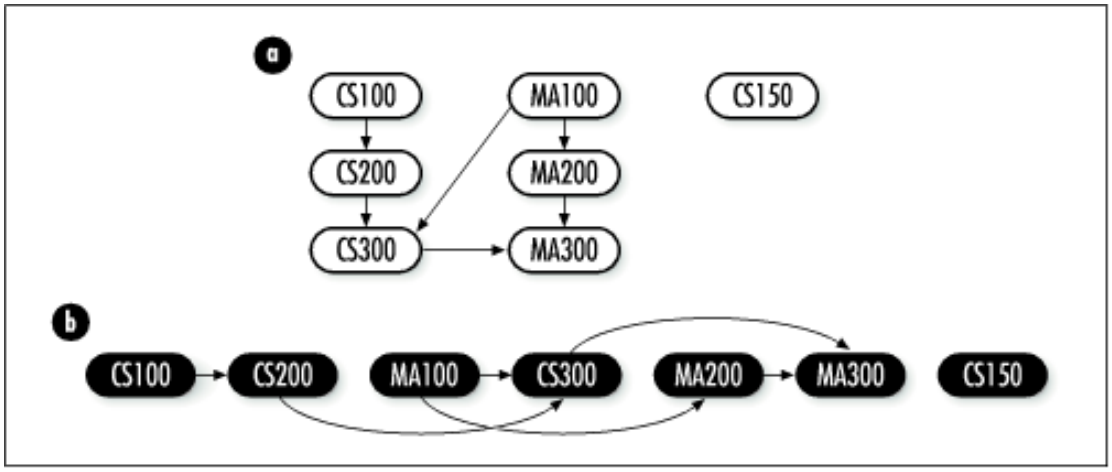
Graph Algorithms

Topological Sort:

A topological sort of a graph is a linear ordering of all its vertices such that if G contains an edge (u, v), then u appears before v in the ordering. Through topological sort, we can arrange the vertices in such a way that every edge goes from left to right.



To find a topological ordering, we need a graph that is:

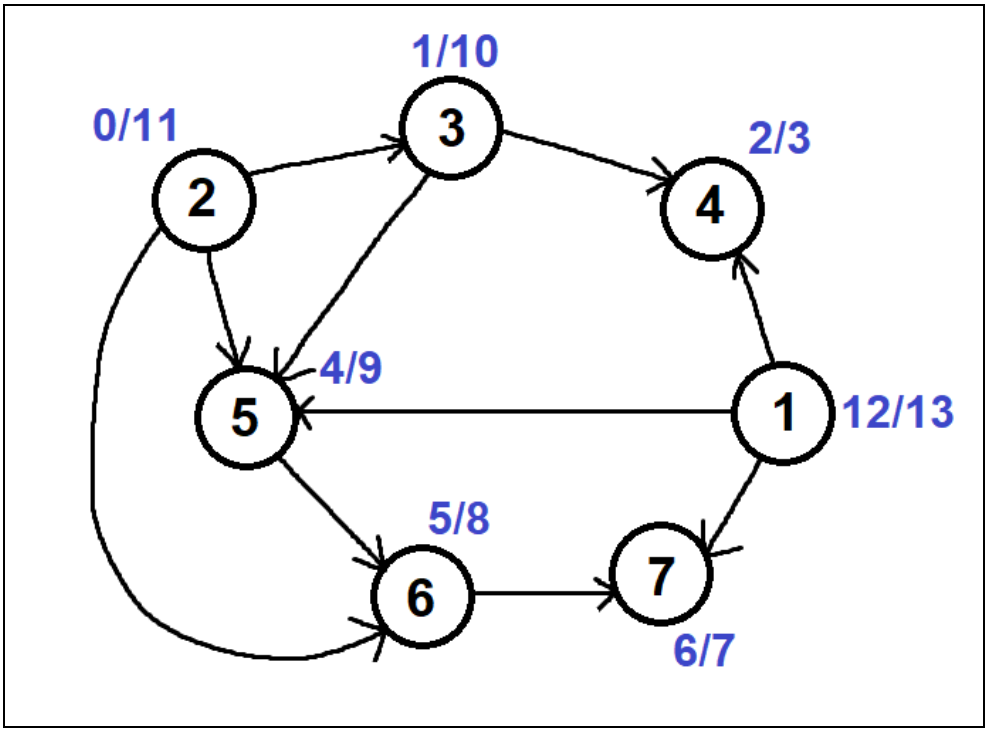
- **Directed**
- **Has no cycle**

Such a graph is called Directed Acyclic Graph (DAG)

Topological Ordering is not possible with undirected edges as an undirected edge between two vertices u and v means, there is an edge from u to v as well as from v to u. Because of this both the nodes u and v depend upon each other and none of them can appear before the other in the topological ordering without creating a contradiction.

