Module - 4

Maximum Flow

Smita M. le
CLASSMATE
Date
Page 31

Flow network →

→ Flow networks are used in transport of items between locations, a network of routes with limited capacity, etc

Examples:-

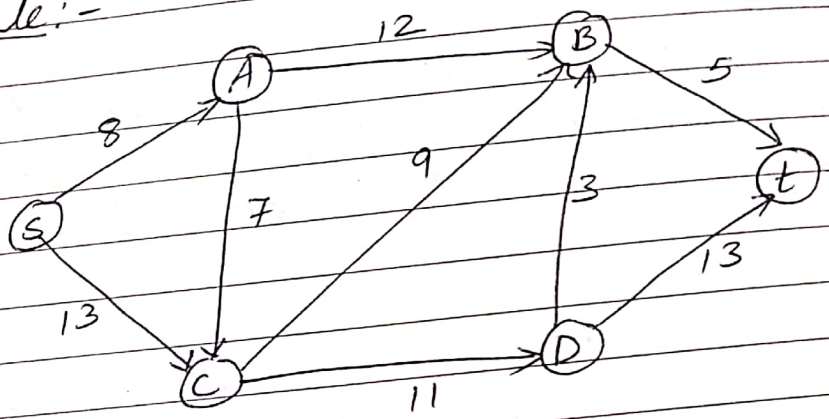
1) Modeling traffic on a network of roads
2) Fluid in a network of pipes
3) Electricity in a network of circuit components.

→ A flow network consists of the following:-

1) A connected directed graph G with non-negative integer weights on the edges, where the weight of an edge is called the capacity of that edge
2) Two distinguished vertices, s and t, called the source and sink, respectively, such that s has no incoming edges and t has no outgoing edges.

→ Given, such a labeled graph, the target is to determine the maximum amount of some item that can be pushed from s to t. The constraint to do this is that the capacity of an edge determines the maximum flow that can go along the edge.

Example:-



→ Here, weight of each edge means the capacity of that edge.
ie. it is the amount of item that can be transferred through that edge.
→ Thus, each edge has an individual capacity, which is the maximum limit of flow that edge could allow.

* Maximum flow →
→ It is defined as the maximum amount of flow that the network would allow to flow from source to sink.
→ There are various algorithms to find the maximum flow.
→ Ford-Fulkerson algorithm is one of the algorithm to find the maximum flow from s to t.

* Ford-Fulkerson Algorithm →

Terms used :-

1) Capacity :- Capacity of a edge is the maximum number of units that are allowed to pass through that edge.

2) Flow :- Flow is the actual units of item moving on the edge.

3) Residual capacity :- It is the number of units that are currently allowed to pass through the edge.
residual capacity = capacity - flow.

4) Augmented path :- It is a path from s to t such that the residual capacity of every edge in that path is greater than zero.

* problem statement :-

→ Given a graph which represents a flow network and every edge has a capacity.

→ Also given two vertices, source 's' and sink 't' in the graph, find the maximum possible flow from s to t with the following constraints :-

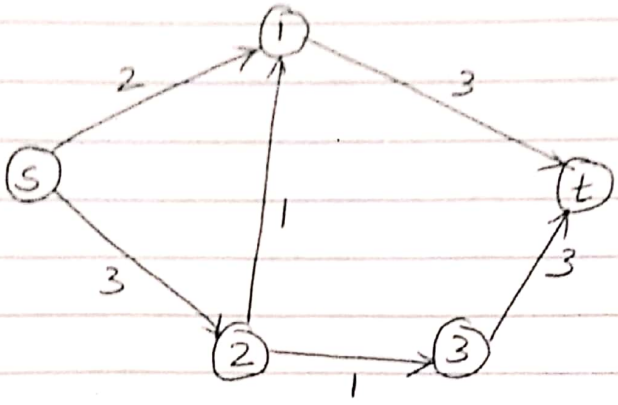
1) Flow on a edge doesn't exceed the given capacity of the edge.
2) Incoming flow is equal to the outgoing flow for every vertex except s and t.

ie. we have to find out maximum number of items that can be moved from s to t.

Algorithm :-

1) Initialize flow to zero
2) path = find_augmented_path (G, s, t)
3) while path exists
{
augment the flow along this path
G_r = create_residual_graph ()
path = find_augmented_path (G_r, s, t)
}
4) Return flow.

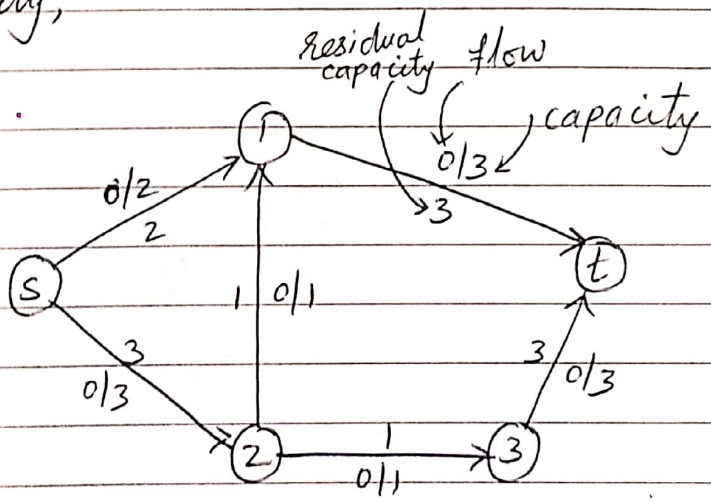
Example:-
Find the maximum flow from s to t.



