Single Source Shortest Paths

Introduction:

In a **shortest- paths problem**, we are given a weighted, directed graphs G = (V, E), with weight function **w: E → R** mapping edges to real-valued weights. The weight of path p = (v_0,v_1,..... v_k) is the total of the weights of its constituent edges:

$$w(P) = \sum_{i=1}^{k} w(v_{i-1}v_i)$$

We define the shortest - path weight from u to v by δ(u,v) = min (w (p): u→v), if there is a path from u to v, and δ(u,v)= ∞, otherwise.

The **shortest path** from vertex s to vertex t is then defined as any path p with weight w (p) = δ(s,t).

The **breadth-first- search algorithm** is the shortest path algorithm that works on unweighted graphs, that is, graphs in which each edge can be considered to have unit weight.

In a **Single Source Shortest Paths Problem**, we are given a Graph G = (V, E), we want to find the shortest path from a given source vertex s ∈ V to every vertex v ∈ V.

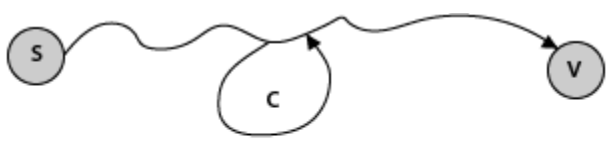
Variants:

There are some variants of the shortest path problem.

- **Single- destination shortest - paths problem:** Find the shortest path to a given destination vertex t from every vertex v. By shift the direction of each edge in the graph, we can shorten this problem to a single - source problem.
- **Single - pair shortest - path problem:** Find the shortest path from u to v for given vertices u and v. If we determine the single - source problem with source vertex u, we clarify this problem also. Furthermore, no algorithms for this problem are known that run asymptotically faster than the best single - source algorithms in the worst case.
- **All - pairs shortest - paths problem:** Find the shortest path from u to v for every pair of vertices u and v. Running a single - source algorithm once from each vertex can clarify this problem; but it can generally be solved faster, and its structure is of interest in the own right.

Shortest Path: Existence:

If some path from s to v contains a negative cost cycle then, there does not exist the shortest path. Otherwise, there exists a shortest s - v that is simple.



Cost of C < 0

Negative Weight Edges

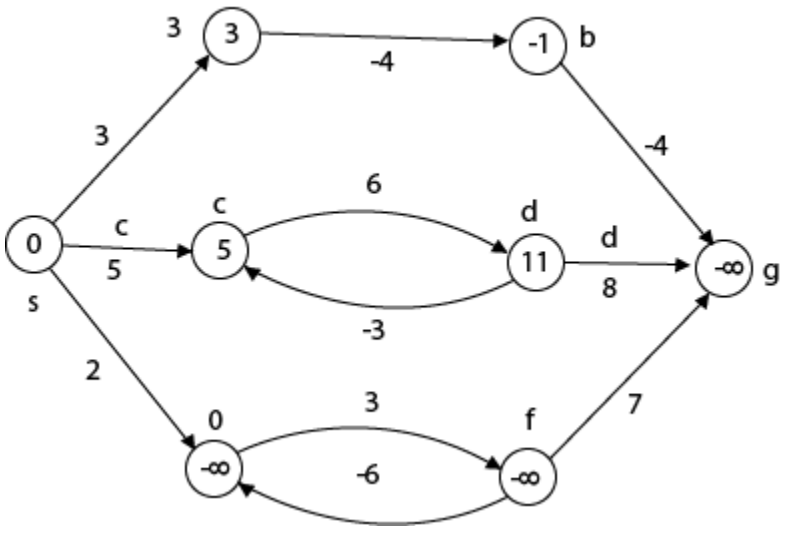
It is a weighted graph in which the total weight of an edge is negative. If a graph has a negative edge, then it produces a chain. After executing the chain if the output is negative then it will give - ∞ weight and condition get discarded. If weight is less than negative and - ∞ then we can't have the shortest path in it.

Briefly, if the output is -ve, then both condition get discarded.

1. - ∞
2. Not less than 0.

