Module ④

All-Pair Shortest Path & MaxFlow

- All pair shortest path
- Floyd warshall algorithm
- Transitive closure
- Johnson algorithm
- Arbitrage
- max flow
  - Flow N/w
  - Residual graph
  - Augmenting path
- Ford Fulkerson

All pair shortest Path (APSP):

- Finding the shortest path bet^n every pair of vertices in weighted graph.
- Finding shortest distance bet^n every pair of vertices/nodes
- We typically want o/p in matrix format.

⊙ We can solve an APSP problem by running a SSSP algorithm |V| times, once for each vertex as a source.

⊙ If all edge weights are non-negative, we can use Dijkstra's algorithm. ($O(V.E \log V)$)

⊙ If negative edge weights are allowed, Dijkstra's algorithm can no longer be used.

⊙ Instead, we can use Bellman-Ford algorithm from each vertex, giving $O(V^3)$ complexity (which can be as bad as $O(V^4)$.

Most of previous algorithms, however, will use an adjacency-matrix rather than the adjacency-list representation.

- So, suppose vertices are 1, 2, 3 ....., n
  So, input is an n x n matrix
  Wij representing edge weights.

$$W_{ij} = \begin{cases} 0 & \text{if } i = j \\ W \text{ of } (i,j) & \text{if } i \neq j \text{ \& there is direct edge} \\ \infty & \text{if } i \neq j \text{ \& No direct edge.} \end{cases}$$

- Now output matrix $D = d_{ij}$
  where d_{ij} is shortest path's weight from vertex i to vertex j.

- We will also compute a predecessor matrix $(\Pi) = \pi_{ij}$
  where,
  $\pi_{ij} = x$ ↳ Last Node visited before reaching to destination
  (π_{ij}: source, destination)

  if $i = j$ then
  $\pi_{ij} = NIL$ — same src & destination
  else
  $\pi_{ij} = x$ — i.e predecessor of j (last node visited before reaching j)

- Similar to the predecessor subgraph of SSSP, we define a predecessor subgraph for source i

as

$G_{\pi,i} = (V_{\pi,i}, E_{\pi,i})$ where

this subgraph is derived from predecessor matrix for source vertex i.

Where

- $V_{\pi,i} = \{ j \in V : \pi_{ij} \neq NIL \} + i$

  set consist of all vertices j where, π_{ij} Not NIL except i itself

- $E_{\pi,i} = \{ (\pi_{ij}, j) : j \in V_{\pi,i} - i \}$

  Set of pair of vertices, where j belongs to $V_{\pi,i}$ ensuring that doesn't include i itself.

```
PRINT_ALL_PAIR_SHORTEST_PATH
    if (i = j)
        print "i
    else if  πij = NIL
        print  "No path"
    else
        PRINT_ALL_PAIR_SHORTEST_PATH (π, i, πij)
        print j
```

Dynamic Programming approach / Principles :

1) Structure of an optimal soln:

if k is the predecessor of j on shortest path from i to j, then

$$d(i,j) = d(i,k) + w_{kj}$$

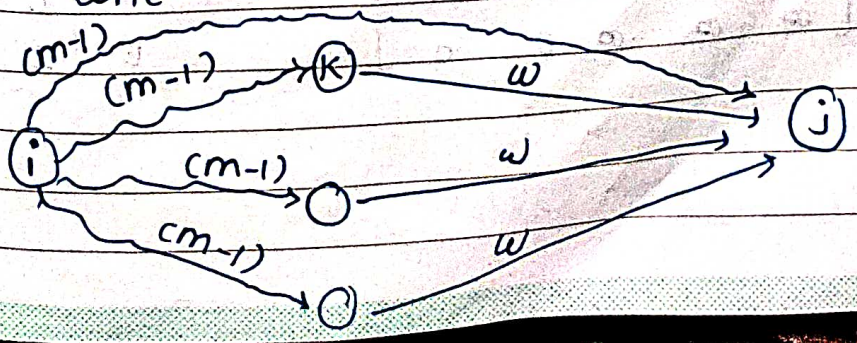
$d(i,k)$ — distance from i to k; w_{kj} — weight from k to j