Each edge has, ~~flow~~
flow | capacity | residual capacity

Residual capacity = Capacity - flow.

Initially,

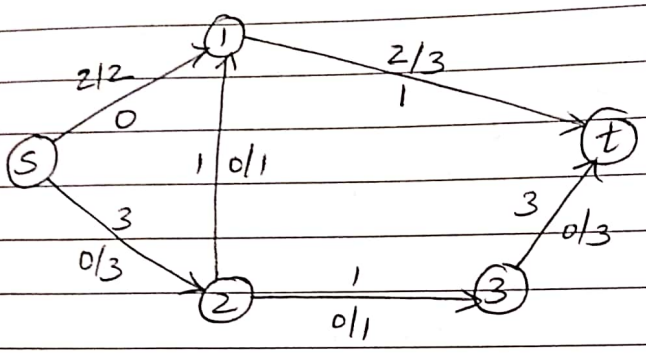


Augmented path is a path from s to t, such that the residual capacity of every edge in that path is greater than zero

Find all such augmented paths:-

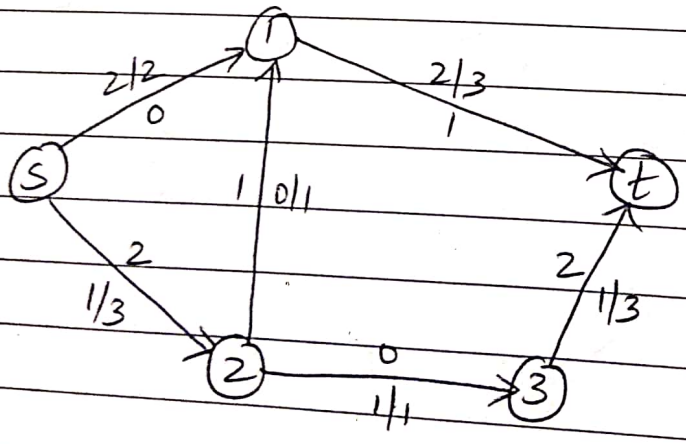
1) $s \xrightarrow{2} 1 \xrightarrow{3} t$

Maximum capacity = 2.



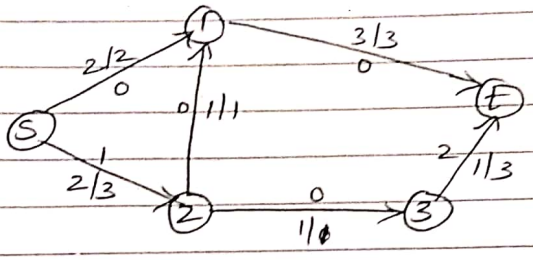
2) $s \xrightarrow{3} 2 \xrightarrow{1} 3 \xrightarrow{3} t$

