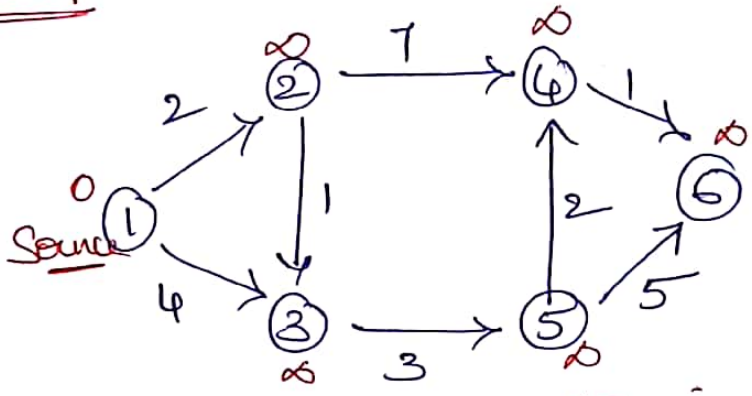
Dijkstra Algorithm

→ Finds shortest paths between single source and all other remaining vertices in a graph.

→ Works for both directed and non directed graphs.

- Big Assumption is it considers only +ve weight edges.
- Works only for connected graphs.

Example

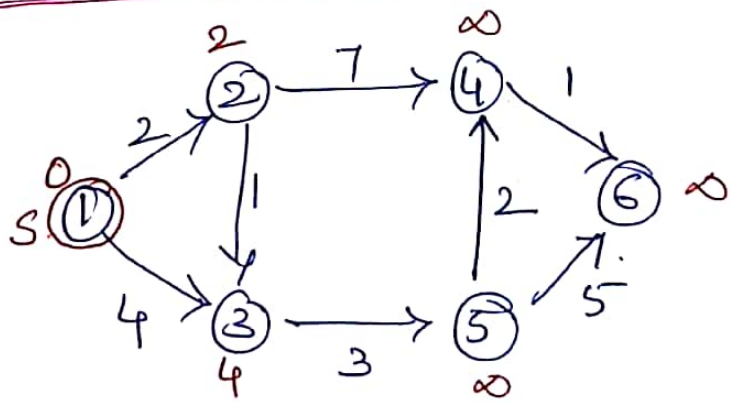


→ Assume source vertex is ①.

→ Dijkstra algorithm performs Relaxation operation on vertices. [To perform this relaxation step, it selects the vertex which has minimum distance value i.e d[v]]

Relaxation

if (d[u] + c(u,v) < d[v])
d[v] = d[u] + c(u,v).



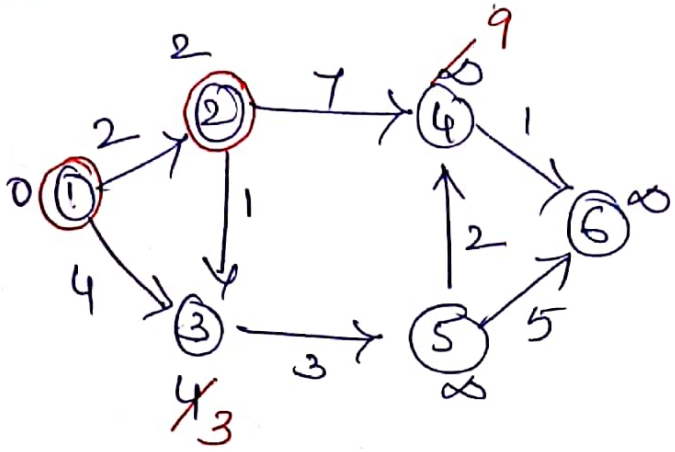
Since vertices ② & ③ are directly connected to source, their distances are

d[2] = 2

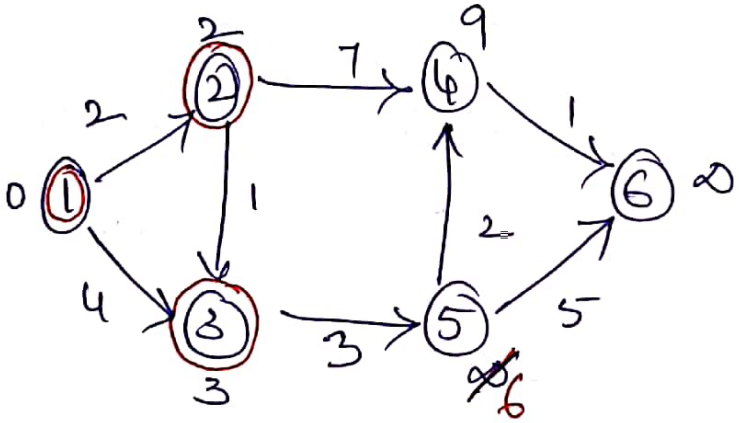
d[3] = 4.

~~for remaining vertices ∞~~.