2) Recursively defining the value of the optimal soln:

Let $l_{ij}^{(m)}$ be the min distance/weight of any path from vertex i to j that contains at most m edges

- $l_{ij}^{(m)} = 0$ — if $i = j$
- $l_{ij}^{(0)} = \infty$ — if $i \neq j$ — (Here m = 0 as there is no direct edge betn i to j)
- $l_{ij}^{(m)} = \min \{ l_{ik}^{(m-1)} + w_{kj} \}$

where $m = 1, 2, 3 \ldots\ldots V-1$



3) Computing the value of an optimal Soln bottom up :

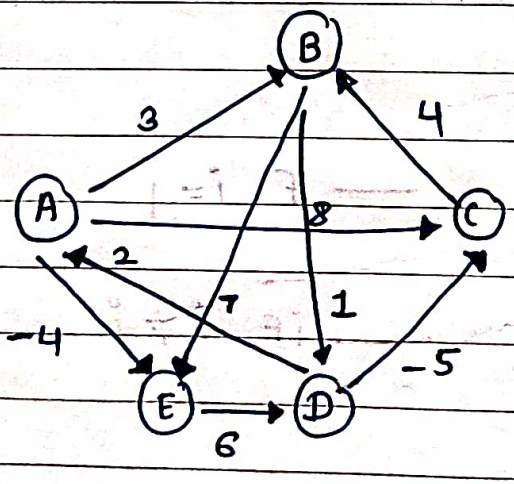
- From input matrix $W = (w_{ij})$
- compute matrices $L^{(1)}, L^{(2)}, L^{(3)} \ldots L^{(n-1)}$ where

$$L^{(m)} = \left(l_{ij}^{(m)}\right) \rightarrow \text{From (2)}$$

- $L^{(n-1)}$ will be the resultant matrix with shortest paths.
- $L^{(1)}$ will be original W

it extends shortest path computed so far by one more edge.

Example



$L1 =$

	A	B	C	D	E
A	0	3	8	∞	-4
B	∞	0	∞	1	7
C	∞	4	0	∞	∞
D	2	∞	-5	0	∞
E	∞	∞	∞	6	0

$\Rightarrow$ $L2 =$

	A	B	C	D	E
A	0	3	8	2	-4
B	3	0	-4	1	7
C	∞	4	0	5	11
D	2	-1	-5	0	-2
E	8	∞	1	6	0

$L_3 =$

	A	B	C	D	E
A	0	1	-3	2	-4
B	3	0	-4	1	-1
C	7	4	0	5	11
D	2	-1	-5	0	-2
E	8	5	1	6	0

$L_4 =$

	A	B	C	D	E
A	0	1	-3	2	-4
B	3	0	-4	1	-1
C	7	4	0	5	3
D	2	-1	-5	0	-2
E	8	5	1	6	0

suppose,

In order to calculate
A to D in L_2 using L_1

find

(A to A) + (A to D)
(A to B) + (B to D)
(A to C) + (B to D)
(A to D) + (D to D)
(A to E) + (E to D)

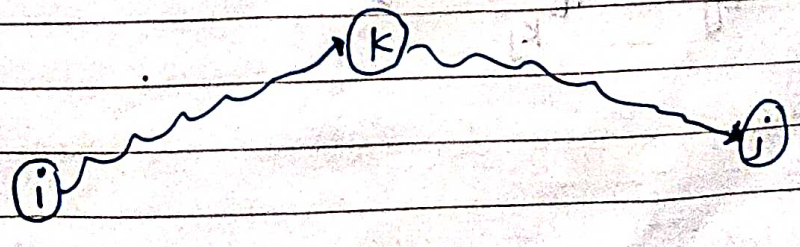
} take min from this & update matrix

complexity : $\theta(n^3)$

if we use repeated squaring = $n^2 \log n$

Floyd Warshall algorithm

- It is APSP algorithm
- Used to find shortest path distances.
- Works for directed / undirected graph.
- Don't work for negative weight cycle graph.
- Complexity = $O(V^3)$