Maximum capacity = 1



3) $s \xrightarrow{2} 2 \xrightarrow{1} 1 \xrightarrow{1} t$

Maximum capacity = 1



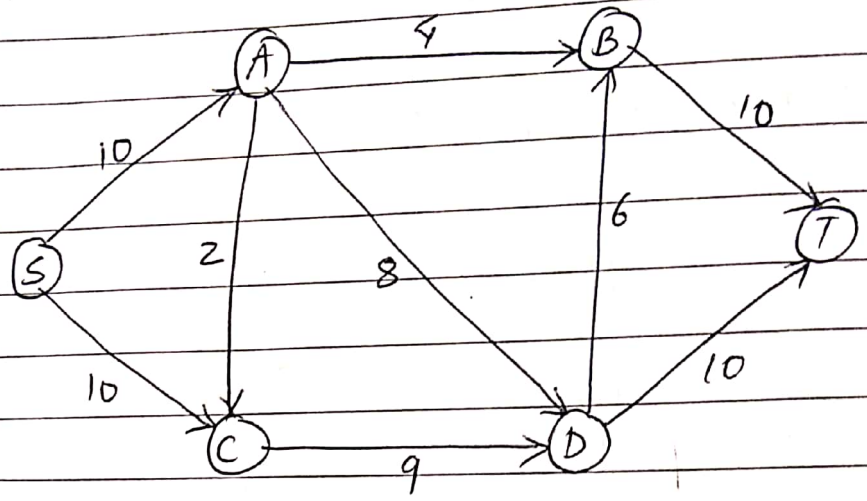
No more augmented paths

Hence, the maximum flow = 2 + 1 + 1
= 4

Smita Attarde

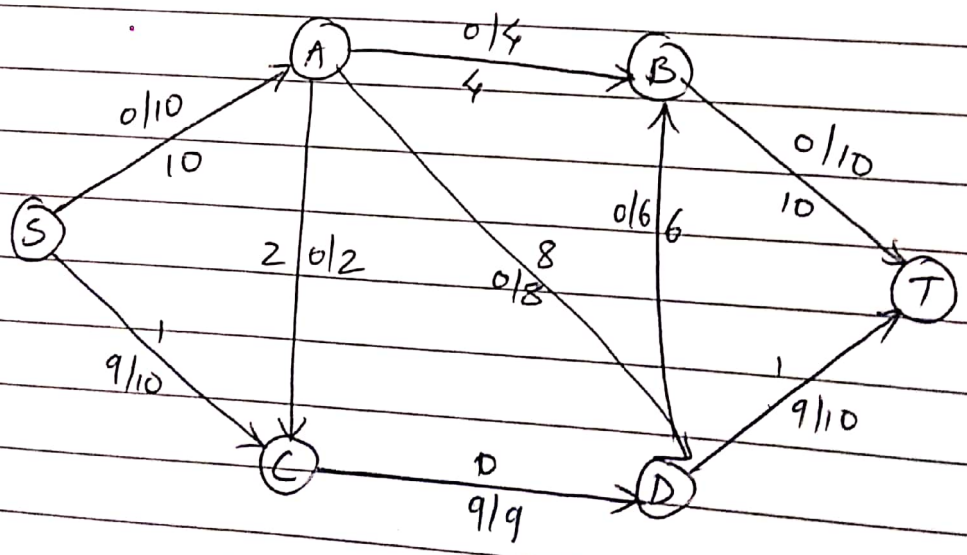
Date
Page 38

Example 2:-

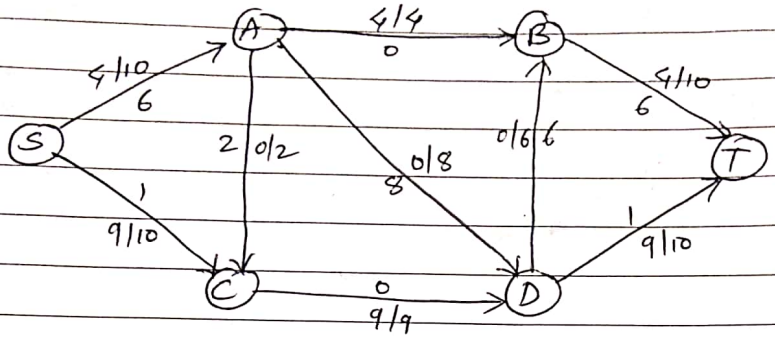


Augmented paths →

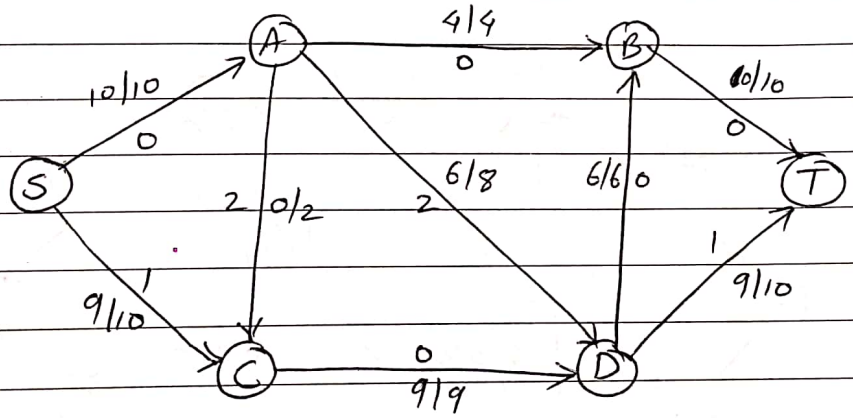
1) S $\xrightarrow{10}$ C $\xrightarrow{9}$ D $\xrightarrow{10}$ T , Max^m capacity = 9



2) S $\xrightarrow{10}$ A $\xrightarrow{4}$ B $\xrightarrow{10}$ T , Max Capacity = 4



3) S $\xrightarrow{6}$ A $\xrightarrow{8}$ D $\xrightarrow{6}$ B $\xrightarrow{6}$ T , Max capacity = 6

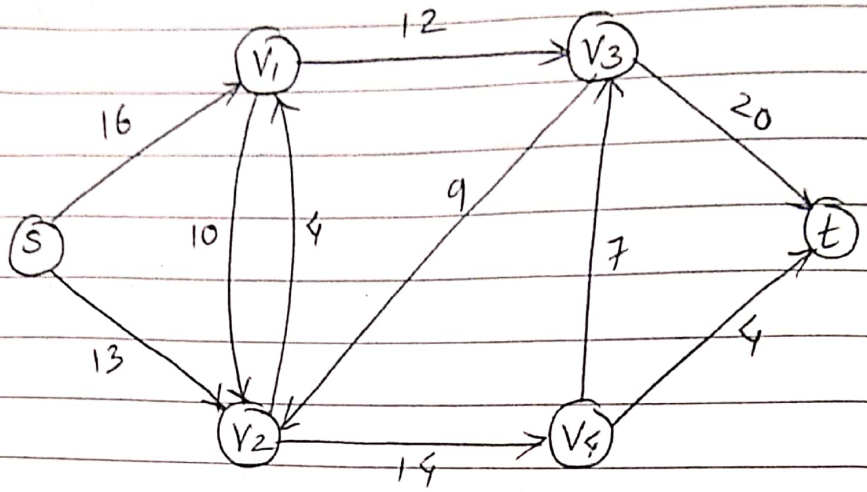


No more augmented paths.

