1. Dicnic's Algorithm

First, let us define a few terms before moving to the actual algorithm.

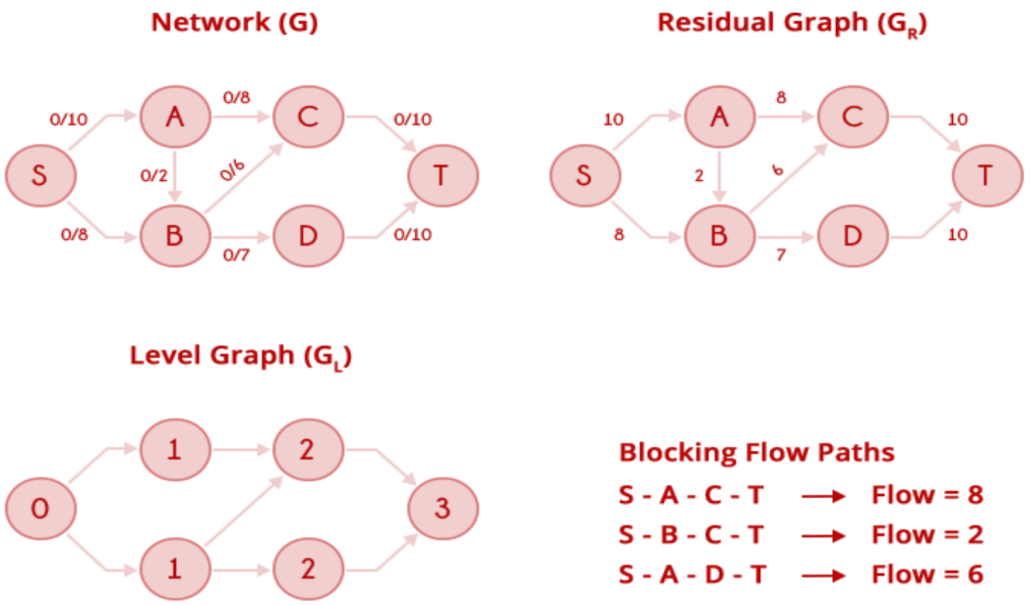
A blocking flow of some network is such a flow that every path from s to t contains at least one edge which is saturated by this flow. Note that a blocking flow is not necessarily maximal.

A layered network of a network G is a network built in the following way. Firstly, for each vertex v we calculate level[v] - the shortest path (unweighted) from s to this vertex using only edges with positive capacity. Then we keep only those edges (v,u) for which level[v]+1=level[u]. Obviously, this network is acyclic.

In Dicnic's Algorithm, on each **phase**, we construct the layered network of the residual network of G. Then we find an arbitrary blocking flow in the layered network and add it to the current flow.

In order to find the blocking flow on each iteration, we may simply try pushing flow with DFS from s to t in the layered network while it can be pushed. In order to do it more quickly, we must remove the edges which can't be used to push anymore. To do this we can keep a pointer in each vertex that points to the next edge which can be used.

Here is an example of a flow network, its residual graph, the level graph and the blocking flows present in the graph.



Here is the implementation of the above idea.

```
/*
        function to find the level graph breadth-first search
        Returns false if no path to sink exists
*/
function bfs() {

        //  queue to store the vertices in bfs
        q = queue()
        level = array(n, -1)
        level[s] = 0
        q.push(s)
        while (!q.empty()) {
                v = q.front()
                q.pop()
                for (edge : adj[v]) {
                        //  if the capacity is full
                        if (edge.cap - edge.flow < 1)
                                continue
                        //  if it is already visited
                        if (level[edge.u] != -1)
                                continue
                        //  add to the level graph
                        level[edge.u] = level[v] + 1
                        q.push(edge.u)
                }
        }

        //  check if sink is reachable
        return level[t] != -1
}
```

```
/*
        function to find an augmenting path in
        the level graph using dfs
*/
function dfs(v, pushed) {
        //  if the min capacity till the vertex is 0
        if (pushed == 0)
                return 0
        //  if sink is reached
        if (v == t)
                return pushed
        //  start with the first unvisited edge
        for (edge : adj[v]) {
                id = adj[v][cid]
                u = edge.u
                //  if the edge is not a part of the level graph
                if (level[v] + 1 != level[u] || edge.cap - edge.flow < 1)
                        continue
                //  get the flow using dfs
                tr = dfs(u, min(pushed, edge.cap - edge.flow))
                if (tr == 0)
                        continue
                //  update the flows if augmenting path is found
                edge.flow += tr
                reverse(edge).flow -= tr
                return tr
        }
        //  return 0 if no path is found
        return 0
}

/*
        Function to find the MaxFlow using Dicnic's
```

```
        algorithm between s and t
*/

function Dicnic(s, t) {
        //  initialize the maxflow to zero
        f = 0
        while (true) {
                //  if level graph does not exist, break
                if (!bfs())
                        break

                //  keep finding augmenting paths
                while (pushed = dfs(s, flow_inf)) {
                        //  add the flow to the netflow from s
                        f += pushed
                }
        }

        //  return maxflow
        return f
}
```

Time Complexity: Each pointer can be moved at most E times, so each phase works in O(VE). There will be at most V phases in the algorithm since level[t] increases in each phase and it can't become greater than V - 1 at the end. Hence the total time complexity of the algorithm is $O(V^2E)$.

Space Complexity: Since we are using adjacency list representation, the space complexity of the above algorithm is the same as that of bfs or dfs which is O(V+E).