Let all intermediate vertices in {1, 2, 3 ____ k-1}

Suppose,

→ k is the highest intermediate vertex betn i to j

then

$$d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$$

→ if $(d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$

then $\pi_{ij}^{(k)} = \pi_{kj}^{(k-1)}$

For Floywarshall algo:

Input = Weighted graph

output =

① D matrix

② π matrix

① D matrix : start from $D^0, D^1, \ldots D^n$

SP lengths

if (i=j)

dij = 0

if (i ≠ j)

$d_{ij} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$

② π matrix : starts from $\pi^0, \pi^1, \ldots \pi^n$

predecessor matrix

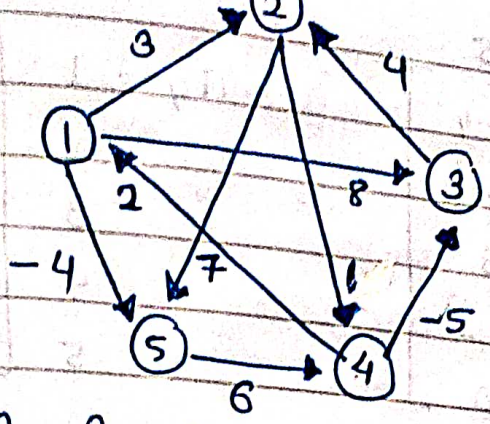
if (i=j)

π_{ij} = NIL

if $(d_{ij}^{(k-1)} > d_{ij}^{(k-1)} + d_{kj}^{(k-1)})$

$\pi_{ij} = \pi_{kj}^{(k-1)}$

Example :



$D_0 =$

	1	2	3	4	5
1	0	3	8	∞	-4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	∞	-5	0	∞
5	∞	∞	∞	6	0

$\Pi_0 =$

	1	2	3	4	5
1	NIL	1	1	NIL	1
2	NIL	NIL	NIL	2	2
3	NIL	3	NIL	NIL	NIL
4	4	NIL	4	NIL	NIL
5	NIL	NIL	NIL	5	NIL

$D_1 =$

	1	2	3	4	5
1	0	3	8	∞	-4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	(5)	-5	0	(-2)
5	∞	∞	∞	6	0

$\Pi_1 =$

	1	2	3	4	5
1	NIL	1	1	NIL	1
2	NIL	NIL	NIL	2	2
3	NIL	3	NIL	NIL	NIL
4	4	1	4	NIL	1
5	NIL	NIL	NIL	5	NIL

$D_2 =$

	1	2	3	4	5
1	0	3	8	(4)	-4
2	∞	0	∞	1	7
3	∞	4	0	(5)	(11)
4	2	5	-5	0	-2
5	∞	∞	∞	6	0

$\Pi_2 =$

	1	2	3	4	5
1	NIL	1	1	2	1
2	NIL	NIL	NIL	2	2
3	NIL	3	NIL	2	2
4	4	1	4	NIL	1
5	NIL	NIL	NIL	5	NIL

$D_3 =$

	1	2	3	4	5
1	0	3	8	4	-4
2	∞	0	∞	1	7
3	∞	4	0	5	11
4	2	-1	-5	0	-2
5	∞	∞	∞	6	0

$\Pi_3 =$

	1	2	3	4	5
1	NIL	1	1	2	1
2	NIL	NIL	NIL	2	2
3	NIL	3	NIL	2	2
4	4	3	4	NIL	1
5	NIL	NIL	NIL	5	NIL

$D_4 =$

	1	2	3	4	5
1	0	3	-1	4	-4
2	3	0	-4	1	-1
3	7	4	0	5	3
4	2	-1	-5	0	-2
5	8	5	1	6	0

$\Pi_4 =$

	1	2	3	4	5
1	NIL	1	4	2	1
2	4	NIL	4	2	1
3	4	3	NIL	2	1
4	4	3	4	NIL	1
5	4	3	4	5	NIL

$D_5 =$

	1	2	3	4	5
1	0	1	-3	2	-4
2	3	0	-4	1	-1
3	7	4	0	5	3
4	2	-1	-5	0	-2
5	8	5	1	6	0

$\Pi_5 =$

	1	2	3	4	5
1	NIL	3	4	5	1
2	4	NIL	4	2	1
3	4	3	NIL	2	1
4	4	3	4	NIL	1
5	4	3	4	5	NIL

▶ Transitive closure :