∴ Maximum flow = 9 + 4 + 6 = 19

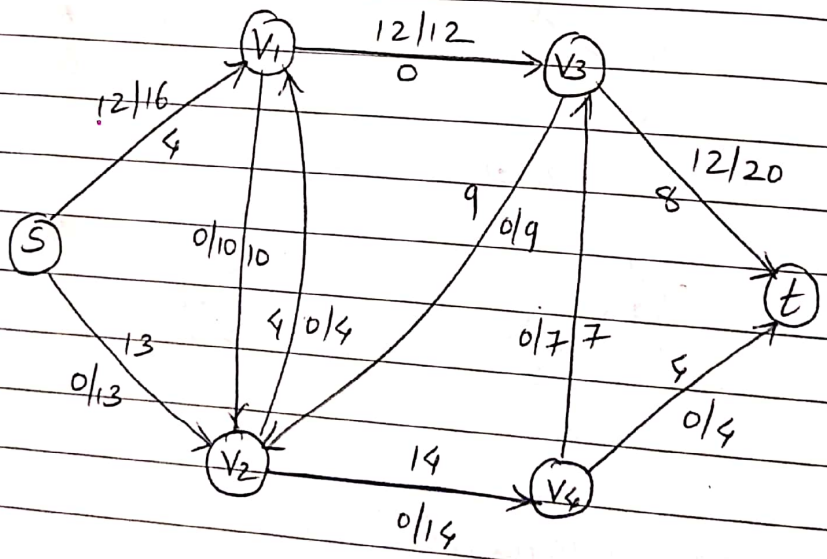
Smita Attarde

Example 3:-



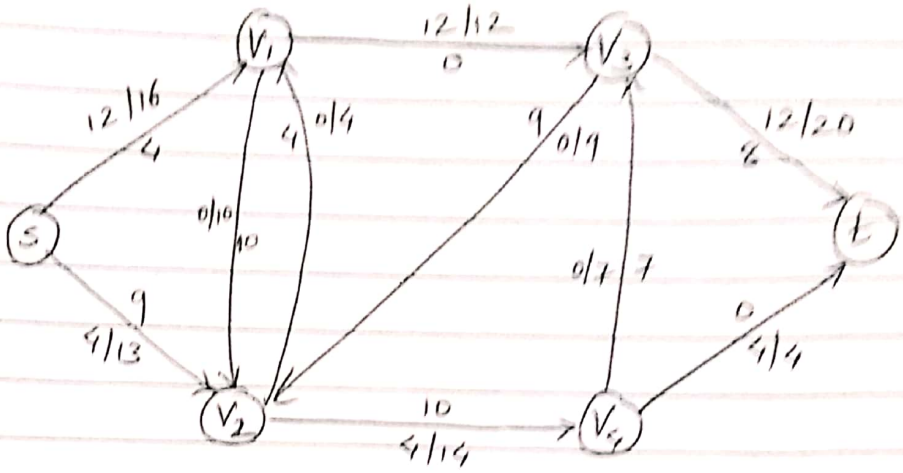
Augmented paths:-

1) $S \xrightarrow{16} V_1 \xrightarrow{12} V_3 \xrightarrow{20} t$, Max^m capacity = 12

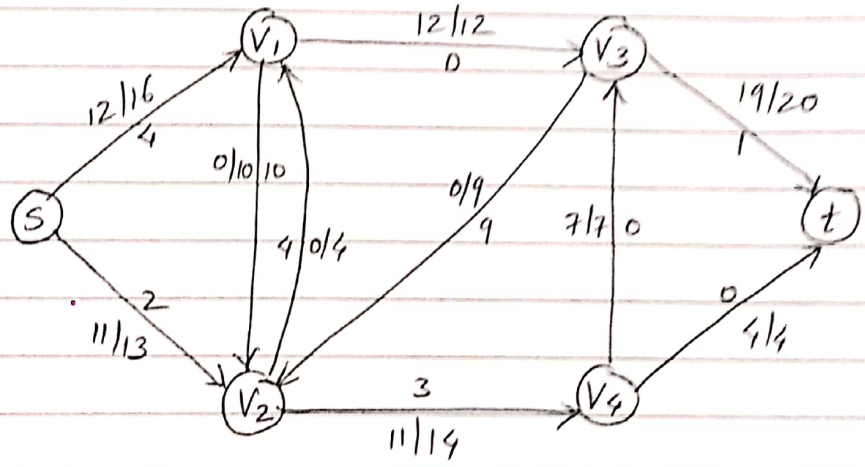


Smita Attarde

2) $s \xrightarrow{13} v_2 \xrightarrow{12} v_4 \xrightarrow{9} t$, Max^m capacity = 9



3) $s \xrightarrow{9} v_2 \xrightarrow{10} v_4 \xrightarrow{7} v_3 \xrightarrow{8} t$, Max^m capacity = 7



No more augmented paths.