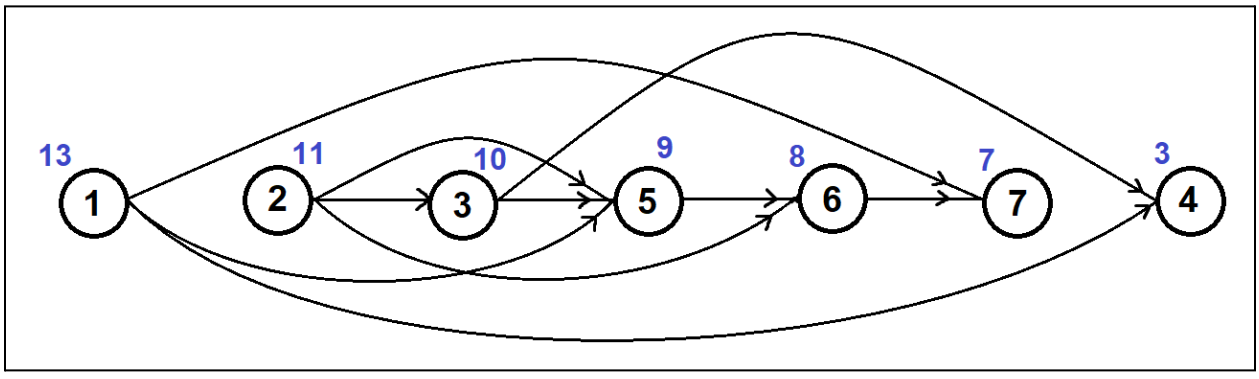
Topological Ordering is not possible with a cycle as it will always create a contradiction because all the vertices in a cycle are indirectly dependent on each other.

Topological Sort can be done through DFS by sorting the vertices according to the departure time in descending order (largest ending time to the smallest ending time).
Let the source vertex be '2' :



If the ending times are arranged in a decreasing manner we will get the ordering of the vertices as shown below.

End Time	13	11	10	9	8	7	3
Vertices	1	2	3	5	6	7	4



Time Complexity: O(V+E) as it is same as DFS

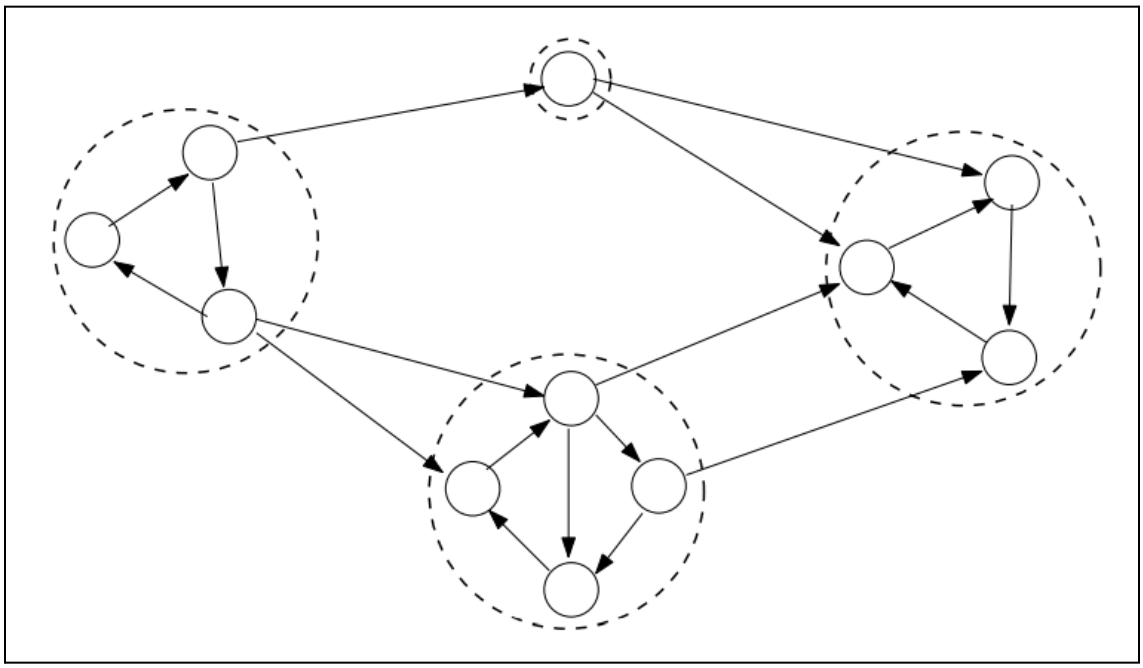
Strongly Connected Components:

In a directed graph G = (V, E), two nodes u and v are strongly connected if and only if there is a path from u to v and a path from v to u.

A Strongly Connected Component is a subgraph where there is a path from any node to any other node.

An undirected graph is always strongly connected. But, we need to check if a directed graph is strongly connected or not.

