-first search).

```
Topological sort()
list = ∅
for each vertex u
        flag[u]=unvisited
        pred[u]=null
for each vertex v with in-degree 0
        DFS(v)
reverse list
print list

DFS (v)
flag[v]=visited
for each vertex w adjacent to v // there is a directed edge from v to w
        if (flag[w]=unvisited)
                pred[w]=v
                DFS(w)
list = list + v
```

4-13%] Minimum Spanning Trees and Metric Travelling Salesman Problem
Consider the following graph:



a-3% Write the order with which edges are added to the minimum spanning tree according to Kruskal's algorithm:
(1,2), (3,4), (1,3), (3,5)

b-3% Write the order with which edges are added to the spanning tree using Prim's algorithm, starting with node 1:
(1,2), (1,3), (3,4), (3,5)

c-7% Use the minimum spanning tree to obtain a "good" tour for the metric travelling salesman problem. List the nodes visited and the total cost, and explain briefly how good is this solution compared to the optimal one.

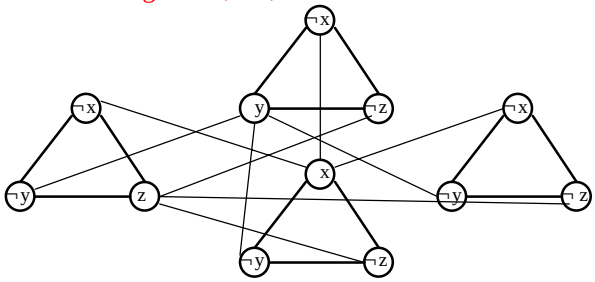
1,2,3,4,5,1 with cost 20. The optimal solution cannot have a cost less than 10.

5-15%] NP-Completeness
Prove that Independent Set is NP-Complete. Use reduction from 3SAT and illustrate it with the following formula:
$(\neg x \vee \neg y \vee z) \wedge (\neg x \vee y \vee \neg z) \wedge (x \vee \neg y \vee \neg z) \wedge (\neg x \vee \neg y \vee \neg z)$

1] IS $\in$ NP
You can verify whether a set of n nodes are independent in polynomial time, by checking if there is an edge between an pairs of nodes ($O(n^2)$).

2] 3SAT can be reduced to IS as follows:

- For each clause you create a triangular subgraph, where every node in the triangle corresponds to a literal.
- Then you add edges between the conflicting nodes, i.e., nodes that cannot be true at the same time.



- The graph has an independent set with *m* nodes (where *m* is the number of clauses in the formula, in this case 4) IF AND ONLY IF the formula is satisfiable.

  Assume that the graph has an IS with *m* nodes. All these nodes belong to different triangles (any two nodes in the same triangle cannot be independent). If we make the literals corresponding to these nodes *true*, then we have a satisfying truth assignment for the formula (since the nodes are independent, they are not in conflict, so they can all be true at the same time).

  Now assume the opposite, that is, the formula is satisfiable. Then we identify a true literal in each clause and pick the node in the triangle of this clause labelled by this literal: This way we collect *m* independent nodes.