∴ Maximum flow = 12 + 4 + 7

= 23

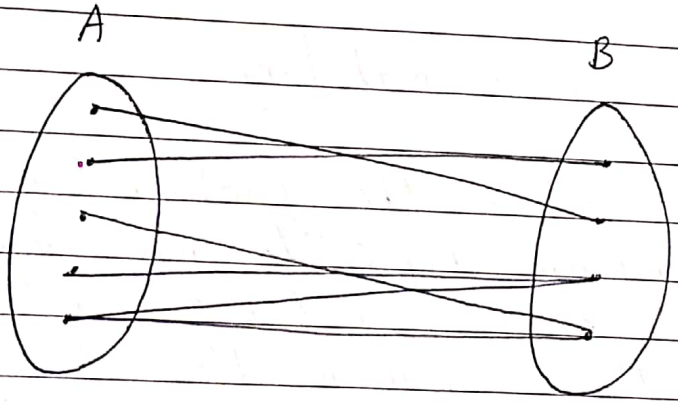
Smita Attarde

Max Bipartite Matching →

Bipartite Graph:-

A graph $G(V, E)$ is called bipartite if its vertex set $V(G)$ can be partitioned into two non-empty disjoint subsets $V_1(G)$ and $V_2(G)$ in such a way that each edge $e \in E(G)$ has its one end point in $V_1(G)$ and other end point in $V_2(G)$. The partition $V = V_1 \cup V_2$ is called bi-partition of G.

ie. A bipartite graph is a graph in which the vertices can be partitioned into two disjoint sets A and B such that every edge connects a vertex in A to a vertex in B.



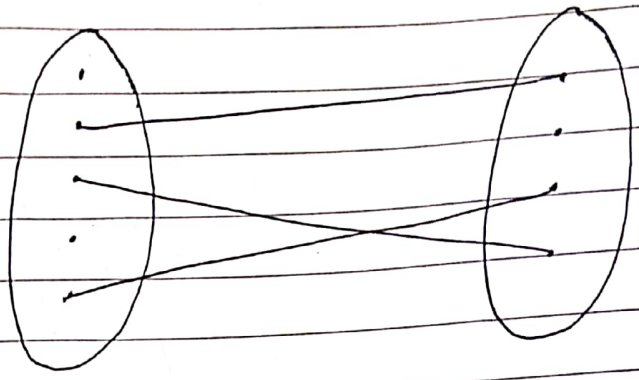
→ A and B are called partite sets.

→ A bipartite matching is a set of edges with no common vertices.

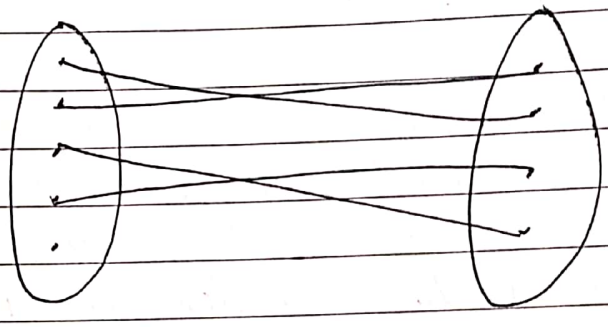
→ Matching with maximum number of edges is called maximum matching

→ Maximum bipartite matching for a graph is not unique.

eg)



Matching

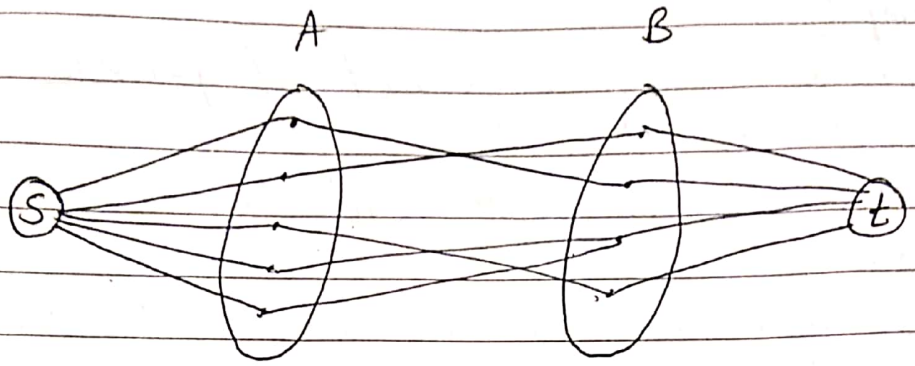


Maximum matching

→ We can reduce the problem of finding the maximum bipartite matching of a graph to a flow network and then finding the maximum flow for the same.

→ This flow is the maximum bipartite matching for the graph.