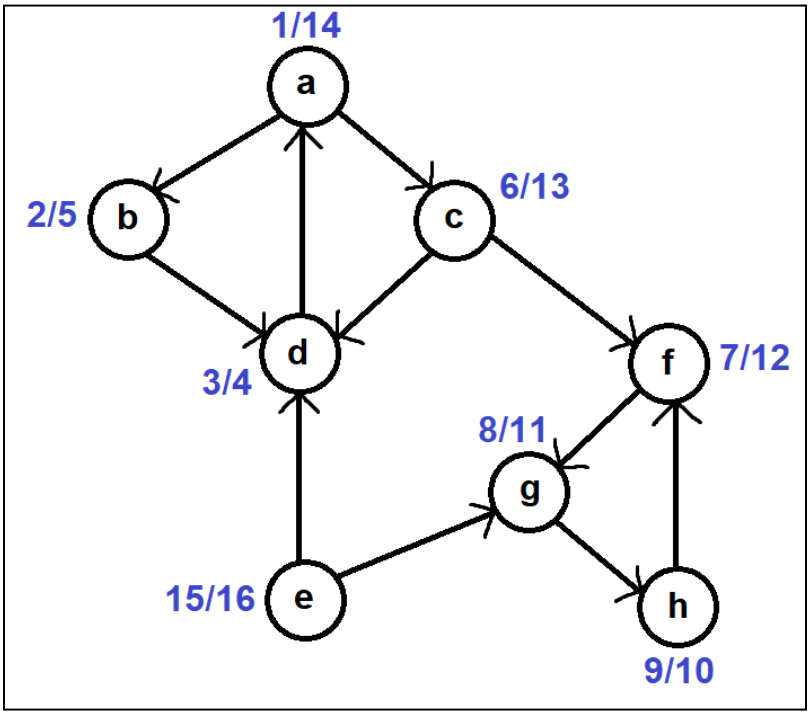
The figure below shows the strongly connected components of a directed graph:



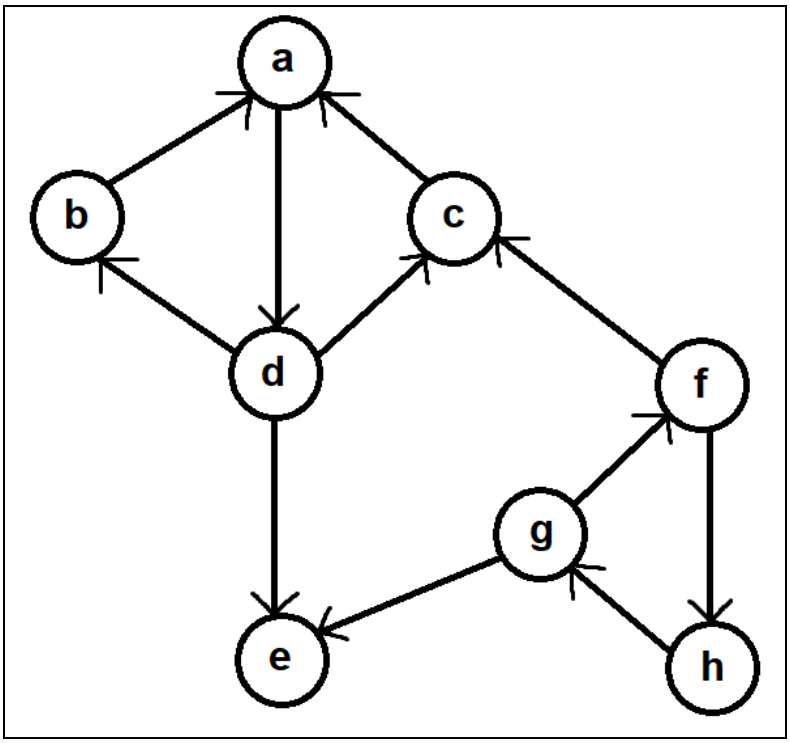
To get the strongly connected components, we can use Kosaraju's Algorithm.

- Run DFS on Graph G
- Draw G^T (reverse the edges)
- Run DFS on G^T with decreasing order of finishing time in G
- Combine vertices that are reachable in G^T
    - Start time of source vertex < Start time of destination vertex
    - Finish time of source vertex > Finish time of destination vertex