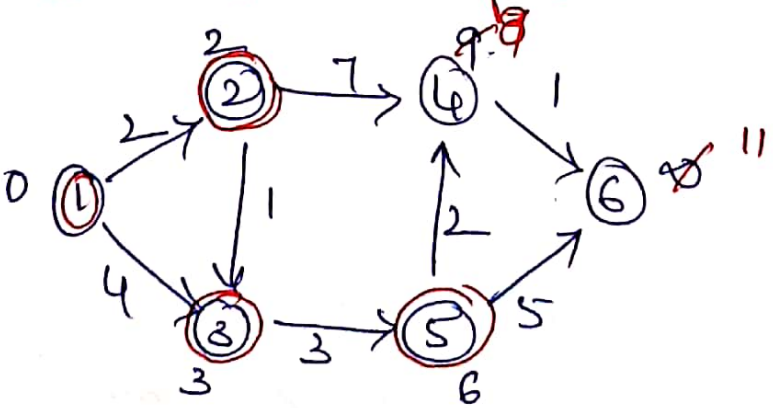
Out of all distance values of vertices, vertex ② has least distance value, i.e. d[2]=2, hence select vertex ② for relaxation step.



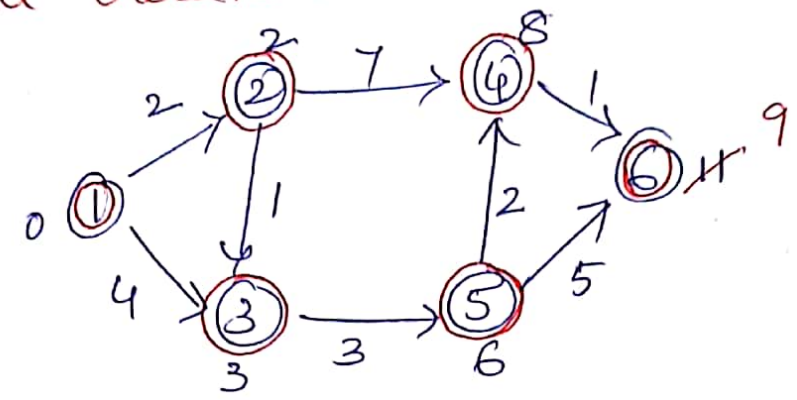
Next least distance value is with vertex ③ so perform relaxation on vertex ③.



Next vertex is ⑤ as it has least distance value of 6.



Next relax vertex ④ whose d[4] = 8.



Now from ⑥ there are no outgoing edges so it stops.

Hence shortest paths from vertex ① are

v	d[v]
2	2
3	3
4	8
5	6
6	9

Time Complexity :—

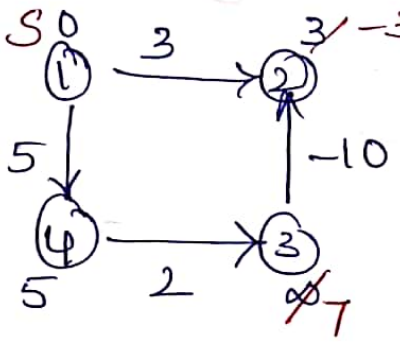
It is finding shortest paths to all vertices from source, hence it is |v| = n.
while finding shortest paths it is relaxing almost n² vertices (if graph is connected)
hence time taken by dijkstra is

$O(n^2)$ → worst case.

Drawback of Dijkstra :—

It doesn't work for the graphs which has -ve edges.

Eg



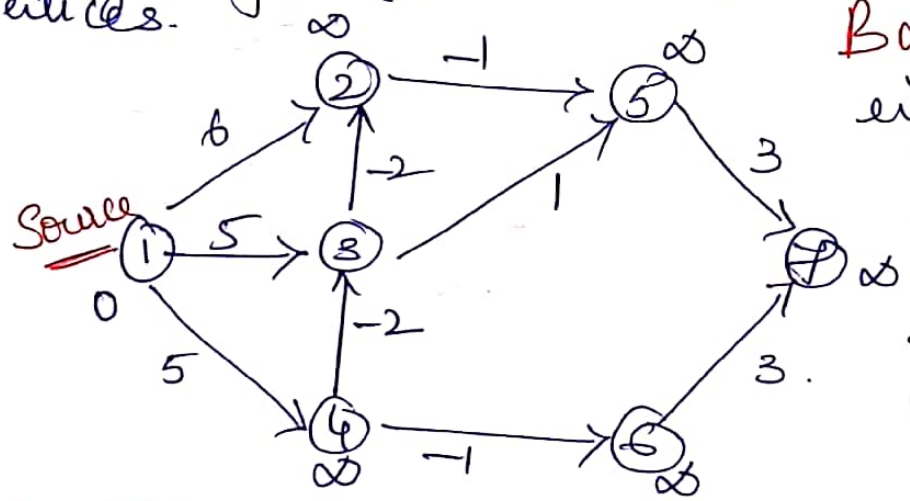
when we try to relax vertex ③ it is relaxing the vertex ② again from 3 to -3. But Dijkstra will not relax the vertices twice, so it leaves d[2] = 3, But when we consider vertex ③ for relaxation step, d[2] gets modified to still reduced value which -3, which should not happen. This is the reason Dijkstra does'nt deal with -ve edges.

Bellman-Ford Algorithm

→ It is also used for finding shortest path between single source and all other remaining vertices.