→ For this, we add a source and sink vertex and connect them to all the vertices in A and B respectively.



→ Assuming the edge capacity of each edge to be 1 unit, we find the maximum flow in the graph to get the maximum bipartite matching.

→ The Ford-Fulkerson algorithm can be used to find the maximum flow in this graph.

Time Complexity →

Let,

V = no. of vertices in the graph G

E = no. of edges in the graph G.

~~No. of edges in the new graph G' = M + 2N.~~

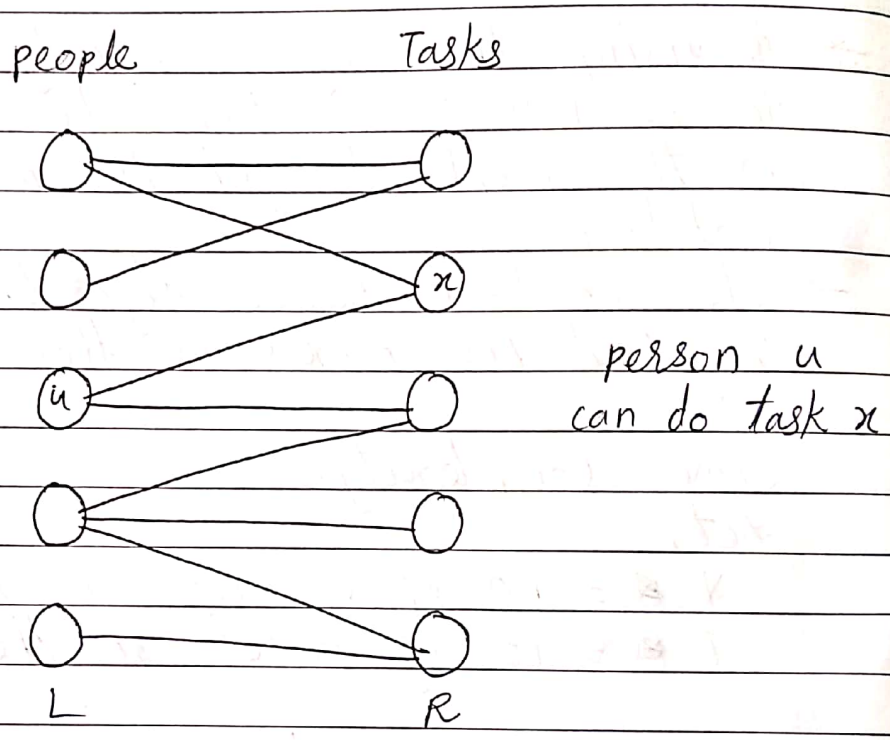
∴ Running time = O(VE).

Example:-

→ Suppose we have a set of people L and a set of jobs R.

→ Each person can do only some of the jobs.

→ This scenario can be modelled as a bipartite graph as:-

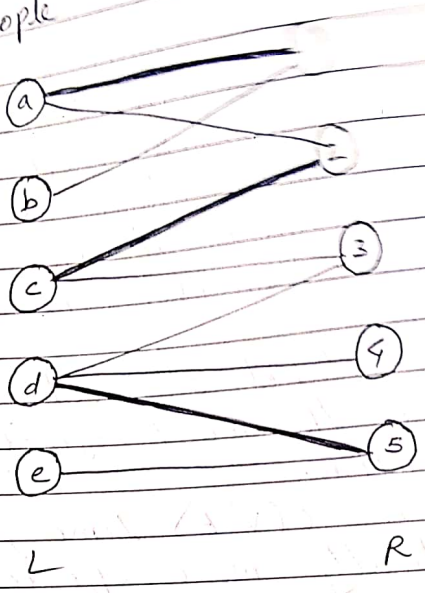


→ A matching will give an assignment of people to tasks.

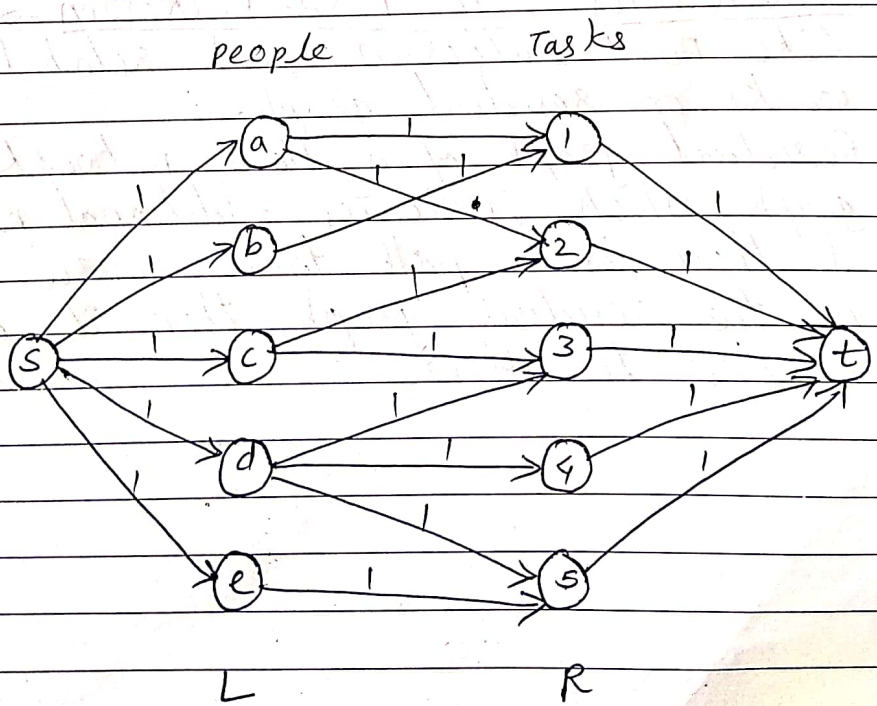
→ The aim is to get as many tasks done as possible.