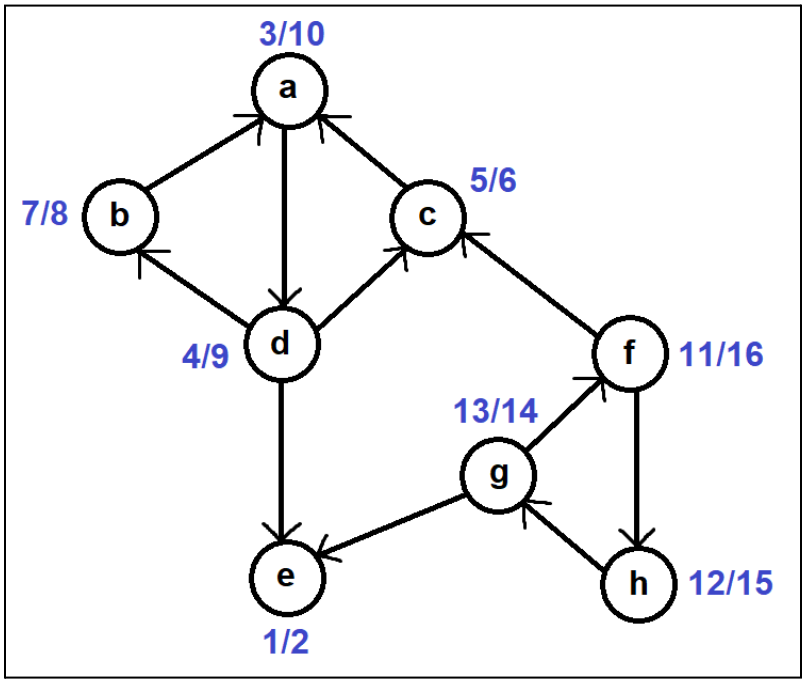
Step 1: Run DFS using any source



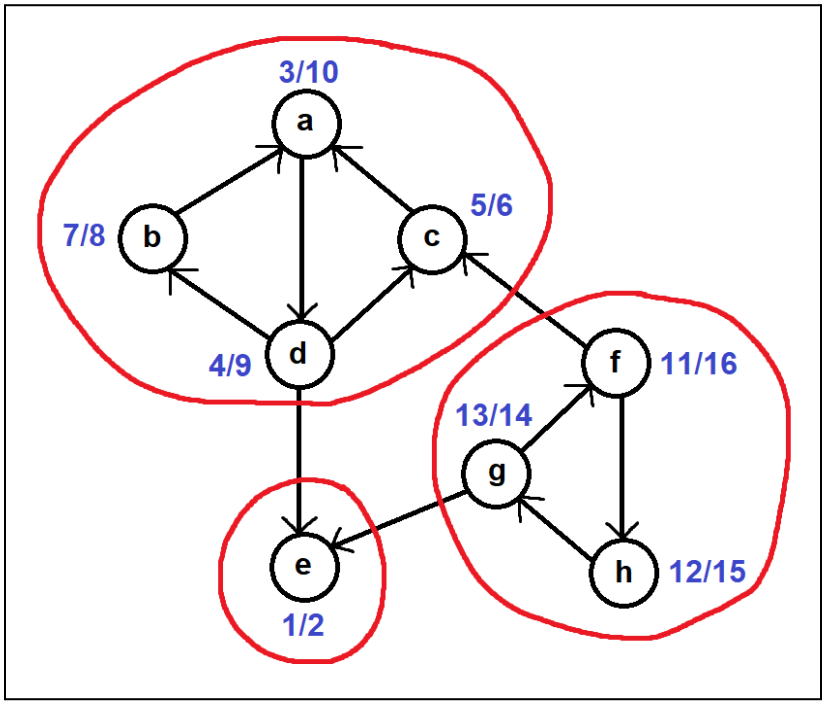
Step 2: Transpose the graph (Reverse Edges)



Step 3: Run DFS starting from decreasing finishing time from step 1



Step 4: Combine vertices starting from increasing starting time from step 3



Time Complexity: O(V+E) as it is same as DFS

Shortest Path:

Dijkstra's Algorithm: Single Source Shortest Path:

We know that by using BFS, we can find the shortest path from a source when the graph is unweighted. How do we approach this problem when the graph is weighted? One possible approach is to calculate the weights of all the possible paths from the source to all other nodes. But there are more optimal solutions.

We will see a greedy algorithm called Dijkstra's Algorithm. Greedy algorithm chooses the best local solution.

Dijkstra's algorithm assumes that all edge weights in the input graph are non-negative

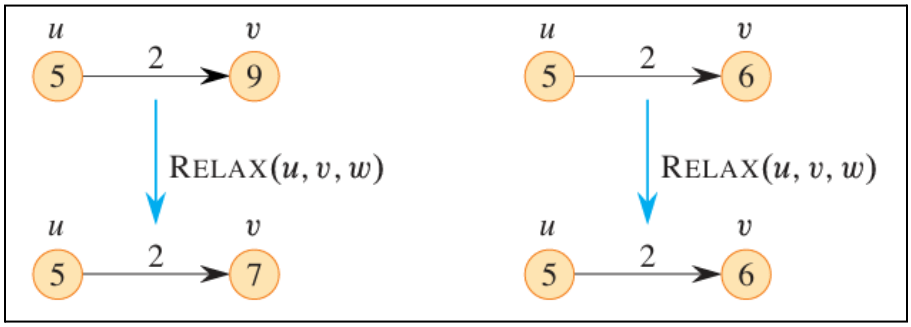
Shortest paths have no cycles, that is, they are simple paths. Since any acyclic path in a graph G = (V, E) contains at most |V| distinct vertices, it also contains at most |V| -1 edges. Assume, therefore, that any shortest path contains at most |V| -1 edges.

Relaxation:

```
Relax(u, v, w):
        do if d[v] > d[u] + w(u, v)
                then d[v] ← d[u] + w(u, v)
                π[v] ← u
```

The process of relaxing an edge (u, v) consists of testing whether going through vertex u improves the shortest path to vertex v found so far and, if so, updating d[v] and π[v].

In Dijkstra's algorithm, each edge is relaxed exactly once.