→ It can handle -ve edges.

→ In Bellman-Ford algorithm relax all ~~edges~~ vertices for $|V|-1$ times.



Basic idea is instead of picking the 'u' with smallest d[u] to relax, just relax all of u's simultaneously.

Relaxation

Consider edge (u, v)

if (d[u] + c(u,v) < d[v])

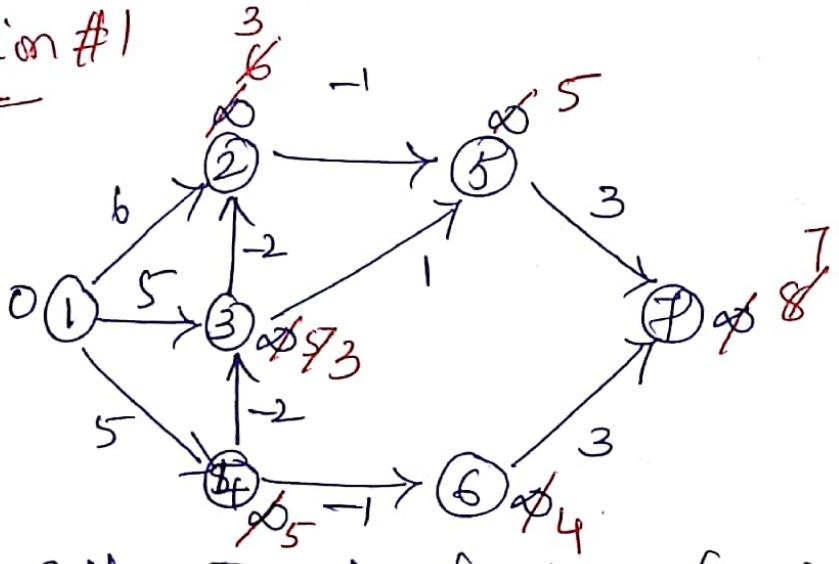
d[v] = d[u] + c(u,v)

Edge list :- (1,2), (1,3), (1,4), (2,5), (3,2), (3,5), (4,3), (4,6), (5,7), (6,7)

→ relax all these ~~edges~~ vertices for $|V|-1$ times
$= 7-1 = 6$ times.

→ Initially mark all distances values for all vertices as ∞, except for source vertex, makes it as 0 i.e d[source] = 0.

Iteration #1



Consider the edge (1,2) (1,3) (1,4)

$0+6<\infty$ ✓ $0+5<\infty$ ✓ $0+5<\infty$

$\therefore d[2]=6$ $\therefore d[3]=5$ $\therefore d[4]=5$

(2,5)

$6-1<\infty$ ✓

$5<\infty$ ✓

$\therefore d[5]=5$

(3,2)

$5-2<6$

$3<6$ ✓

$\therefore d[2]=3$

(3,5)

$5+1<5$

$6<5$ ✗

$\therefore$ don't change

(4,3)

$5-2<5$

$3<5$ ✓

$\therefore d[3]=3$

(4,6)

$5-1<\infty$

$4<\infty$ ✓

$\therefore d[6]=4$

(5,7)

$5+3<\infty$

$8<\infty$ ✓

$\therefore d[7]=8$

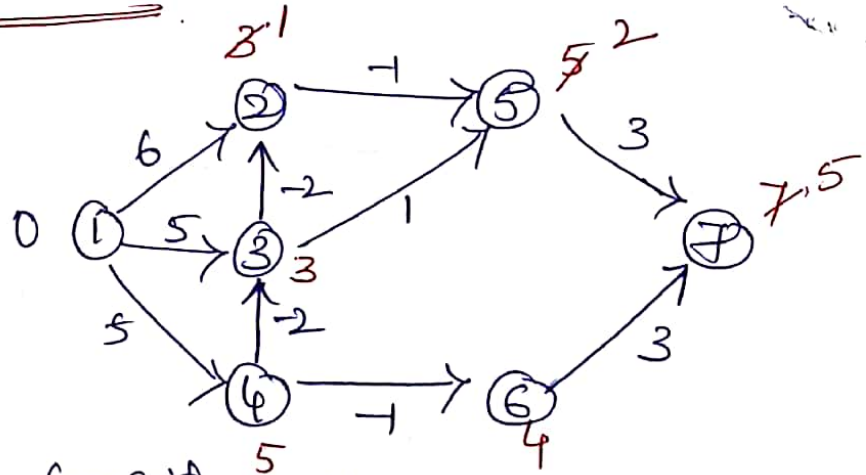
(6,7)

$4+3<8$

$7<8$ ✓

$\therefore d[7]=7$

Iteration #2



Now Consider all edges

for edges (1,2), (1,3), (1,5) No change.