→ The maximum matching has to be obtained ie. maximum matching will be the one that contains as many edges as possible.

eg) one of the



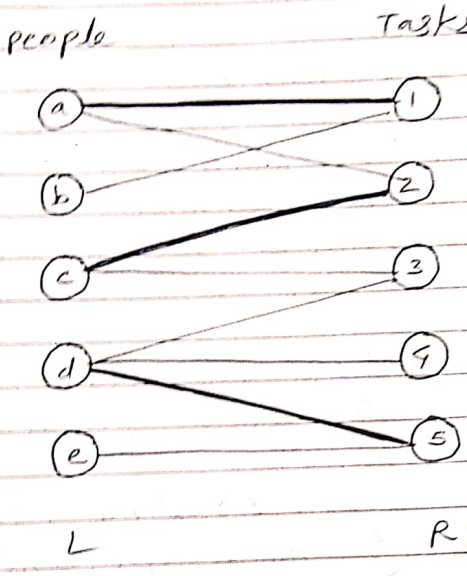
→ Reducing this bipartite graph to a flow network :-



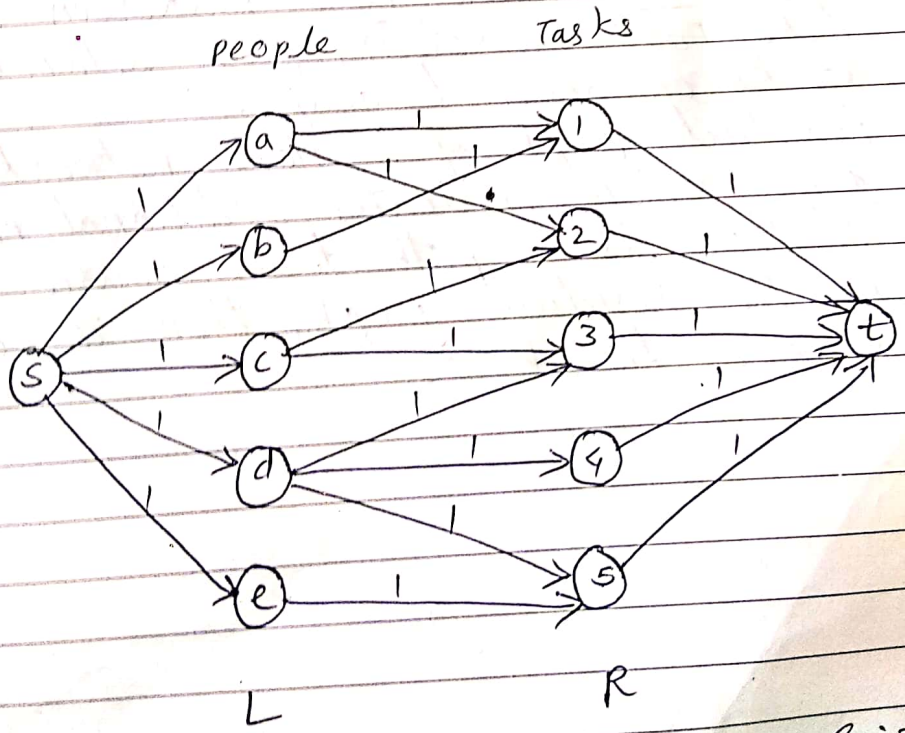
Smita Attarde

Smita Attarde

eg) One of the matching is →



→ Reducing this bipartite graph to a flow network :-



Smita Attarde

* Push - Relabel Algorithm →

→ The push - relabel algorithm (also called as preflow - push algorithm) is used to compute the maximum flows.

→ The name "push - relabel" comes from the two basic operations used in the algorithm.

→ The algorithm maintains a "preflow" and gradually converts it into a maximum flow using the push and relabel operations.

→ The push - relabel algorithm is considered as one of the most efficient maximum flow algorithm.

→ It runs in $O(V^2E)$ time, which is asymptotically more efficient than $O(E^2V)$, which is the complexity of Ford - Fulkerson algorithm.

* Similarities with Ford - Fulkerson →

→ Like Ford - Fulkerson, push relabel also works on residual graph.

→ Residual graph of a flow network is a graph which indicates additional possible flows.

→ If there is a path from source to sink in the residual graph, then it is possible to add the flow.

