CSE 413 Homework 6 Solutions

1)

As a dynamic programming solution, the first step is to determine how the problem depends upon a number of small problems. Consider the problem of finding the path to v using at most i edges. This can be denoted as path(v,i). Assume that the problem of finding a shortest path to any vertex using at most i-1 edges has already been solved.

How can this information be used to determine path(v,i) ? Consider the vertices adjacent to v. Any path from s to v must pass through one of these vertices in order to reach v, using one edge. In other words, the path to v must be the minimum of the costs of the paths to each adjacent vertex plus the cost of the link between that vertex and v. Because of the less than or equal to constraint, path(v,i) may simply be path(v,i-1) if no such path is shorter.

So, path(v,i) = Min(path(v,i-1), {For all w such that w is adjacent to v, Path(w,i-1)+ cost(w->v)}).

Practically, the pseudocode for this (see next page) will compute the matrix for dynamic programming in a the reverse fashion. In other words for some i<=k, first initialize the cost for each vertex as the same as the i-1th column. This makes sure to consider the <= constraint in the problem. Then consider any vertex that can be reached using i-1 edges. Compute the cost of a path from each such vertex to its neighbors, using the path computed in the i-1th iteration. If the new path cost to any neighbor is less than the current cost to each of those vertices, replace the current path with the new path.

It is important to initialize the cost of the path to each vertex as ∞, except for the source vertex, which will be 0. This allows for a check to see if a vertex has been reached by the algorithm yet (by checking if the cost to that vertex is less than ∞).

Note that this algorithm could be simplified by eliminating extra work. If the path to a vertex was not altered in the last iteration, it is not necessary to check that vertex's neighbors again because they have already been checked and the cost to travel to any vertex will never go up with the less than or equal constraint. However, this is not done now to make the pseudocode more readable. It would be simple to check the count of edges used to get the last path to do this, as this variable is already required to trace the path back easily.

Complexity

Note that the algorithm executes of O(k) loop iterations. In this loop, each vertex is checked twice, and each edge of each vertex is checked. Because each edge and each vertex in this loop is touched a constant number of times, this makes the work of the loop O(m+n) per iteration. This gives an O(k(m+n)) time algorithm to fill the matrix. Retrieving the path from the matrix just takes at most one jump for each vertex, giving an O(n) time retrieval, and a O(k(m+n)) time algorithm.

Pseudocode

```
adjacency[node];//this is the vertices adjacent to node
adjacency[node].length;//length of the array
adjacency[node][i].cost;//cost of the ith edge adjacent to node
adjacency[node][i].destination;//index of the destination vertex

matrix[n][k+1]; //matrix for doing dynamic programming
matrix[node][links].cost = the cost of a k link path from s to node;
matrix[node][links].origin = the node one before this one in the path, for tracing;
matrix[node][links].total_edges = total edges used to get there, used for backtrack
for(int i=0; i<n;i++)
{
        matrix[i][0].cost = ∞;
        matrix[i][0].origin = none;
        matrix[i][0].total_edges = 0;
}
matrix[s][0].cost = 0;
matrix[s][0].origin = first;


for(int i=1; i<k;i++)//for 1,2,3...k link shortest paths
{
        //start out each one at the same as the i-1 edge path,
        //this takes into account the <= criteria in the problem
        //so no longer path overwrites a shorter one
        for(int j=0; j<n;j++)
                matrix[j][i] = matrix[j][i-1];//copy all fields
        for(int j=0; j<n;j++)//go through each node
        {
                //find the i-1 paths that exist, then check their
                //neighbors and update path costs
                if(matrix[j][i-1].cost<∞)
                {
                        for(int k=0; k<adjacency[j].length)
                        {//each edge adjacent to this node
                        //compute the i edge path using node j
                        //to each of its neighbors
                                //new_cost is the i edge path from s to
                                //the neighbor of j, going through j.
                                new_cost=matrix[j][i-1].cost+adjacency[j][k].cost;

                                //if the new i edge path is better than the current
                                //path, overwrite it
                                if(new_cost<matrix[adjacency[j][k].dest][i]].cost)
                                {
                                matrix[adjacency[j][k].dest][i].cost = new_cost;
```

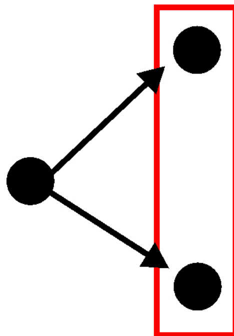
```
                    matrix[adjacency[j][k].dest][i].origin = j;
                    matrix[adjacency[j][k].dest][i].total_edges =
                    matrix[j][i-1].total_edges+1;//adds one more edge
                    }
                }
            }
        }
}

//Now that the matrix is built, retrieve the desired answer.
final_cost = matrix[t][k];
Path = {};
current_vertex = t;
current_edges = k;
//trace the path back
while(current != s)
{
        Path = {current_vertex, Path};
        current_vertex = matrix[current_vertex][current_edges].origin;
        current_edges = matrix[current_vertex][current_edges].total_edges;
}
return [final_cost, path];
```

Problem 2

Consider the topological sort problem. When topological sort has been performed, each vertex has a label such that if v is labeled k, all vertices that can be reached from v have labels greater than k. This can be taken advantage of to find a Hamiltonian path.