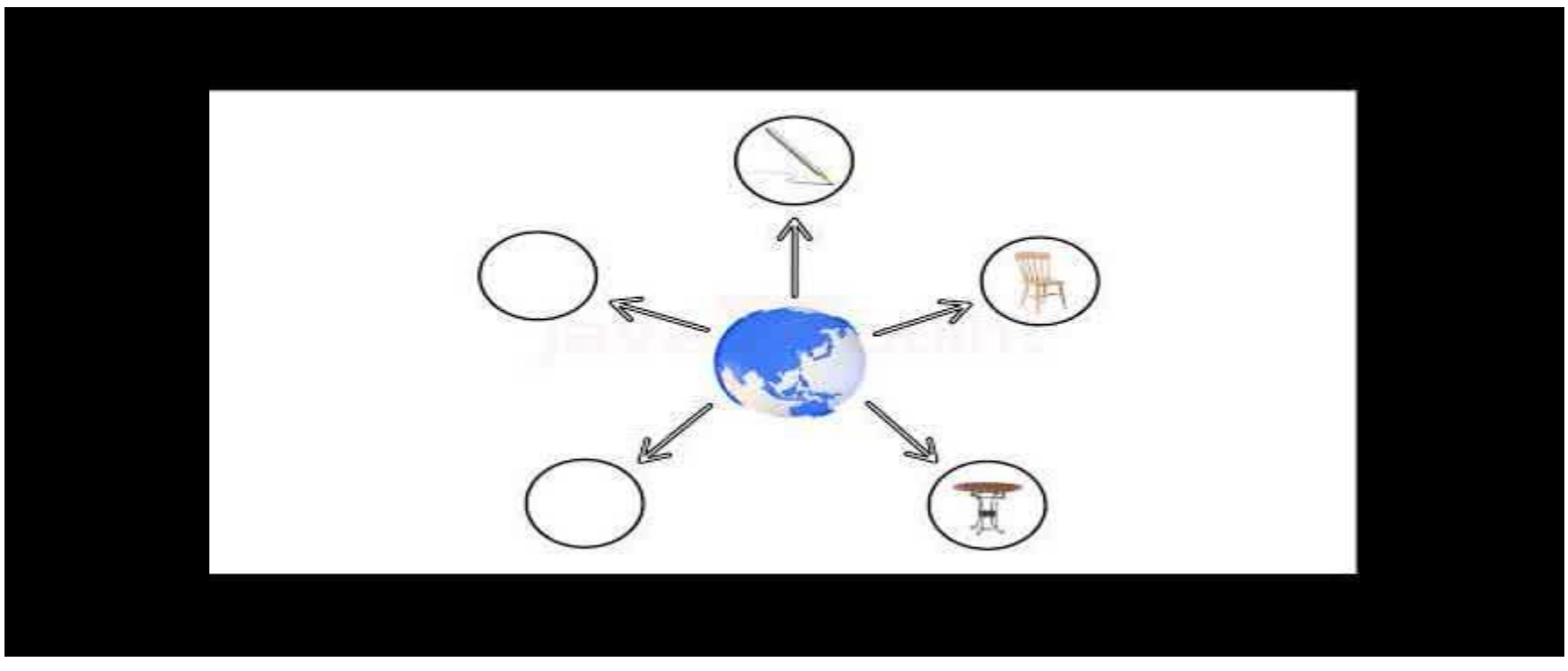
And we cannot have the shortest Path.

Example:



1. Beginning from s
2. Adj [s] = [a, c, e]
3. Weight from s to a is 3

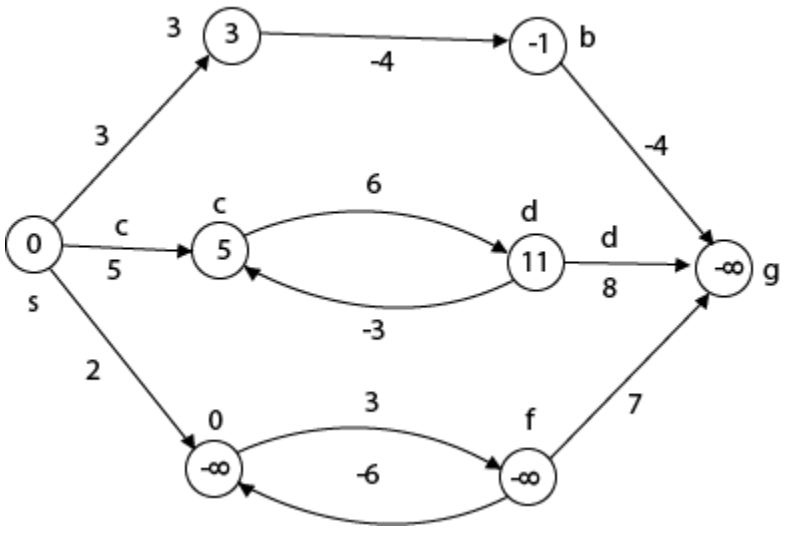
Suppose we want to calculate a path from s→c. So We have 2 paths /weight



1. s to c = 5, s→c→d→c=8
2. But s→c is minimum
3. So s→c = 5

Suppose we want to calculate a path from s→e. So we have two paths again

1. s→e = 2, s→e→f→e=-1
2. As -1 < 0 ∴ Condition gets discarded. If we execute **this** chain, we will get - ∞. So we can't get the shortest path ∴ e = ∞.