* Differences with Ford-Fulkerson →

1) Push relabel algorithm works in a more localized manner ie. rather than examining the entire residual network to find an augmented path, push relabel algorithm works on one vertex at a time.
2) In Ford-Fulkerson, net difference between total outflow and total inflow for every vertex (except source and sink) is maintained zero. Push-relabel algorithm allows inflow to exceed the outflow before reaching the final flow. In final flow, the net difference is zero for all except source and sink.
3) Time complexity of push-relabel is more efficient than Ford-Fulkerson

push-relabel complexity is $O(V^2E)$
Ford-Fulkerson complexity is $O(E^2V)$.

→ The concept behind the push-relabel algorithm (considering a fluid flow problem) is that, we consider edges as water pipes and nodes as joints.
→ The source is considered to be at the highest level and it sends water to all the adjacent nodes.
→ Once a node has excess water, it pushes water to a smaller height node.

Smita Attarde

→ If water gets locally trapped at a vertex, the vertex is relabeled, which means its height is increased
→ Thus, each vertex has associated with it a height variable and a excess flow.
→ Height is used to determine whether a vertex can push flow to an adjacent vertex or not.
[A vertex can push flow only to a smaller height vertex]

* Excess flow:-
It is the difference of total flow coming into the vertex minus the total flow going out of the vertex.

∴ Excess flow of u = Total inflow to u − Total outflow from u.

→ Like Ford-Fulkerson, each edge has associated to it a flow and a capacity.

push-Relabel Algorithm →

1) Initialize preflow.
2) While it is possible to perform a push() or relabel() on a vertex.
ie. While there is a vertex that has excess flow -
{
Do push()
or
relabel()
}
3) Return flow.

There are three main operations in the push-relabel algorithm:-

1) Initialize preflow operation:-
a) Initialize height and flow of every vertex as zero.
b) Initialize height of source vertex equal to the total number of vertices.
c) Initialize flow of every edge as zero
d) Initially, for all vertices adjacent to source s, flow and excess flow is equal to the capacity.

2) push operation:-
→ It is used to make the flow from a node which has excess flow.
→ If a vertex has excess flow and there is an adjacent vertex with smaller height in the residual graph, then we push the flow from the vertex to the adjacent vertex with lower height.
→ The amount of pushed flow through the pipe (edge) is equal to the minimum of excess flow and capacity of the edge.

3) Relabel operation:-
→ It is used when a vertex has excess flow and none of its adjacent is at lower height.
→ To perform push operation, the height of the vertex is increased.
→ To increase height, we pick the minimum height adjacent in the residual graph ie. an adjacent to whom we can add flow, and add 1 to it.

Relabel-to-front algorithm →

→ The push-relabel algorithm applies the basic operations in any order.
→ By choosing the order carefully and managing the network data structure efficiently, the maximum flow problem can be solved faster than $O(V^2E)$.
→ The relabel-to-front algorithm is a push-relabel algorithm whose running time is $O(V^3)$, which is asymptotically at least as good as $O(V^2E)$ and even better for dense networks.

→ The relabel-to-front algorithm maintains a list of vertices in the network.
→ Beginning at the front, the algorithm scans the list, repeatedly selecting an over-flowing vertex u, and then "discharging" it ie. performing push and relabel operations until u has no longer a positive excess flow.
→ Whenever a vertex is relabeled, it is moved to the front of the list. Hence the name "relabel-to-front" and the algorithm begins its scan.
→ The analysis of the relabel-to-front algorithm depends on the "admissible" edges ie. those edges in the residual network through which the flow can be pushed.
→ The admissible network consists of those edges through which we can push flow.

* Admissible edges and networks →

Let $G(V, E)$ is a flow network with source s and sink t.
f is a preflow in G
h is a height function.

Then, we say that,
(u, v) is an admissible edge, if
$c_f(u, v) > 0$ and $h(u) = h(v) + 1$.

Else (u, v) is inadmissible

[c_f is the current flow & h is the height]

Algorithm →

→ In the relabel-to-front algorithm, we maintain a linked list L consisting of all vertices in $V - \{s, t\}$

→ The vertices in L are topologically sorted according to the admissible network.

→ Let N[u] be the neighbour list already created for each vertex u.

→ Let next[u] points to the vertex that is immediately after u in the list L.
next[u] = Nil, if u is the last vertex in the list.

```
Relabel-to-front (G, s, t)
{
    Initialize_preflow (G, s)
    L ← V[G] - {s, t} in any order
    for each vertex u ∈ V[G] - {s, t}
    {
        current[u] = head[N[u]]
        u = head[L]
        while u ≠ Nil
        {
            old_height = h[u]
            discharge (u)
            if h[u] > old_height, then
                move u to the front of list L
            u = next[u]
        }
    }
}
```

→ Here, discharge function performs the basic push and relabel operations on a vertex until that vertex has a excess flow.

Analysis:-

→ The relabel-to-front algorithm runs in $O(V^3)$ time, which is better than $O(V^2E)$, which is the running time complexity of push-relabel algorithm.