Consider the algorithm in 7.14 on page 201 of Manber. This algorithm works by taking all vertices of indegree 0 and adding them to a Queue to be processed. These vertices are all on the same level, and are labeled in one pass. Next, all the out edges from these vertices are removed, in other words for each vertex v that has been in this pass, consider all the vertices w with edges from v to w. Decrease the indegree of w by 1 for each edge coming from v. Do this for all vertices v with indegree zero. Then, add the vertices that just achieved indegree of 0 to the queue and repeat the process until all vertices are labeled. Note that vertices that are disconnected are labeled in the first pass.

If there is to be a Hamiltonian path in an acyclic graph, only one vertex can have indegree of zero. If more than one vertex has indegree greater than zero, then one vertex cannot be reached in a path and so this graph cannot possibly have a Hamiltonian path. If the vertex of indegree zero is removed, this must also be true for the remaining vertices, as each must be reached with no backtracking. So, simply apply this idea on each iteration. If more than one vertex is ever added to the queue, then no Hamiltonian path exists. Otherwise, a Hamiltonian path does exist.



Basically, this algorithm checks for this structure.

Pseudocode

This is exactly the same as 7.14 in the text, however check Queue.size() at each iteration of the loop, and if it is greater than one, return false.

Complexity

Just as topological sort, this is O(|V|+|E|) time.

Problem 3

Again, topological sort will be used. Consider a cycle. In any cycle, a path can be found from any vertex within that cycle to any other vertex. In order for this to happen, no vertex within a cycle can have an in degree of zero (otherwise no other vertex in the cycle could reach it, contradicting the fact that it belongs to a cycle). Note that the topological sort algorithm in 7.14 works by labeling all vertices with indegree 0, then removing the vertices that with an out edge from the original vertex, and so forth. If a vertex is in a cycle, it will never have indegree 0, and will never be labeled by this algorithm. Because of this, the label counter, G_label will never reach n. So if there are labels leftover, there is no cycle.

Pseudocode

At the end of the topological sort on 7.14, add the following line.

```
if(G_label<n)
        return "Cycle Exists";
```

Otherwise, return the sorted labels of vertices.

Complexity

Again, as with topological sort the complexity is O(|V|+|E|) time.

Problem 4

As stated in the hints, a tree must have n-1 edges. Before checking anything else, determine the total number of edges in the graph. If this is not n-1, simply return that the graph is not a tree. Because there are potentially as many as n^2 edges, the problem is how to determine if the tree has n-1 edges in O(n) time?

First, note that each edge is counted in the adjacency list twice, once for each endpoint of the edge. The goal is to determine if there are n-1 edges in the graph. If there are n-1 edges in the graph, then there will be 2n-2 entries in the adjacency list. If there are ever more than 2n-2 entries in the adjacency list, then there are more than n-1 edges in the graph. So, traverse the adjacency list for each vertex, and increment a counter for each edge encountered. If the count ever exceeds 2n-2, then immediately return that the graph is not a tree. If the count never exceeds 2n-2 after all edges in the adjacency list have been counted, check to see if the edge count is 2n-2. If it is, then the graph has n-1 edges. If the graph does not have n-1 edges, return that it is not a tree. This counting takes O(n) time because at most 2n-2 comparisons are made. At this point, it is known that there are n-1 edges in the graph. Therefore, any graph algorithm that takes O(|V| + |E|) time will take O(n) time.

It is known that a tree is connected. If a graph has n-1 edges and has a cycle, then some vertex must be disconnected from the graph as there are not enough edges to connect all vertices and have a cycle. So, run depth first search on the graph. If any vertex is unlabeled at the end, then some vertex could not be reached from the starting vertex, and the graph is disconnected and has a cycle and is not a tree. Depth first search runs in O(|V| +|E|) time, which as stated above is O(n) time for any graph that is being checked at this stage of the algorithm.

If the graph passes these tests, then it is a tree.

Complexity

The complexity of the individual portions of the algorithm is detailed above. Three O(n) time checks are made, giving an O(n) time algorithm.

Pseudocode

```
int total =0;
for(int i=0; i<n;i++)
    for(k = adjacency[i].start to adjacency[i].stop)
        if(total>(2N-2))
            return "Not a Tree";//there are greater than 2N edges
        else
            total++;

total/=2;//there are two entries for each edge, at source and dest

if(total!=n-1)
    return "Not a Tree";

DFS_Label(graph);
for(int i=0; i<n;i++)
    if(vertices[i].labeled=false)
        return "Not a Tree";

return "Is a Tree"
```