This figure illustrates the effects of negative weights and negative weight cycle on the shortest path weights.

Because there is only one path from "s to a" (the path <s, a>), δ (s, a) = w (s, a) = 3.

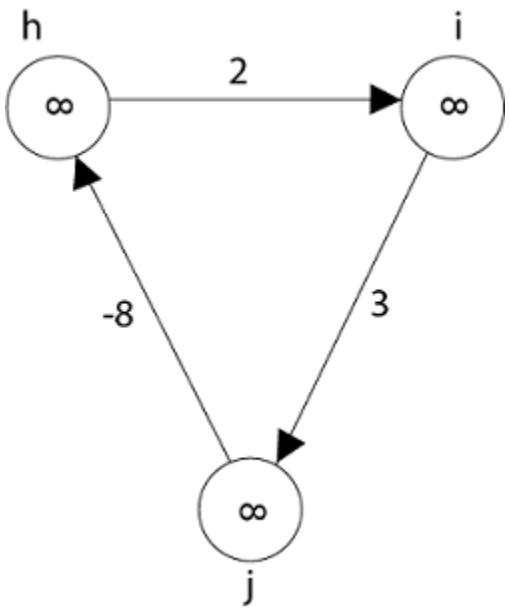
Furthermore, there is only one path from "s to b", so δ (s, b) = w (s, a) + w (a, b) = 3 + (-4) = - 1.

There are infinite many path from "s to c" : <s, c> : <s, c, d, c>, <s, c, d, c, d, c> and so on. Because the cycle <c, d, c> has weight δ (c, d) = w (c, d) + w (d, c) = 6 + (-3) = 3, which is greater than 0, the shortest path from s to c is <s, c> with weight δ (s, c) = 5.

Similarly, the shortest path from "s to d" is <s, c, d> with weight δ (s, d) = w (s, c) + w (s, d) = 11.

Analogously, there are infinite many paths from s to e: <s, e>, <s, e, f, e>, <s, e, f, e, f, e> and so on. Since the cycle <e, f, e> has weight δ (e, f) = w (e, f) + w (f, e) = 3 + (-6) = -3. So - 3 < 0, however there is no shortest path from s to e. B8y traversing the negative weight cycle <e, f, e>. This means path from s to e has arbitrary large negative weights and so δ (s, e) = - ∞.

Similarly δ (s, f) = - ∞ because g is reachable from f, we can also find a path with arbitrary large negative weight from s to g and δ (s, g) = - ∞



Vertices h, i, g also from negative weight cycle. They are also not reachable from the source node, so distance from the source is - ∞ to three of nodes (h, i, j).

Representing: Shortest Path

Given a graph G = (V, E), we maintain for each vertex v ∈ V a **predecessor** π [v] that is either another vertex or NIL. During the execution of shortest paths algorithms, however, the π values need not indicate shortest paths. As in breadth-first search, we shall be interested in the **predecessor subgraph** G_n= (V_n,E_n) induced by the value π. Here again, we define the vertex set V_π, to be the set of vertices of G with non - NIL predecessors, plus the source s:

$$V_\pi = \{v \in V: \pi [v] \neq NIL\} \cup \{s\} \}$$

The directed edge set E_Π is the set of edges induced by the Π values for vertices in V_Π:

$$E_\Pi = \{(\Pi[v], v) \in E: v \in V_\Pi - \{s\}\}$$

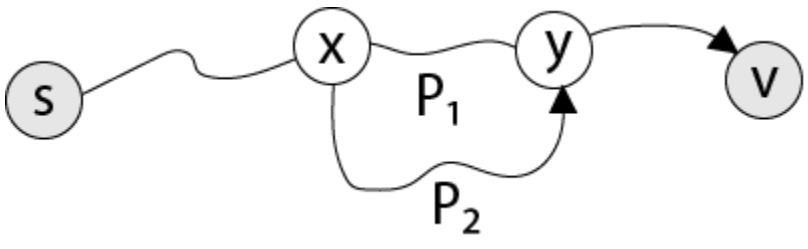
A **shortest - paths tree** rooted at s is a directed subgraph G = (V' E'), where V'∈ V andE'∈E, such that

1. V' is the set of vertices reachable from s in G
2. G' forms a rooted tree with root s, and
3. For all v ∈ V', the unique, simple path from s to v in G' is the shortest path from s to v in G.