Consider (2,5)
$3-1<5$
$2<5$ ✓
$\therefore d[5]=2$

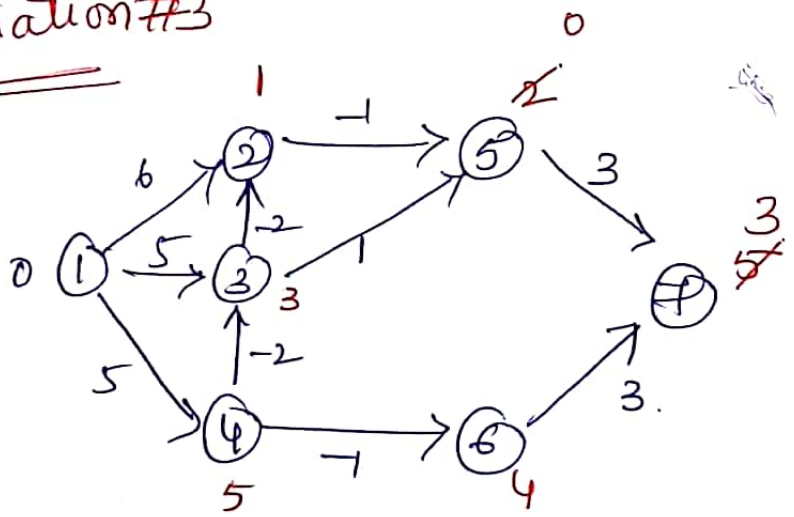
(3,2)
$3-2<3$
$1<3$ ✓
$\therefore d[2]=1$

(3,5) (4,3) (4,6)
No change.

(5,7)
$2+3<7$
$5<7$ ✓
$\therefore d[7]=5$

(6,7)
No change.

Iteration #3



(1,2), (1,3) (1,4) → No change

(2,5)
$1-1<2$
$0<2$ ✓
$\therefore d[5]=0$

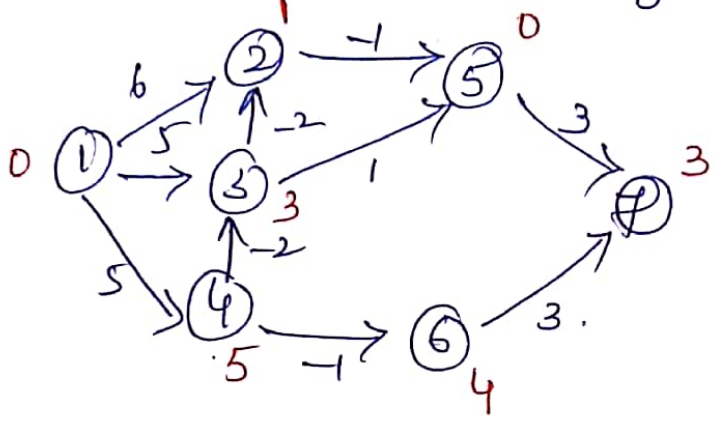
(3,2) (3,5) (4,3) (4,6)
No change

(5,7)
$0+3<5$
$3<5$ ✓
$\therefore d[7]=3$

(6,7)
No change.

Iteration #4

(1,2)(1,3),(1,4),(2,5)(3,2),(3,5),(4,3),
(4,6),(5,7),(6,7) all edges
will not change.



Hence shortest paths from (1).

v	d[v]
1	0
2	1
3	3
4	5
5	0
6	4
7	3

Time Complexity

There |E| edges, it is performing |V|-1 times relaxations.

$\therefore O(|V||E|)$

if Graph is complete graph

$|E| = n^2$

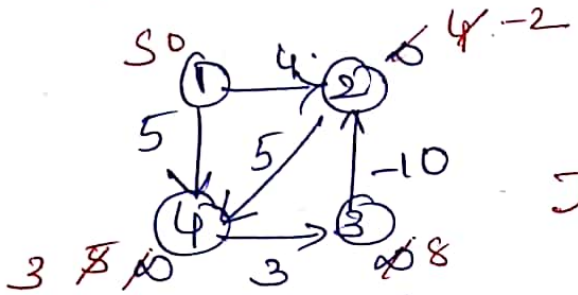
$\therefore O(n^3) \rightarrow$ worst case.

Difference b/w Dijkstra & Bellman Ford

Main difference b/w Dijkstra & Bellman-Ford is Dijkstra never look back once the shortest path is computed to a vertex, But Bellman-Ford revisits the vertex to reduce the cost for shorter path, i.e each vertex is relaxed more than one time i.e it relaxes |V|-1 times, where as in Dijkstra a vertex is relaxed OnlyOnce.

Drawback of Bellman-Ford

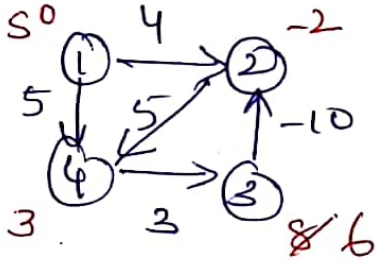
Consider the graph



This after $|V|-1 = 4-1 = 3$ times.

edge list (3,2)(4,3)(1,4)(1,2)(2,4)

Let's do one more iteration



→ d[3] is getting changed to 6 even after 3 iterations.

With -ve weight cycle we can't define shortest paths.

