Time complexity of Ford Fulkerson algorithm

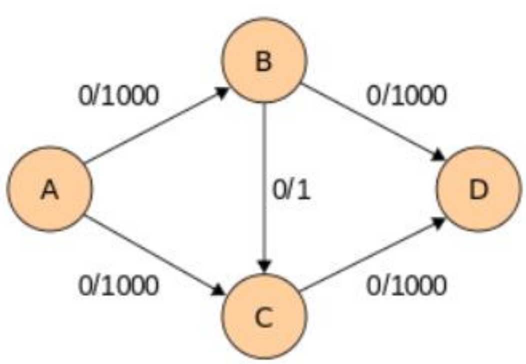
```
class FordFulkerson
{
    static List<int>[] graph = new List<int>[500];
    static List<Arc.Edge> edges;
    static int[] parent;
    2 references
    static bool DFS(ref List<int>[] graph, int node, int sink, int source)
    {

        if (node == sink)  Base case
            return true;

        foreach (var i in graph[node])  Θ(E)
        {
            if (parent[edges[i].to] == -1 && edges[i].capacity > edges[i].flow && edges[i].to != source)
            {
                parent[edges[i].to] = i;
                if (DFS(ref graph, edges[i].to, sink, source)) return true;
            }
        }
        return false;
    }
```

Depth first search is used to find an augmenting path form the source to the sink. DFS takes Θ (V+E) as it starts with iterating over list of adjacent edges of "node" then does the same to all the other nodes of the graph.

```
static int Fordfulkerson(List<int>[] graph, int n, int s, int t)
{
    int max_flow = 0;
    while (true) O(Max Flow)
    {
        parent = new int[n];
        for (int i = 0; i < n; ++i)
            parent[i] = -1;
        DFS(ref graph, s, t, s); Θ(V+E)
        if (parent[t] == -1)
            break;

        int path_flow = int.MaxValue;
        for (int node = parent[t]; node != -1; node = parent[edges[node].from])
        {
            path_flow = Math.Min(path_flow, edges[node].capacity - edges[node].flow);
        }

        for (int node = parent[t]; node != -1; node = parent[edges[node].from])
        {
            Arc.add_flow(node, path_flow, ref edges);
        }
        max_flow += path_flow;

    }

    return max_flow;
}
```

At each iteration ,the DFS is done which takes Θ (V+E),In addition the maximum number of iterations is the Maximum flow value ,This is the worst case scenario: Flow is updated by 1 in each step for a graph .

So the total time complexity is O((E+V).f) which is O(E.f).



It takes **2000 steps** to find the maximum flow in the above graph as DFS is random so it’s possible to pick the middle edge with capacity of 1 every single time so limit flow that can be pushed form source to the sink to be 1 “Bottleneck value”. If we used Breadth First Search instead of Depth First Search in Ford Fulkerson algorithm, it will take **2 steps**.

Time Complexity of Edmonds Karp Algorithm

DFS Analysis and Pseudocode

V = represent # of Vertices

E = represent # of Edges

```
bfs (startNode, endNode)
            {

                initialize parentsList with -1 (to be unvisited)      θ(V)
                Q = (startNode)

                while (Queue not empty)
                {
                      currentNode = Dequeue(Q)     θ(V)
θ(E)
                           for (from 0 to graph[currentNode]. Count)
                    {
                          idx = graph[currentNode][i]
                          e = edges[idx]
                    if (parentsList[to] == -1 and capacity > flow and to !=startNode)

                            parentsList[e.to] = idx
                            if (to == endNode)
                                return
                            Enqueue(to)
                      }
                }

            }
```

Total Complexity of BFS = θ (E + V)

The BFS is used to get the augmented path from the source to the sink.

Edmonds Karp Analysis and Pseduocode

```
EdmondsKarp( startNode,   endNode)
           {
                 maxFlow = 0;
               while (true) O (V E)
               {
                   bfs (startNode, endNode)   O(V+ E )
                   if (parentsList[endNode] == -1)
                       break;

                     flow = MaxValue;
                   for each (node! = -1)  O(E )
O(E)           flow = Min(flow, edges[node].capacity - edges[node].flow);


                   maxFlow += flow;
                     currentNode = parentsList[endNode];
                   while (currentNode != -1)  O (E)
                   {
                       Arc.add_flow(currentNode, flow, ref edges);
                       currentNode = parentsList[edges[currentNode].from];
                   }
               }
               return maxFlow;
           }
```

Total Complexity = $o(VE^2)$

Time Complexity of Dinic Algorithm

1-BFS O(E+V)

```
BFS(s)
{
    initialize list of vertices levels ; O(V)
    set level of source =0
    while (Q not empty) O(E)
    {
        u = DE-QUEUE(Q);
        for each v in G.Adj[u]
        {
            if (cap>0 and level[to] <0)
               level[to]+=level[v]+1
                EN-QUEUE(Q, v);
         }
    }
}
```

2-DFS O(E+V)

```
DFS(v,t,f)
 If v==t return f (base case)
for each iter[v] in G.Adj[u]
{
if (cap>0 and level[v] <level[to])
             DFS(to,t,min_cap)

  if (remain_cap> 0)
       {
         Cap -= remain_cap;
         G[To][Rev].Cap += d;
        return d;
                             }
}
```

2-max flow O(EV**2)

```
MaxFlow(s, t)
  {
  flow = 0;
           while (true) // O(V**2 * E)
                    {
                      BFS(s); // O(E)
                        if (level[t] < 0)
                            return flow;
                        for (0:V)
                            iter[i] = 0;
                        var f = DFS(s, t, int.MaxValue);
                        while (f > 0) // O(EV)
                        {
                            flow += f;
                            f = DFS(s, t, int.MaxValue);
                        }
                    }
                }
```