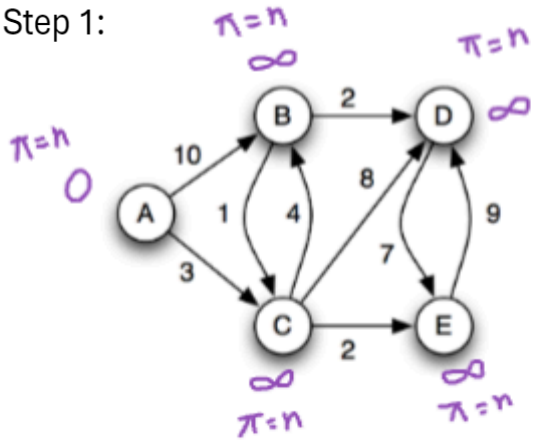


Pseudo Code:

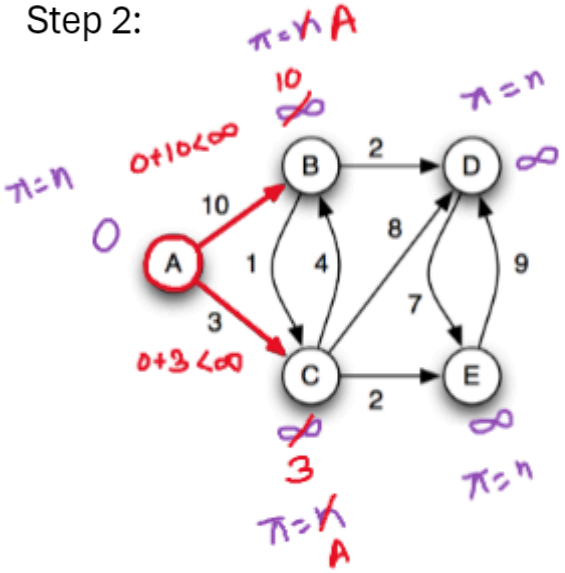
```
DIJKSTRA(G, s)
 1  for each v ∈ V[G]
 2        do d[v] ← ∞
 3           π[v] ← NIL
 4  d[s] ← 0
 5  Q ← V[G]
 6  while Q ≠ ∅
 7        do u ← EXTRACT-MIN(Q)
 8           for each vertex v ∈ Adj[u]
 9                do if d[v] > d[u] + w(u, v)
10                      then d[v] ← d[u] + w(u, v)
11                           π[v] ← u
```

Simulation:

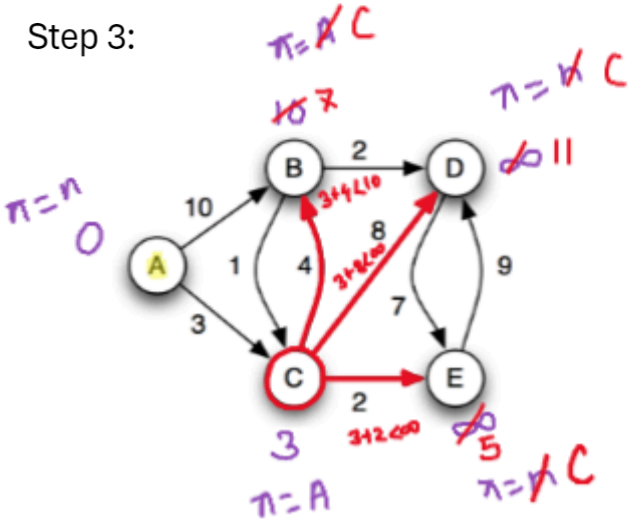
Step 1:



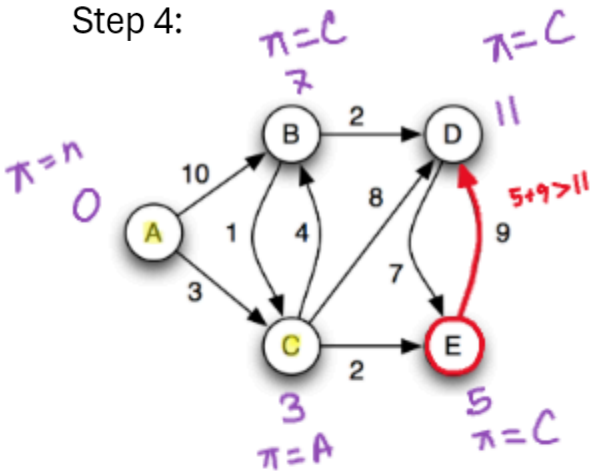
Step 2:



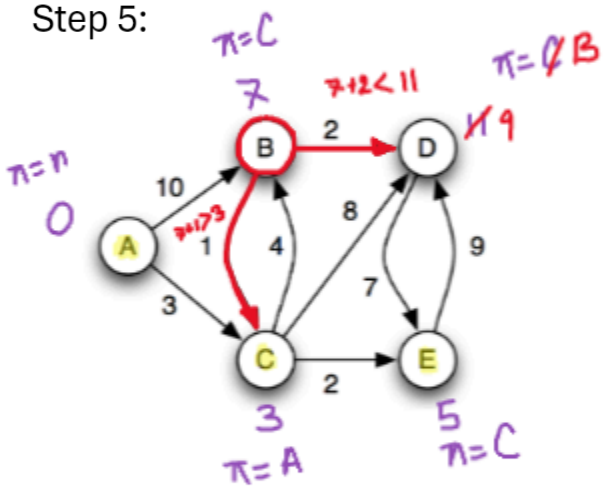
Step 3:



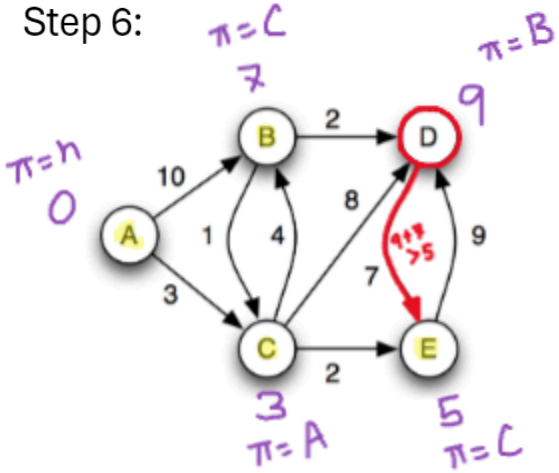
Step 4:



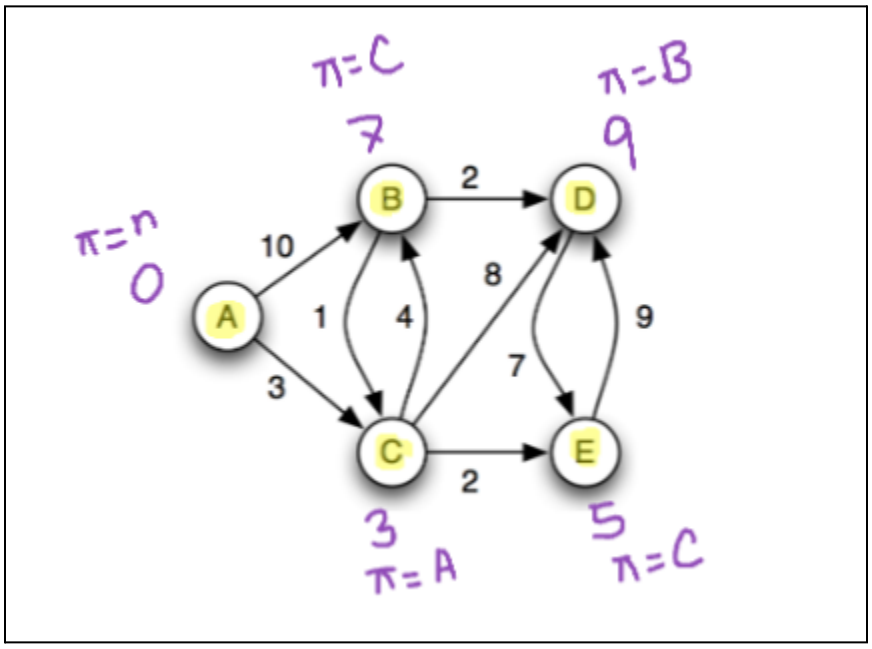
Step 5:



Step 6:



The final graph after applying Dijkstra's algorithm would be:



Time Complexity Analysis:

We will use a binary min-heap data structure as a priority queue to perform EXTRACT-MIN in Dijkstra's Algorithm. A min-heap is a tree-based data structure with the property that every node has a smaller key than its children.

T(findMin) = O(1)
T(removeMin) = O(log(V))
T(updateKey) = O(log(V))

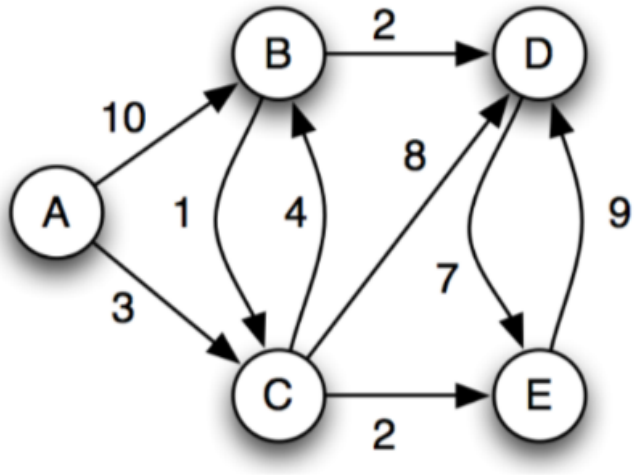
We need to findMin at most V vertices so O(1) * O(V) = O(V)
We need to removeMin at most V vertices so O(logV) * O(V) = O(VlogV)
We need to updateKey at most E edges so O(logV) * O(E) = O(ElogV)
Total = O(V) + O(VlogV) + O(ElogV)

Time Complexity = O((V + E) * log(V))
= O(ElogV) as E is V^2 at worst case

Drawbacks:

- It needs non-negative edge weights
- If the weights change, we need to re-run the whole Dijkstra's algorithm

Alternative Simulation:



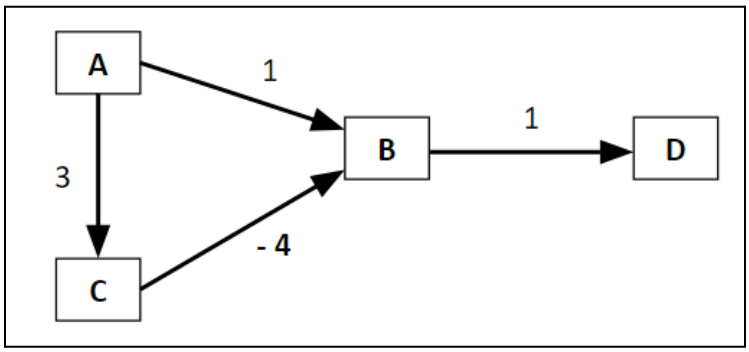
Given A as the source

	A	B	C	D	E
initial	0 (Null)	∞ (Null)	∞ (Null)	∞ (Null)	∞ (Null)
A	0 (Null)	10 (A)	3 (A)	∞ (Null)	∞ (Null)
C	0 (Null)	7 (C)	3 (A)	11 (C)	5 (C)
E	0 (Null)	7 (C)	3 (A)	11 (C)	5 (C)
B	0 (Null)	7 (C)	3 (A)	9 (B)	5 (C)
D	0 (Null)	7 (C)	3 (A)	9 (B)	5 (C)

(initial indicates the initial state considering A as the source. Dijkstra's Algorithm is applied and each time, the node with the minimum distance is dequeued and written in the first column)

Follow-up Question:

Q) Apply Dijkstra's Algorithm on the following graph and show that it does not work for negative edges.



Dijkstra's algorithm assumes that once it finds the shortest path to a vertex, it won't find a shorter path later. This assumption holds true when all edge weights are non-negative. If there are negative weights, Dijkstra's algorithm can fail because a vertex might seem to have a "shortest" path initially, but a path with a negative-weight edge found later could provide a shorter path. Dijkstra's algorithm does not revisit vertices once it considers them "done," so it can miss this better path.

Bellman-Ford Algorithm: Single Source Shortest Path for Negative Edges: [Optional]

The Bellman-Ford algorithm solves the single-source shortest-paths problem in the general case where edge weights may be negative. Given a weighted, directed graph G(V, E) with source vertex s and weight function w, the Bellman-Ford algorithm returns a boolean value indicating whether there is a negative-weight cycle that is reachable from the source. The algorithm indicates no solution exists if there is such a cycle. If there is no such cycle, the algorithm produces the shortest paths and their weights.

Bellman-Ford Algorithm is a dynamic programming algorithm. Dynamic programming algorithm chooses the best global solution by calculating all solutions and picking the best one.

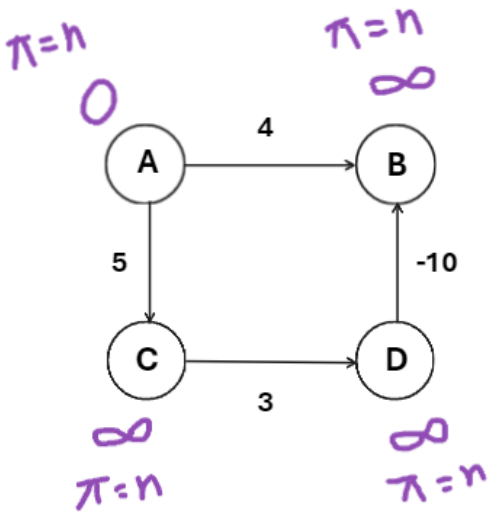
Bellman Ford Algorithm relaxes each edge of the graph |V|-1 times. The longest path from a vertex can be a maximum number of edges which is |V-1|.

Pseudo Code:

```
BELLMAN-FORD (G, s)
 1  for each v ∈ V[G]
 2        do d[v] ← ∞
 3           π[v] ← NIL
 4  d[s] ← 0
 5  for i = 1 to |V[G]| − 1
 6       for each edge (u, v) ∈ E[G]
 7            for each vertex v ∈ Adj[u]
 8                 do if d[v] > d[u] + w(u, v)
 9                       then d[v] ← d[u] + w(u, v)
10                            π[v] ← u
```

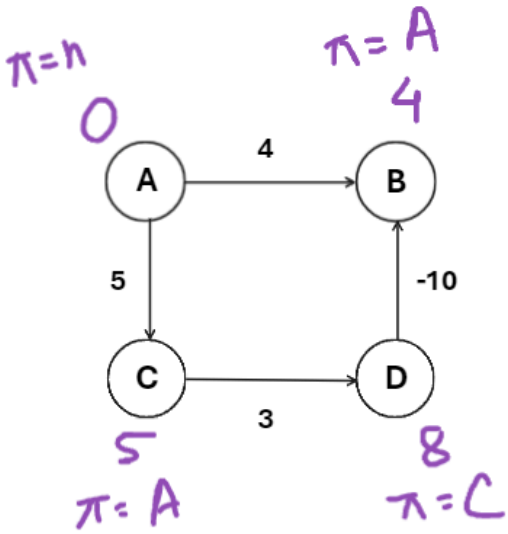
Simulation:

Initial:



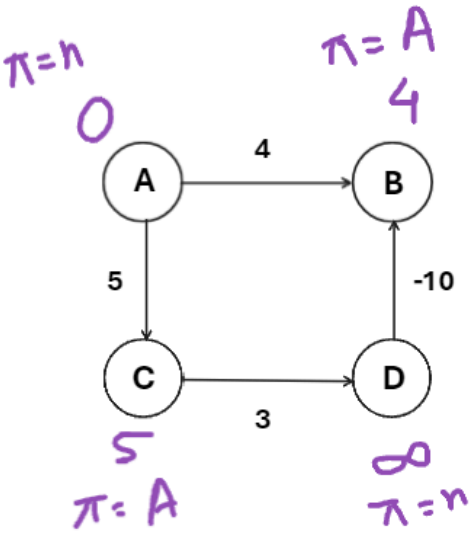
E = (D-B), (C-D), (A-C), (A-B)

i = 2



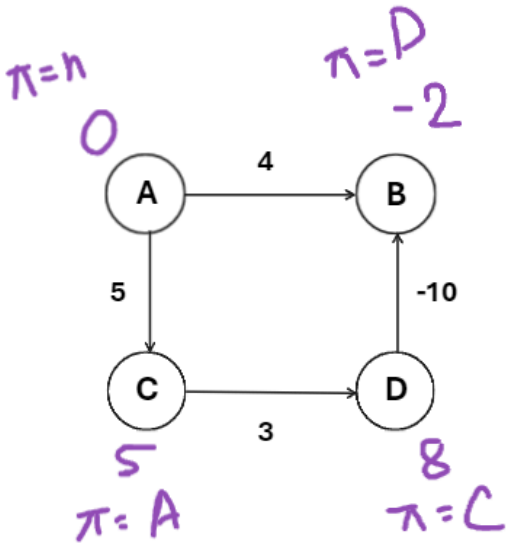
E = (D-B), (C-D), (A-C), (A-B)

i = 1



E = (D-B), (C-D), (A-C), (A-B)

i = 3



E = (D-B), (C-D), (A-C), (A-B)

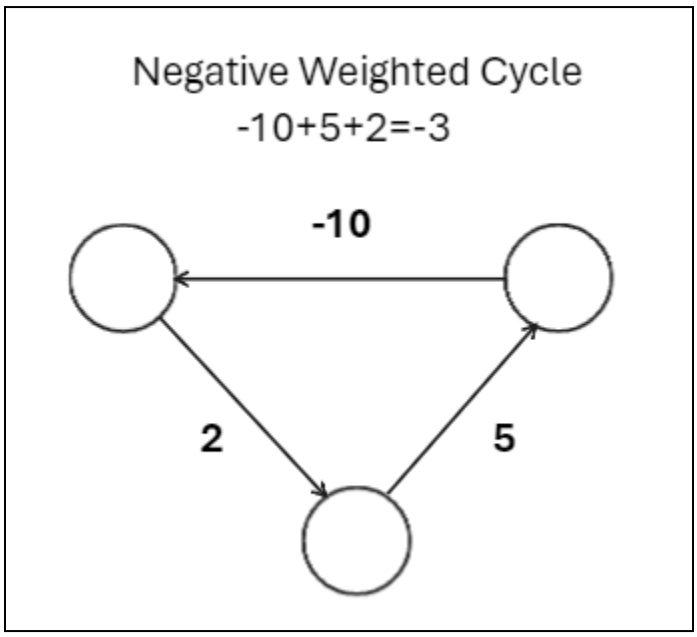
Time Complexity Analysis:

T(Relax) = O(1)
Relax E edges for V-1 times = O((V - 1) * E)

Time Complexity = O(V * E)

Drawbacks:

- It is slower than Dijkstra's algorithm
- It does not work when there is a negative weighted cycle



The idea of “Shortest Path” does not exist in a graph with a negative cycle; if we continue to go around the negative cycle an infinite number of times, then the path's cost will continue to decrease.

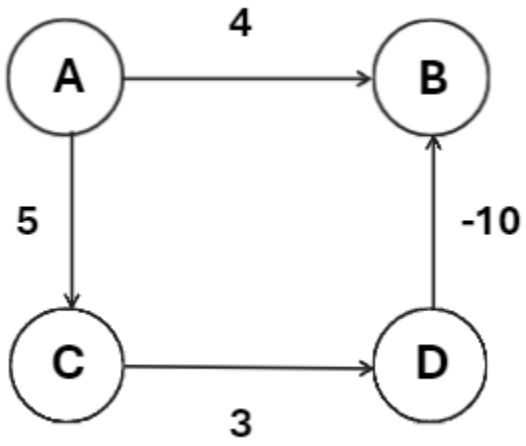
Follow-up Question:

Q) How can the Bellman-Ford algorithm detect a negative weighted cycle?

Ans:

After relaxing all edges V-1 time, Run one more time for relaxing all the edges, if there is any update, then a negative weighted cycle exists.

Alternative Simulation:



Given A as the source
V = 4
Iteration = V-1 = 3
Edge List = [DB, CD, AC, AB]

	A	B	C	D
i = 0	0 (Null)	∞ (Null)	∞ (Null)	∞ (Null)
i = 1	0 (Null)	4 (A)	5 (A)	∞ (Null)
i = 2	0 (Null)	4 (A)	5 (A)	8 (C)
i = 3	0 (Null)	-2 (D)	5 (A)	8 (C)
i = 4	0 (Null)	-2 (D)	5 (A)	8 (C)

(i = 0 indicates the initial state considering A as the source. Bellman-Ford is applied and the edges are relaxed V-1 times till i = 3. One more iteration i = 4 is done to check if there is any negative weighted cycle)