[Go one more round more, and see if distances changes]

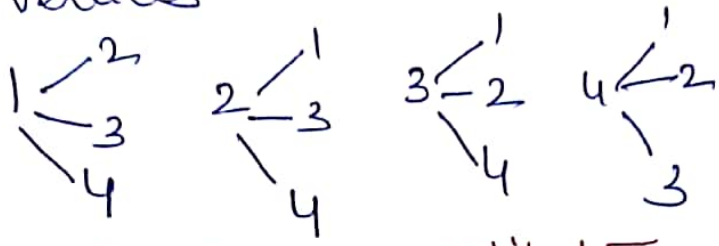
If you do one more iteration some distances again still reduces, which should not happen.

∴ Bellman-Ford fails to detect shortest path if graph contains -ve cycle [Repeatedly traversing the -ve cycle pushes cost without any bound].

And Bellman-Ford alg can be used to detect -ve cycle in the graph i.e if distances gets changed even after $|V|-1$ iterations, then graph contains -ve cycle. [d[v] values will keep on changing if -ve cycle is present].

All pairs shortest-paths Problem
Floyd-Warshall Algorithm

→ Find shortest path between every pair of vertices.



what if I run dijkstra on all vertices, which find shortest-paths b/w allpairs.

but- dijkstra takes $O(n^2)$ time

If I run dijkstra for all vertices i.e n

then it takes $O(n^3)$ time

Can we do better than this ! ☺

Floyd-Warshall gives the better solution

Floyd-Warshall computes the shortest paths as:-

① First it will compute shortest paths b/w all pairs, by considering vertex ① as intermediate node.

② Next it computes, by considering vertex ① & ② as intermediate node

③ Next it computes shortest paths b/w all pairs, by considering vertex ③ as the intermediate node.

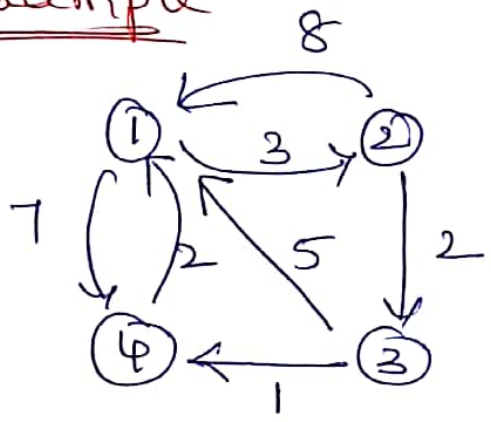
(1,2,3)

⋮

So on.

[This is how we take sequence of decisions, at each step, the algorithm makes use of result of previous step → Dynamic Programming]

Example



A^0 = original weight matrix

	1	2	3	4
1	0	3	∞	7
2	8	0	2	15
3	5	8	0	1
4	7	8	∞	0

if there exists direct edge b/w u & v, then weight is taken, otherwise it is ∞.

Step 1 Consider vertex ① as intermediate node, to compute A^1.

Since vertex ① is intermediate node row 1 & col 1 will not change

Now compute $A^1[2,3]$ from A^0.

A^1 =

	1	2	3	4
1	0	3	∞	7
2	8	0	(2)	(15)
3	5		0	
4	7			0

$\therefore A^1[2,3] = \min\{A^0[2,3], A^0[2,1] + A^0[1,3]\}$

$= \min\{2, 8+\infty\}$

$= 2$

$\therefore$ no shortest path via ①

$A^1[2,3] = 2$ only.

$A^1[2,4] = \min\{A^0[2,4], A^0[2,1] + A^0[1,4]\}$

$= \min\{\infty, 8+7\}$

$= 15$

$\therefore A^1[2,4] = 15$

$\therefore$ Shortest path b/w 2 & 4 is via (1)

⋮

So on.

Finally we get-

$$A^1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & 2 & 15 \\ 5 & 8 & 0 & 1 \\ 7 & 8 & \infty & 0 \end{bmatrix} \end{array}$$

Step 2

Now Compute A^2 from A^1 i.e consider vertices (1) & (2) as intermediate nodes.

$$A^2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & 5 & 7 \\ 8 & 0 & 2 & 15 \\ 8 & 0 & & \\ 8 & & & 0 \end{bmatrix} \end{array}$$

2nd row 2nd col no change as vertex (2) itself intermediate node.

$A^2[1,3] = \min\{A^1[1,3], A^1[1,2] + A^1[2,3]\}$

$= \min\{\infty, 3+2\}$

$= 5$

$A^2[1,4] = \min\{A^1[1,4], A^1[1,2], A^1[2,4]\}$

$= \min\{7, 3+15\}$

$= 7$. No change.

Hence $A2 =$

	1	2	3	4
1	0	3	5	7
2	8	0	2	15
3	5	8	0	1
4	2	5	7	0

Step 3 Consider vertices ①, ② & ③ as intermediate vertices.

we get

$A3 =$

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

Step 4 Vertex ④ also as intermediate node

we get

$A4 =$

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

Hence the equation for optimal solution is

$$A^k[i,j] = \min\{A^{k-1}[i,j], A^{k-1}[i,k] + A^{k-1}[k,j]\}$$

Time Complexity

To compute matrix at every step it takes n^2 time and there are total 'n' steps.

$\therefore O(n^3)$

[Floyd Warshall can also handle -ve edges].