Transitive closure of a directed graph $G(V, E)$ is defined as graph $G^* = (V, E^*)$ where

$E^* = \{(i, j) :$ there is path from i to $j\}$

- Ways to compute transitive closure

① Changing min and + in floyd-warshall to logical OR ($\vee$) and logical AND ($\wedge$). using boolean bit (1) and (0)

We'll get $T^{(k)}$ as transitive closure's result

$T^{(k)} = t_{ij}^{(k)}$

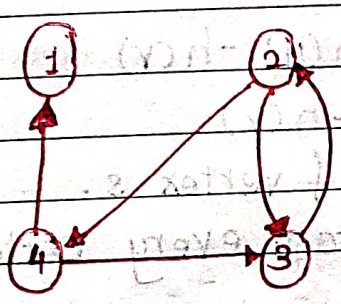
Calculate $T^0, T^1 \_\_\_\_ T^n$

$t_{ij}^{(k)} = 1$ — if there is path from i to j with all intermediate vertices in V_k

initial → $t_{ij}^{(0)} = \begin{cases} 1 & \text{— if } i=j \text{ or } (i,j) \in E \\ 0 & \text{— otherwise} \end{cases}$

then $t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)})$

Example



$\vee$ (OR)			$\wedge$ (AND)		
1	1	1	1	1	1
1	0	1	1	0	0
0	1	1	0	1	0
0	0	0	0	0	0

T_0 =

	1	2	3	4
1	1	0	0	0
2	0	1	1	1
3	0	1	1	0
4	1	0	1	1

T_1 =

	1	2	3	4
1	1	0	0	0
2	0	1	1	1
3	0	1	1	0
4	1	0	1	1

T_2

	1	2	3	4
1	1	0	0	0
2	0	1	1	1
3	0	1	1	1
4	1	0	1	1

T_3 =

	1	2	3	4
1	1	0	0	0
2	0	1	1	1
3	0	1	1	1
4	1	1	1	1

T_4 =

	1	2	3	4
1	1	0	0	0
2	1	1	1	1
3	1	1	1	1
4	1	1	1	1

Johnson Algorithm:

- It is All pair shortest path algorithm
- Used for sparse graph (Not Fully connected graph is called sparse graph)
- Works with the help of
  - → Bellman-Ford Algo
  - → Dijkstra's Algo

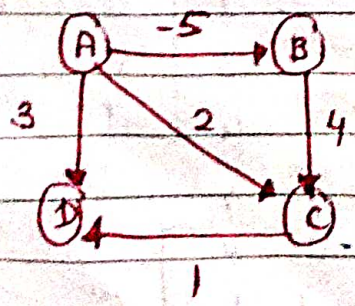
Steps

1. Add new vertex s
2. Add edges from s to all other remaining vertices.
3. Apply Bellman-Ford algorithm & calculate h(0), h(1)...
4. Reweight :
   - = Original weight + h(y) - h(v)
   - = w(u,v) + h(u) - h(v)
5. Remove added edges & vertex s.
6. Run Dijkstra's algo from every vertex.

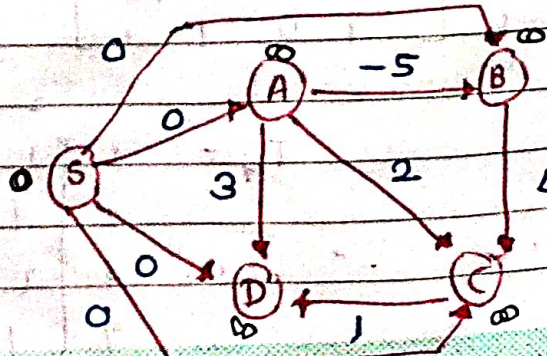
* Johnson is best for sparse graph as its complexity depends on no. of edges in graph.

