Avik Bag
Professor Peysakhov
CS 260 – Assignment #9
23rd August, 2015

Written Problems

Problem 1

We are given that there is a node T_n where n stands for the number of nodes, and t_n stands for the time taken for complete it's task. It's given in the form that it takes t_i time from node T_i to T_j. This essentially means that there is a graphical representation of the nodes. Therefore to find the shortest time to complete all the tasks, in other words go through every node in the hortest time possible, will require the implementation of Floyd's algorithm.

```
define minTime(var cost: list[i][j] list of lists)
begin
   //create a copy of the cost matrix
   A = copy(cost)
   //ensure that the diagonal is set to 0's
   for i in range 0 to len(cost)-1
            A[i][i] = 0
   //after this, a triple nested for loop is implemented
   //this is to compare the path via an intermediary node.
   //if the cost of the transition through this intermediary
   //step is cheaper the direct path, the cost is replaced by the
   //arc cost through the intermediary step. Thus finding the
   //cheapest or most cost effective path.

   for k in 0 to n-1
            for i in 0 to n-1
                     for k in 0 to n-1
                              if A[i][k] + A[k][j] < A[i][j]
                                       A[i][j] = A[i][k] + A[k][j]
End
```

Problem 2

Tree Arc	Backward Arc
A -> B	D -> B
B -> C, F	
C -> D	

Forward Arc	Cross Arcs
A -> D	F -> D
A -> F	E -> D
	E -> F

Depth first numbering
Vertex E is not visited at all.

1	2	3	4	5
A	B	C	D	F

Problem 3

Considering the input of this function is a adjacency matrix in the form of a dictionary, where the key represents the vertex, and the corresponding list contains the connections to the next node. This function is only valid in the case of a directed acyclic graph.

// this makes a list of all possible vertices that can be visited from the root node. The resulting list will then be compared to the keys present in the adjacency list dictionary. If the number of elements in the visited array is less than the number of keys in the dictionary, the supposed node isn't rooted, else true.

```
global array visited[] of type node
def rootedCheck(var D: dictionary(node, list), var key)
begin
    connections = D.get_value(key)
    if len(connections) == 0
        break;
    if key not in visited then:
        visited.append(key)
    for i in 0 to len(connections)-1
        rootCheck(D, connections[i])
end
```

Problem 4

To find the longest path between any two nodes, we will make use of a modified version of the floyd's shortest path algorithm. The adjacency matrix will be established, where any connection will be represented by '1' and any non connections will be set to -1.

```
define longestPath(var adj: list[i][j] list of lists)
begin
   //create a copy of the cost matrix
   A = copy(adj)
   n = len(adj)
   //ensure that the diagonal is set to 0's
   for i in range 0 to n-1
           A[i][i] = 0
            p = []
            n = len(adj)
            res = copy.deepcopy(adj)
            for x in range(0,n):
                init = [0] * n
                p.append(init)
            for x in range(0,n):
                for y in range(0,n):
                        if x==y:
                            p[x][y] = x
                        elif (adj[x][y] != INF):
                            p[x][y] = y
                            p[y][x] = x
                        else:
                            p[x][y] = -1
                            p[y][x] = -1
            for k in 0 to n-1
                for i in 0 to n-1
                    for k in 0 to n-1
                        if A[i][k] + A[k][j] > A[i][j]
                            A[i][j] = A[i][k] + A[k][j]
                            p[i][j] = k
         End
```

The time complexity for this algorithm is $O(n^3)$ because of the fact that there are three nested for loops.

Problem 5

Strongly connected components are:

- E
- A
- B, D, C
- F

Problem 6

To find all the simple paths from one vertices to another in a directed graph, a modification of the breadth first search algorithm will be used.

```
def bfs(graph, root, vertex):
    queue = [root]
    visited = [root]

    while queue:
        node = queue[0]
        edges = graph.get(node)
        path  = queue[0]
        for x in edges:
            if x not in visited:
                queue.append(x)
                visited.append(x)
        y = queue.pop(0))
        if (y == vertex)
            print path
            path.pop()
            path.pop()
        else
            path.append(y)
```