→ Bellman-Ford generalizes Dijkstra's
Floyd-Warshall further generalizes, such
that not from only one vertex, it
computes shortest paths b/w every
pair of vertices.

Network Flows

Ford-Fulkerson Algorithm

Finds maximum flow value in a given flow network.

Problem Given a graph, which represents flow network, in which every edge has a capacity. Also given two vertices source s and sink t in the graph, the algorithm tries to find out max possible flow from s to t with following constraints.

Constraints :-

1. Flow on any edge should not exceed the given capacity.
2. In-flow must be equal to out-flow for every vertex except for s & t.

Algorithm :-

The following is the single idea of the Algorithm.

1. Start with initial flow as 0.
2. While there is an augmenting path from s to t
   Add this path flow to flow.
3. Return Flow.

Residual Graph :— It is a graph which indicates additional possible flow. If there is exists such path in residual graph from s to t, then there is a possibility to add flow.

Residual Capacity :—

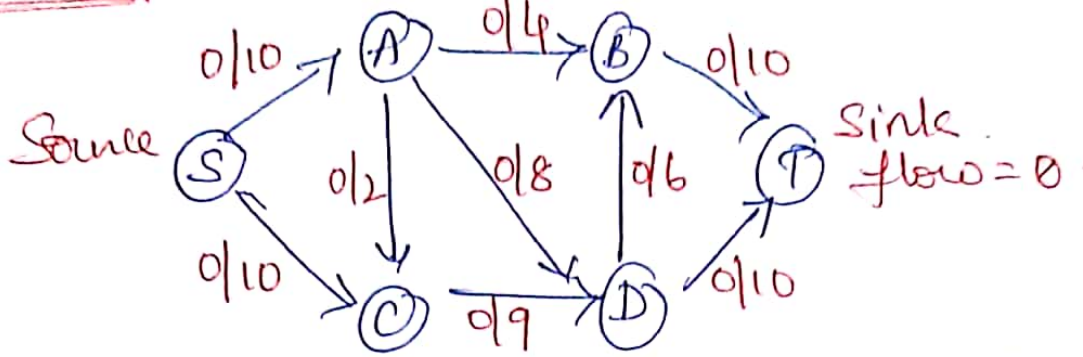
= Original Capacity − Flow.

Minimal Cut :— Also known as **Bottleneck Capacity** which decides max possible flow from s to t through augmented path.

Augmented Path :— Can be created in two ways.

1. Non-Full forward edges.
2. Non-Empty backward edges.

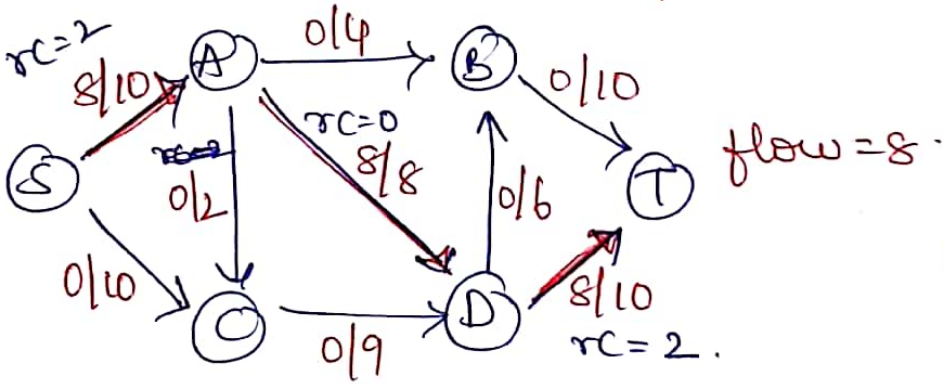
Example



Initially consider flow = 0 as in the algorithm.

→ Now select one augmenting path.

S →(10) A →(8) D →(10) T. it has Bottleneck capacity = 8.

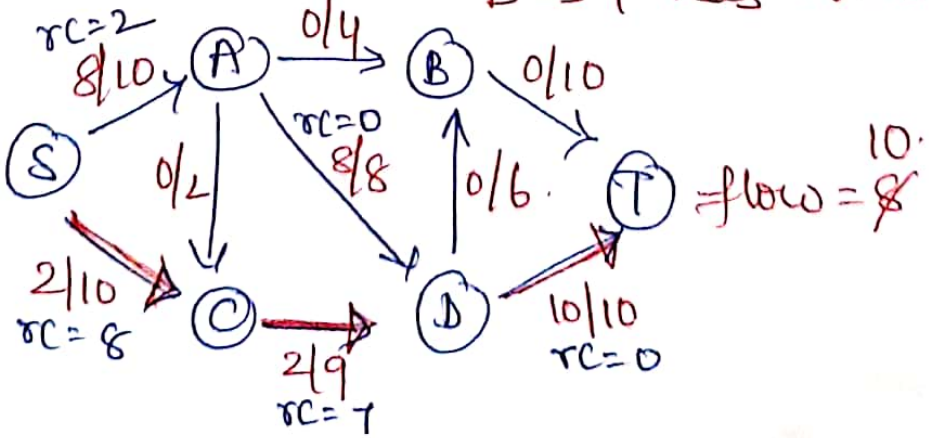


rc - residual capacity.