$$O(V^2 \log(V)) + VE$$

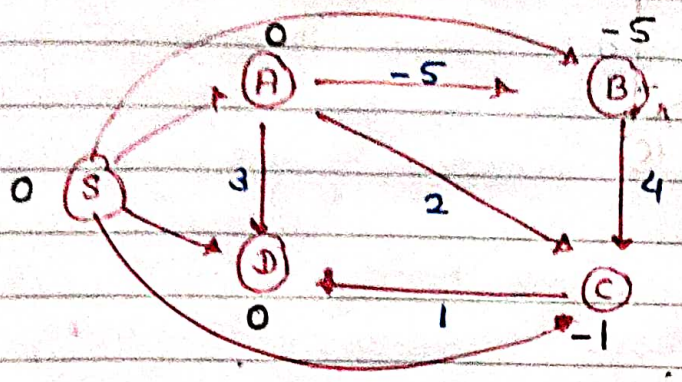
Example



0 & 2. Add s & edges from s



③ Apply bellman Ford



Now we get $h(A) = 0$

$h(B) = -5$

$h(c) = -1$

$h(D) = 0$

④ Reweight :

New $W(A,B) = w(A,B) + h(A) - h(B)$

$= -5 + 0 - (-5)$

$= -5 + 0 + 5$

- New $W(A,B) = 0$

New $W(B,C) = w(B,C) + w(B) - w(C)$

$= 4 + (-5) - (-1)$

$= 4 - 5 + 1$

- New $W(B,C) = 0$

New $W(C,D) = 1 + (-1) - 0$

$= 1 - 1 - 0 = 0$

- New $W(C,D) = 0$

New $W(A,C) = 2 + 0 - (-1)$

$= 2 + 0 + 1$

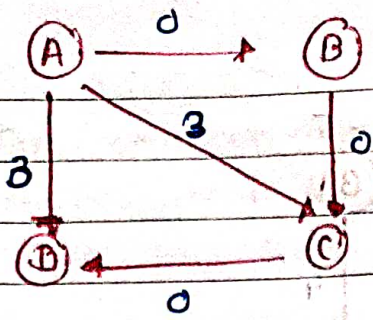
- New $W(A,C) = 3$

New $W(A,D) = 3 + 0 - 0$

$= 3$

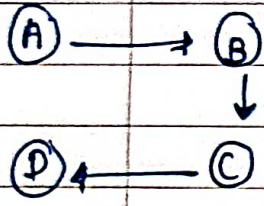
- New $W(A,D) = 3$

(5)



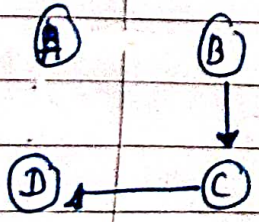
(6) Run Dijkstra :

i] A as source :