Shortest paths are not naturally unique, and neither is shortest - paths trees.

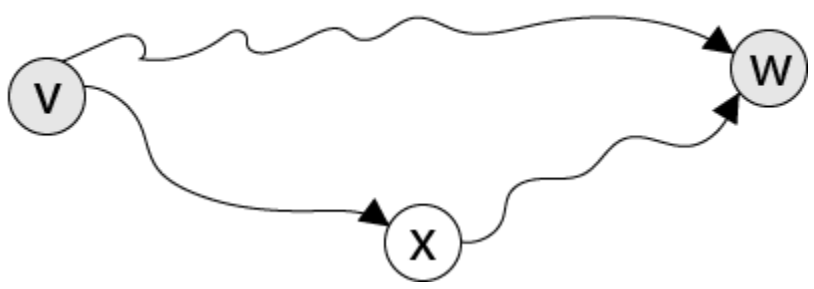
Properties of Shortest Path:

1. Optimal substructure property: All subpaths of shortest paths are shortest paths.



Let P_1 be x - y sub path of shortest s - v path. Let P_2 be any x - y path. Then cost of $P_1 \leq$ cost of P_2,otherwise P not shortest s - v path.

2. Triangle inequality: Let d (v, w) be the length of shortest path from v to w. Then, d (v, w) ≤ d (v, x) + d (x, w)



3. Upper-bound property: We always have d[v] ≥ δ(s, v) for all vertices v ∈ V, and once d[v] conclude the value δ(s, v), it never changes.

4. No-path property: If there is no path from s to v, then we regularly have d[v] = δ(s, v) = ∞.

5. Convergence property: If s->u->v is a shortest path in G for some u, v ∈ V, and if d[u] = δ(s, u) at any time prior to relaxing edge (u, v), then d[v] = δ(s, v) at all times thereafter.

Relaxation

The single - source shortest paths are based on a technique known as **relaxation**, a method that repeatedly decreases an upper bound on the actual shortest path weight of each vertex until the upper bound equivalent the shortest - path weight. For each vertex v ∈ V, we maintain an attribute d [v], which is an upper bound on the weight of the shortest path from source s to v. We call d [v] the **shortest path estimate**.

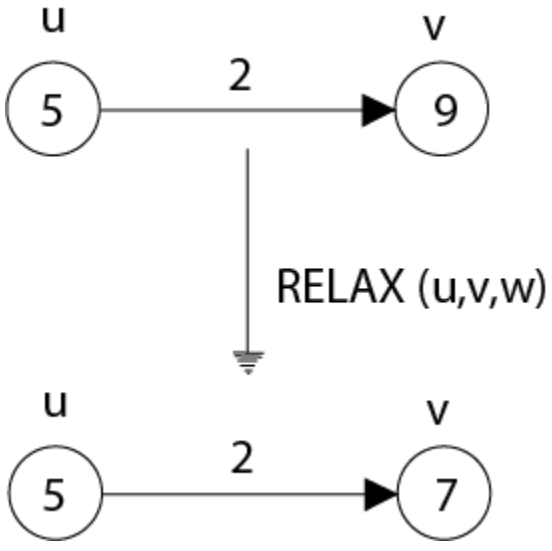
INITIALIZE - SINGLE - SOURCE (G, s)

```
1. for each vertex v ∈ V [G]
2. do d [v] ← ∞
3. π [v] ← NIL
4. d [s] ← 0
```