→ Now select another augmenting path arbitrarily.

S →(10) C →(9) D →(2) T

then Bottleneck capacity = 2, since D → T has residual capacity = 2

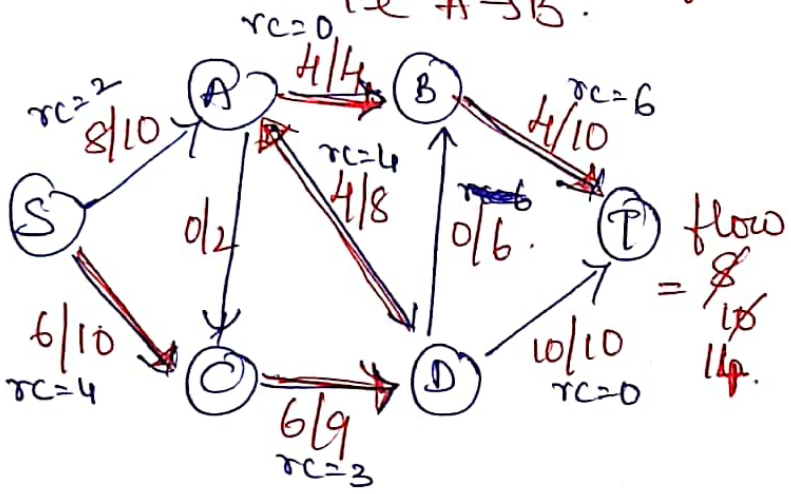


→ Now select another augmenting path

$S \xrightarrow{8} C \xrightarrow{7} D \xrightarrow{8} A \xrightarrow{4} B \xrightarrow{10} T$.

Here bottleneck capacity = 4.

i'e A → B.



D → A can be selected, because augmenting paths can be ~~selected~~ selected via either

① Using non-full forward edges (or)

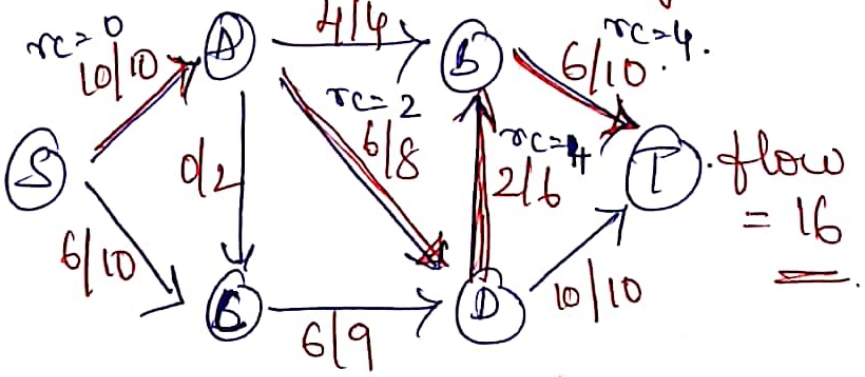
② Using non-empty backward edges.

Here D → A is non-empty backward edge.

→ Select another augmenting path.

$S \xrightarrow{2} A \xrightarrow{4} D \xrightarrow{6} B \xrightarrow{6} T$.

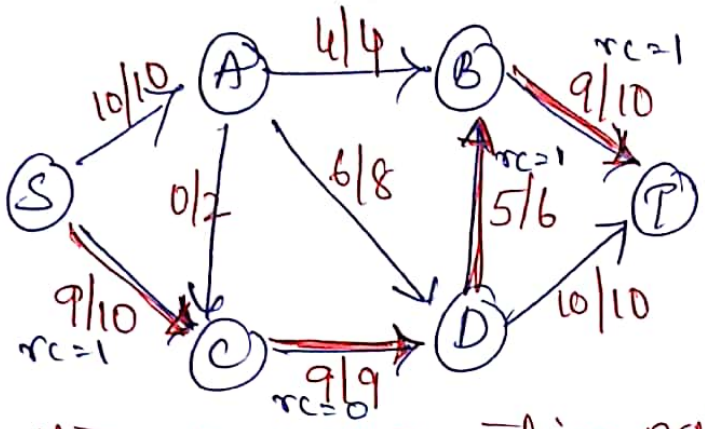
Bottleneck capacity = 2.



→ Another path is possible

$S \xrightarrow{3} C \xrightarrow{3} D \xrightarrow{4} B \xrightarrow{4} T$.

∴ Bottleneck capacity = 3.



Is there any augmenting path possible?

S→A is not possible, because it is full

S→C, residual capacity = 1, but

C→D is full. and.

C→A backward edge is not possible because it is not non-empty (non-zero) value, it is zero. (0/2).

Hence S→A, C→D, C→A } blocked, therefor, no other augmenting path is possible.

∴ Max Flow = 19 ✓

→ Use BFS to find augmenting paths.

Edmonds-Karp Alg is used to find augmenting paths, which basically uses BFS.