(A)	A	B	C	D	
(0)	∞	∞	∞		
		∞	∞	∞	{A}
	(0)	3	3		
			3	3	{A, B}
			(0)	3	
				3	{A, B, C}
				(0)	
				0	{A, B, C, D}

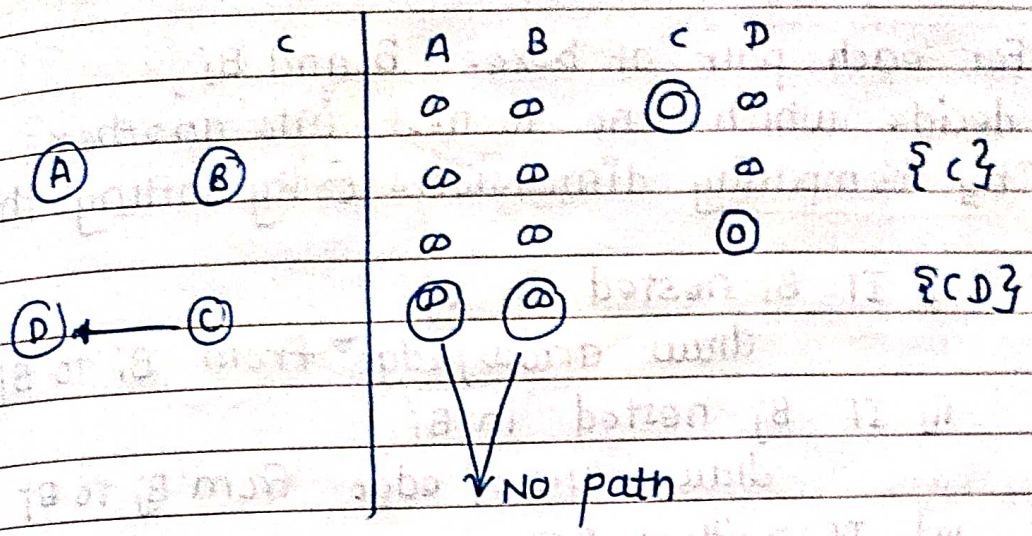
ii] B as source



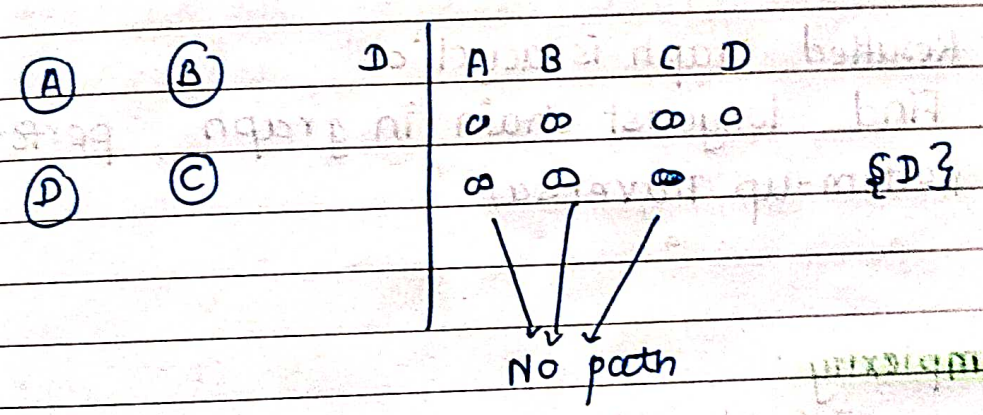
A	B	C	D	
∞	(0)	∞	∞	
∞		∞	∞	{B}
∞		(1)	∞	
∞			∞	{B C}
∞			(2)	
∞				{B C D}

No path

iii) C as source :



iv) D as source



Application of shortest path algorithm :

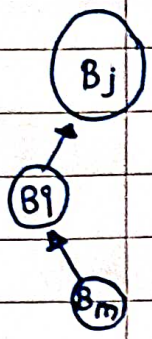
1] Nesting of boxes
2] Arbitration

1) Nesting of boxes :

Base Problem :
To check if one box can nest into another.

Soln approach :
1] Given n boxes, create a nesting graph G with boxes (vertices) $B_1, B_2 \_\_\_\_ B_n$

2) For each pair of boxes Bi and Bj, decide which one to nest into another by comparing dimensions (By sorting them)



i] IF Bi nested in Bj
draw arrow/edge from Bi to Bj

ii] IF Bj nested in Bi
draw arrow/ edge from Bj to Bi

iii] If neither nests
no arrow/edge

3) Resulted graph is acyclic

4) Find longenst chain in graph, performing bottom-up traversal

Complexity:

- Sorting dimension - $O(d \log d)$
- comparing = $O(d)$
- drawing graph: = $n^2 d$
- find chain = n^2

$\therefore$ Total = $(nd + \max(d \log d, n))$

(2) Arbritation:

- Detecting Arbritrage opportunities in currency exchange involves identifying cycles in a weighted directed directed graph
  where product of edges weights around cycle > 1

steps :

(1) $w_1 \times w_2 \times w_3 \ldots w_n > 1$ ∴ cycle detected

(2) Take log on both sides.