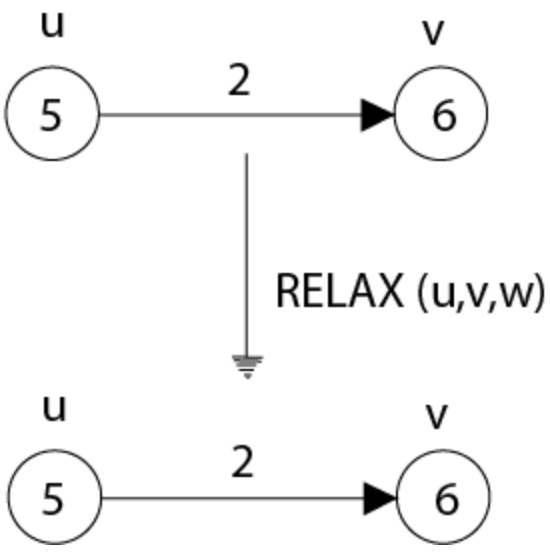
After initialization, π [v] = NIL for all v ∈ V, d [v] = 0 for v = s, and d [v] = ∞ for v ∈ V - {s}.

The development of relaxing an edge (u, v) consists of testing whether we can improve the shortest path to v found so far by going through u and if so, updating d [v] and π [v]. A relaxation step may decrease the value of the shortest - path estimate d [v] and updated v's predecessor field π [v].

Fig: Relaxing an edge (u, v) with weight w (u, v) = 2. The shortest-path estimate of each vertex appears within the vertex.



(a) Because v. d > u. d + w (u, v) prior to relaxation, the value of v. d decreases



(b) Here, v. d < u. d + w (u, v) before relaxing the edge, and so the relaxation step leaves v. d unchanged.

The subsequent code performs a relaxation step on edge (u, v)

RELAX (u, v, w)

```
1. If d [v] > d [u] + w (u, v)
2. then d [v] ← d [u] + w (u, v)
3. π [v] ← u
```

Dijkstra's Algorithm

It is a greedy algorithm that solves the single-source shortest path problem for a directed graph G = (V, E) with nonnegative edge weights, i.e., w (u, v) ≥ 0 for each edge (u, v) ∈ E.

Dijkstra's Algorithm maintains a set S of vertices whose final shortest - path weights from the source s have already been determined. That's for all vertices v ∈ S; we have d [v] = δ (s, v). The algorithm repeatedly selects the vertex u ∈ V - S with the minimum shortest - path estimate, insert u into S and relaxes all edges leaving u.

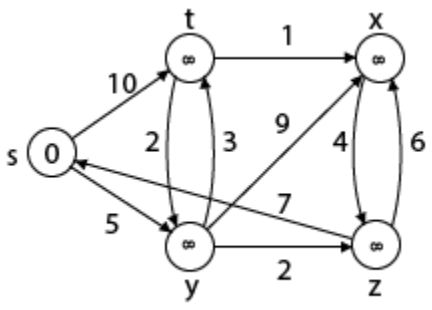
Because it always chooses the "lightest" or "closest" vertex in V - S to insert into set S, it is called as the **greedy strategy**.

Dijkstra's Algorithm (G, w, s)

```
1. INITIALIZE - SINGLE - SOURCE (G, s)
2. S←∅
3. Q←V [G]
4. while Q ≠ ∅
5. do u ← EXTRACT - MIN (Q)
6. S ← S ∪ {u}
7. for each vertex v ∈ Adj [u]
8. do RELAX (u, v, w)
```

Analysis: The running time of Dijkstra's algorithm on a graph with edges E and vertices V can be expressed as a function of |E| and |V| using the Big - O notation. The simplest implementation of the Dijkstra's algorithm stores vertices of set Q in an ordinary linked list or array, and operation Extract - Min (Q) is simply a linear search through all vertices in Q. In this case, the running time is O ($|V^2|+|E|=O(V^2)$).

Example:



Solution:

Step1: Q =[s, t, x, y, z]

We scanned vertices one by one and find out its adjacent. Calculate the distance of each adjacent to the source vertices.

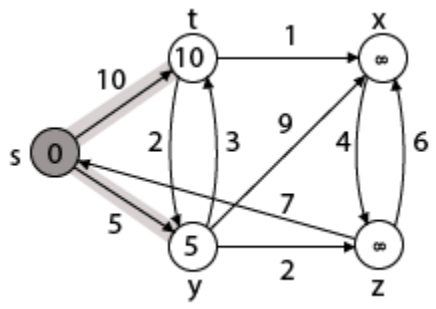
We make a stack, which contains those vertices which are selected after computation of shortest distance.

Firstly we take's' in stack M (which is a source)

1. M = [S] Q = [t, x, y, z]

Step 2: Now find the adjacent of s that are t and y.

