ACA Theoretical Assignment

Q4 **A transitive reduction of a directed graph G = (V,E) is any graph G' with the same vertices but with as few edges as possible, such that the transitive closure of G' is the same as the transitive closure of G. If G is a dag, then the transitive reduction of G is unique.**

Write an algorithm to compute the transitive reduction of a digraph and compute its time complexity.

Solution- Let us assume there exists a graph G'' such that G'' not equal to G' and the transitive closure of G'' is also equal to G. Suppose there is a edge (a,b) in G' which is not present in G'', since transitive closure of G'' is equal to that of G there also exists another path in graph G'' through vertex C. Since there is an extra edge in graph G' it is not minimal. Therefore, G'' does not exist and G' is unique.

An algorithm that can be used to calculate the transitive reduction of a digraph is similar to Floyd's Algorithm, which is used to assign minimal distance between each pair of vertices. If minimal distance is >0 then it is assigned 1 otherwise 0 , the time complexity for the same is O(N^3)

Code for this is –

```
void transitiveClosure(int [,]graph)
  {
    /* reach[,] will be the output matrix that
    will finally have the shortest distances
    between every pair of vertices */
    int [,]reach = new int[V, V];
    int i, j, k;

    /* Initialize the solution matrix same as
    input graph matrix. Or we can say the
    initial values of shortest distances are
    based on shortest paths considering no
    intermediate vertex. */
    for (i = 0; i < V; i++)
      for (j = 0; j < V; j++)
        reach[i, j] = graph[i, j];

    /* Add all vertices one by one to the
    set of intermediate vertices.
    ---> Before start of a iteration, we have
        reachability values for all pairs of
        vertices such that the reachability
        values consider only the vertices in
        set {0, 1, 2, .. k-1} as intermediate vertices.
    ---> After the end of a iteration, vertex no.
         k is added to the set of intermediate
         vertices and the set becomes {0, 1, 2, .. k} */
    for (k = 0; k < V; k++)
    {
        // Pick all vertices as source one by one
        for (i = 0; i < V; i++)
        {
            // Pick all vertices as destination
            // for the above picked source
            for (j = 0; j < V; j++)
            {
                // If vertex k is on a path from i to j,
                // then make sure that the value of
                // reach[i,j] is 1
                reach[i, j] = (reach[i, j] != 0) ||
                              ((reach[i, k] != 0) &&
                               (reach[k, j] != 0)) ? 1 : 0;
            }
        }
    }

    // Print the shortest distance matrix
    printSolution(reach);
}

/* A utility function to print solution */
void printSolution(int [,]reach)
{
    Console.WriteLine("Following matrix is transitive" +
                      " closure of the given graph");
    for (int i = 0; i < V; i++)
    {
        for (int j = 0; j < V; j++){
            if (i == j)
              Console.Write("1 ");
            else
              Console.Write(reach[i, j] + " ");
          }
          Console.WriteLine();
        }
```

Q3 **Consider a diagram of a telephone network, which is a graph G whose vertices repre- sent switching centers and whose edges represent communication lines joining pairs of centers. Edges are marked by their bandwidth, and the bandwidth of a path is the bandwidth of its lowest bandwidth edge. Give an algorithm that, given a diagram and two switching centers a and b, outputs the maximum bandwidth of a path between a and b.**

Consider the algorithm-

```
b[s] = ∞
for each v∈V −{s} b[v] = 0
S=∅
Q = V // Initialize Q, a max-priority queue while Q ≠ ∅
u = EXTRACT-MAX(Q) S = S ∪ {u}
for each v ∈ Adj[u]
if b[v] < min(b[u], w(u, v)) b[v] = min(b[u], w(u, v))
```

The key differences from Dijkstra's algorithm are:

• The field b[v] is initialized to infinity when v = s and 0 otherwise. This is in contrast to Dijkstra's algorithm where it's the opposite.

• Q is a max-priority queue, instead of a min-priority queue.

• When relaxing edge (u, v), the field b[v] is updated to the maximum of the previous b[v] and min(b[u], w(u, v)).

The running time of MAX-DIJKSTRA is the same as that of Dijkstra, because the above modifications do not affect the running time asymptotically.
Let's prove its correctness. Denote the bandwidth of the maximum bandwidth path from s
to v by β(v). The proof is quite similar to that of Dijkstra.

In the given scenario, we are dealing with a graph where each edge has a weight representing its capacity or bandwidth. The goal is to find the maximum bandwidth path from a source vertex s to a target vertex t.

To achieve this, we use a bandwidth algorithm that maintains two values for each vertex u: b[u] and $\beta(u)$. Here, b[u] represents the current estimate of the maximum bandwidth path from s to u, and $\beta(u)$ represents the

true maximum bandwidth from s to u.

Initially, b[u] is set to a lower bound, such as 0, for all vertices u in the graph. This lower bound ensures that $b[u] \le \beta(u)$ holds for

all vertices $v \neq s$.

As the algorithm progresses, the b[u] values are updated through the relaxation of edges. When an edge (u, v) is relaxed, the algorithm checks if the bandwidth value b[u] can be improved by considering the weight of the edge (u, v). If b[u] can be updated to a higher value, it becomes min(b[u],

w(u, v)), where w(u, v) is the weight of the edge (u, v).

The key observation is that at any point, $b[v] \le \beta(v)$ holds for all vertices $v \neq s$. This can be proved by induction. Initially, b[v] is set to satisfy this condition. When relaxing an edge (u, v), if b[v] is updated, it becomes min(b[u], w(u, v)), which is guaranteed to be less than or equal to

$\min(\beta(u), w(u, v))$, and hence $\le \beta(v)$.

Furthermore, we aim to show that when a vertex u is extracted from the priority queue Q, we have $b[u] = \beta(u)$. Initially, this is evident for the source vertex s, as it is the first vertex extracted. Now, for any other vertex $u \neq s$, if $b[u] < \beta(u)$ at the time of extraction, it implies that

there exists a path from s to u with a higher bandwidth than b[u].

Consider the maximum bandwidth path P from s to u. Let y be the first vertex along P that is not in the set S (the vertices extracted from Q) at the time u is extracted. Let x be y's predecessor along P, which is in S. According to our assumption, $b[x] = \beta(x)$ since x has been extracted before u. Moreover, since the edge (x, y) lies on P and has been relaxed, we have b[y] =

$\beta(y)$.

By considering the path P, we can observe that $b[y] = \beta(y) \geq$
$\beta(u) \geq b[u]$. The first inequality follows directly from the definition of
bandwidth, indicating that the maximum bandwidth on the path P is at least as
high as the bandwidth from s to u. The second inequality follows from the

earlier observation that $b[v] \leq \beta(v)$ for all $v \neq s$.

Since $b[u] \geq b[y]$ for u to have been extracted from Q, it
implies that $\beta(u) = b[u]$. In other words, the bandwidth value b[u] for any
vertex u is equal to its true bandwidth $\beta(u)$ when it is extracted from the

priority queue.

This result demonstrates the correctness of the bandwidth
algorithm, ensuring that the final bandwidth values obtained for each vertex

represent their true maximum bandwidth from the source vertex s.

Done By- Samyak Jain(220954)