$\log(w_1) + \log(w_2) + \ldots \log(w_n) > \log(1)$

$\log(w_1) + \log(w_2) + \ldots \log(w_n) > 0$

(3) Take negative log

$(-\log(w_1)) + (-\log(w_2)) + \ldots (-\log(w_n)) < 0$

This implies, if we can find cycle, where the sum of -ve logarithm of edge weight is negative, then there is opportunity for currency arbitrage.

Max Flow :-

- maximum amount of flow that N/w allow to flow from source to sink.

- (Source) — produce whatever is flowing in n/w (with all outgoing edges)
- (sink) — Consumes whatever is flowing in n/w (with all incoming edges)
- (Bottleneck capacity) - min capacity of any edge within the path
- (flow) — q / b
  - q → flow
  - b → capacity
- (flow N/w) — Graph with nodes and directed edges (1) sink and (1) source and non-negative weights on edges.

Residual graph → Graph represents flow of N|w. with remaining capacity available on edges.

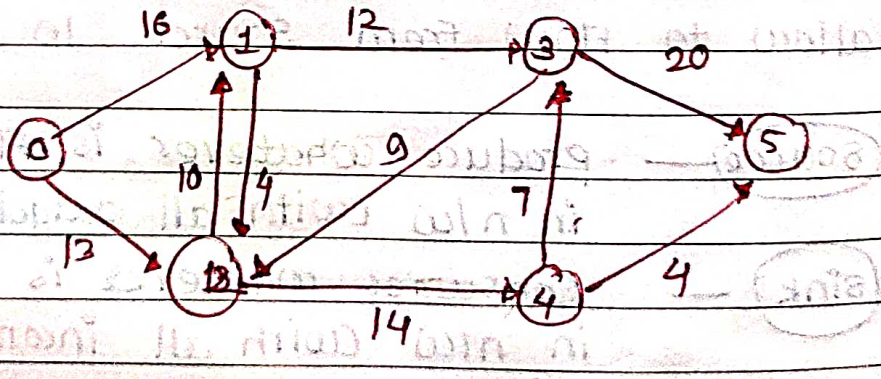
Augmenting path → simple path from src to sink in Residual graph.

Residual capacity → original capacity − current flow

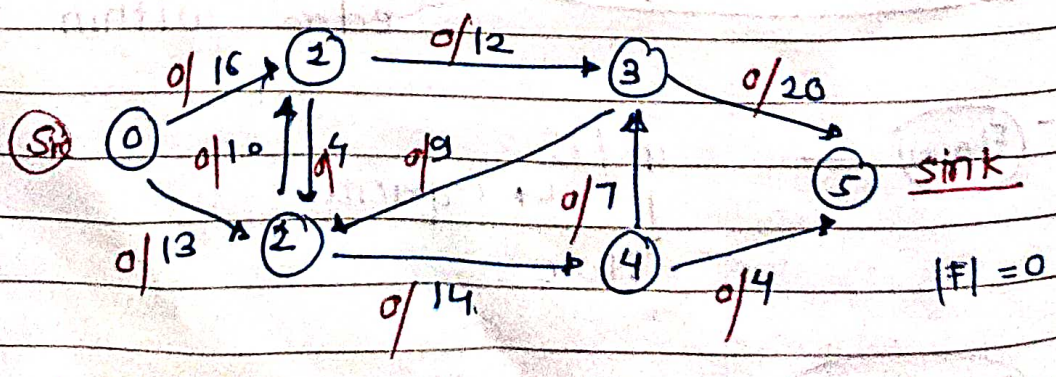
Ford fulkerson (FF) Algo :

- Greedy algo computes max-flow in Flow N|w
- Problem involves finding feasible flow through a single src to single sink that is max.
- Goal to find how much max stuff can be push from src to sink