Ford-Fulkers in simple steps:—

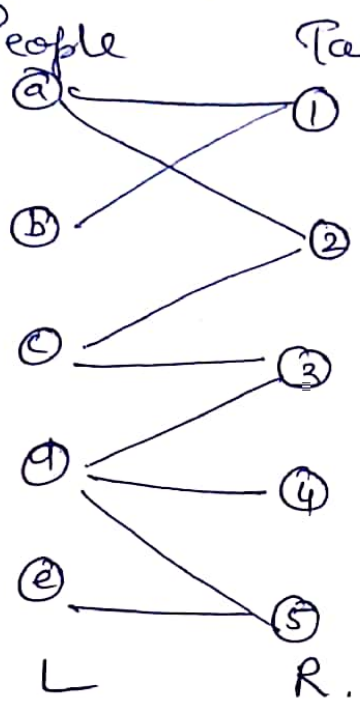
repeat

1. $G \rightarrow G_f$.
2. Find augmenting paths: $S \rightarrow T$ in G_f.
3. Augment the paths by bottleneck capacity
4. $f \rightarrow f1$.

Until no new augmenting paths found.

Bipartite Matching.

Suppose we have a set of people L and set of Tasks R.



→ This is a bipartite graph, in this if there is an edge means, it indicates that Person is capable of doing that task.

→ Each person can do only certain jobs not all [For Ex Person (d) can do Tasks 3, 4 & 5].

→ A matching gives an assignment of people to tasks

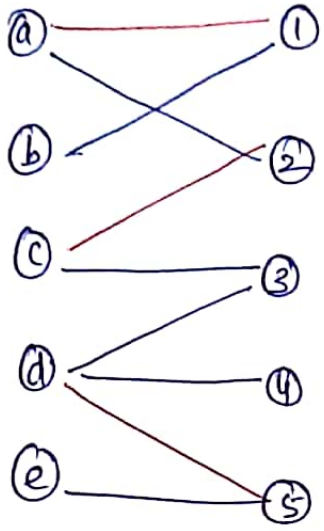
→ Problem is that, Find the matching i.e assignment from people to tasks such that we get as many tasks done as possible.

Constraints are:-

(1) One person can do only one task

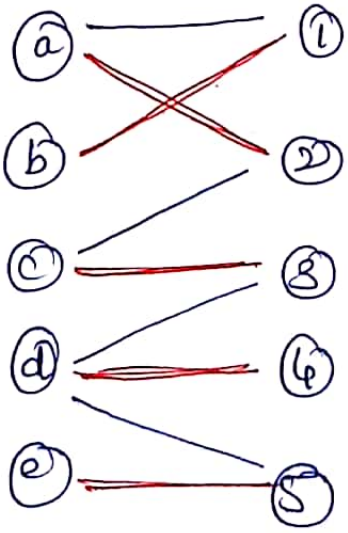
(2) One task need only one person.

Eg



* This is a bad assignment, since this is not maximum tasks getting done.

If I am smart! 😊



This is giving maximum matching

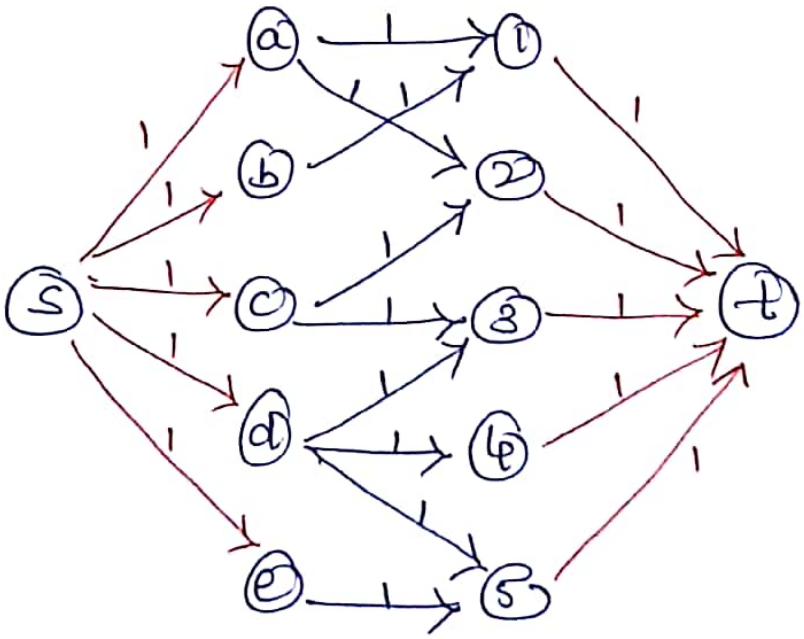
Solution :-

Transform Bipartite matching instance into Network flow problem.

Steps

1. Given a bipartite graph $G = (A \cup B, E)$, direct edges from A to B.
2. Add new vertices s and t.
3. Add an edge from s to every vertex in A.
4. Add an edge from every vertex in B to t.
5. Make all capacities 1.
6. Solve max flow problem on this new graph i.e maxflow will give max matching.

The edges used in the max network flow will correspond to the largest possible matching!



→ Because the capacities are all 1, we will either:

- use edge completely
or
- not use an edge at all.

Let M be the set of edges going from A to B that we use, then

We show that

① M is matching

② M is the largest-possible matching.