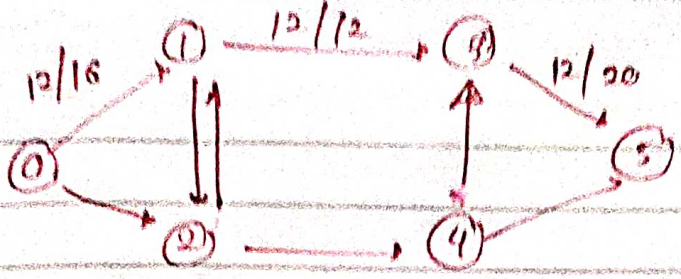
Example :



i] Initial flow should be 0



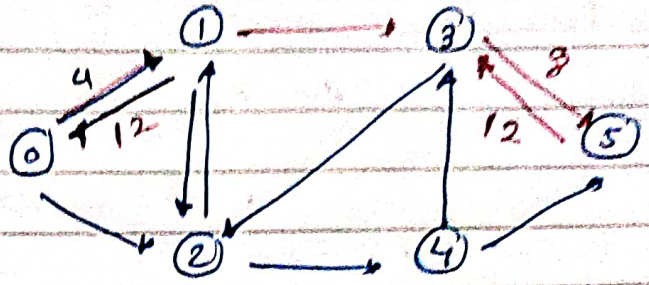
ii] - select any path from src(s) to sink(t)
- find flow



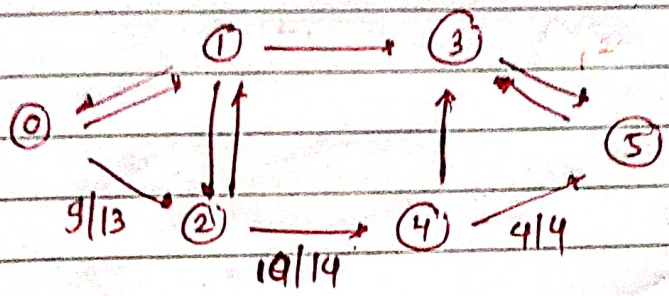
Augmented Path 2
0 - 1 - 3 - 5

$|f| = 12$

Residual graph



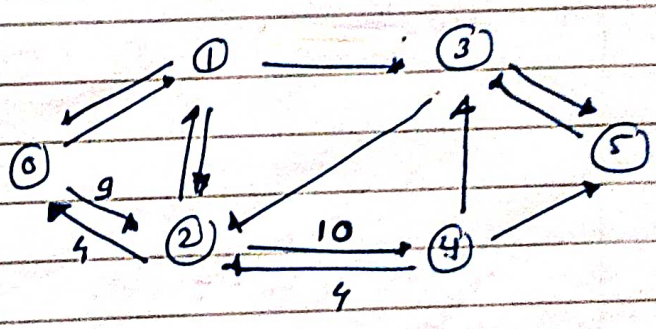
iii] add reverce edges and construct residual graph

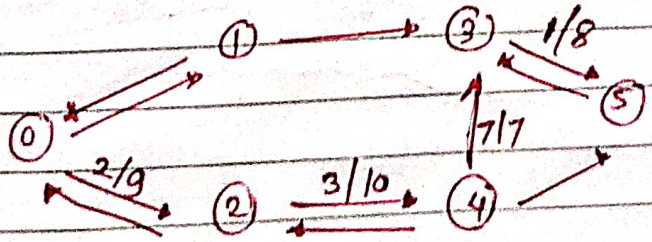


0 - 2 - 4 - 5

$|f| = 12 + 4 = 16$

Residual graph

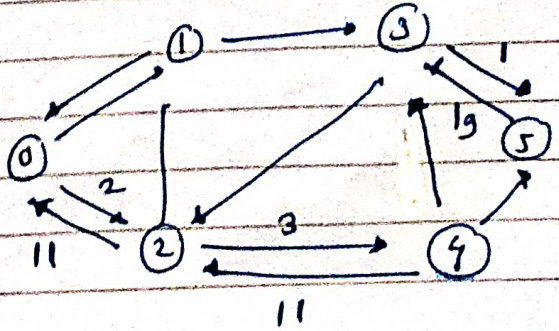




0 - 2 - 3 - 5

$|f| = 16 + 7 = 23$

→ Residual graph



∴ max flow = 23

iv) Now no agumented path step