1. Adj [s] → t, y [Here s is u and t and y are v]



Case - (i) s → t

d [v] > d [u] + w [u, v]

d [t] > d [s] + w [s, t]

∞ > 0 + 10 [false condition]

Then **d [t] ← 10**

π [t] ← 5

Adj [s] ← t, y

Case - (ii) s→ y

d [v] > d [u] + w [u, v]

d [y] > d [s] + w [s, y]

∞ > 0 + 5 [false condition]

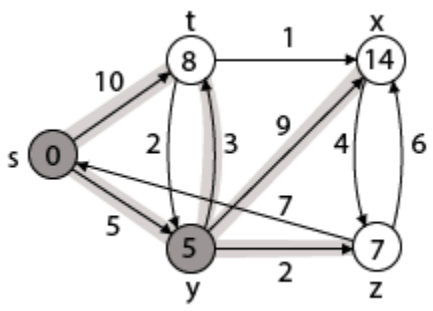
∞ > 5

Then **d [y] ← 5**

π [y] ← 5

By comparing case (i) and case (ii)
Adj [s] → t = 10, y = 5
y is shortest

y is assigned in 5 = [s, y]



Step 3: Now find the adjacent of y that is t, x, z.

1. Adj [y] → t, x, z [Here y is u and t, x, z are v]

Case - (i) y →t

d [v] > d [u] + w [u, v]

d [t] > d [y] + w [y, t]

10 > 5 + 3

10 > 8

Then d [t] ← 8

π [t] ← y

Case - (ii) y → x

d [v] > d [u] + w [u, v]

d [x] > d [y] + w [y, x]

∞ > 5 + 9

∞ > 14

Then d [x] ← 14

π [x] ← 14

Case - (iii) y → z

d [v] > d [u] + w [u, v]

d [z] > d [y] + w [y, z]

∞ > 5 + 2

∞ > 7

Then d [z] ← 7

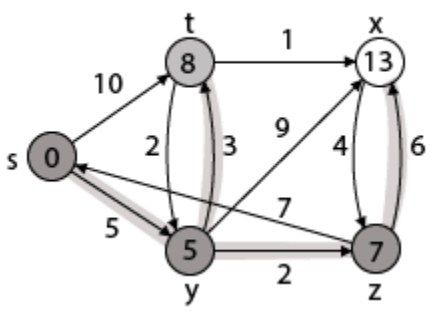
π [z] ← y

By comparing case (i), case (ii) and case (iii)

Adj [y] → x = 14, t = 8, z =7

z is shortest

z is assigned in 7 = [s, z]



Step - 4 Now we will find adj [z] that are s, x

1. Adj [z] → [x, s] [Here z is u and s and x are v]

Case - (i) z → x

d [v] > d [u] + w [u, v]

d [x] > d [z] + w [z, x]

14 > 7 + 6

14 > 13

Then d [x] ← 13

π [x] ← z

Case - (ii) z → s

d [v] > d [u] + w [u, v]

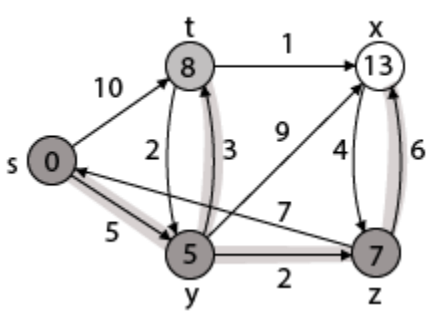
d [s] > d [z] + w [z, s]

0 > 7 + 7

0 > 14

∴ This condition does not satisfy so it will be discarded.

Now we have x = 13.



Step 5: Now we will find Adj [t]

Adj [t] → [x, y] [Here t is u and x and y are v]

Case - (i) t → x

d [v] > d [u] + w [u, v]

d [x] > d [t] + w [t, x]

13 > 8 + 1

13 > 9

Then d [x] ← 9

π [x] ← t

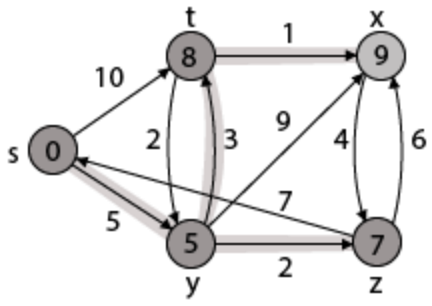
Case - (ii) t → y

d [v] > d [u] + w [u, v]

d [y] > d [t] + w [t, y]

5 > 10

∴ This condition does not satisfy so it will be discarded.

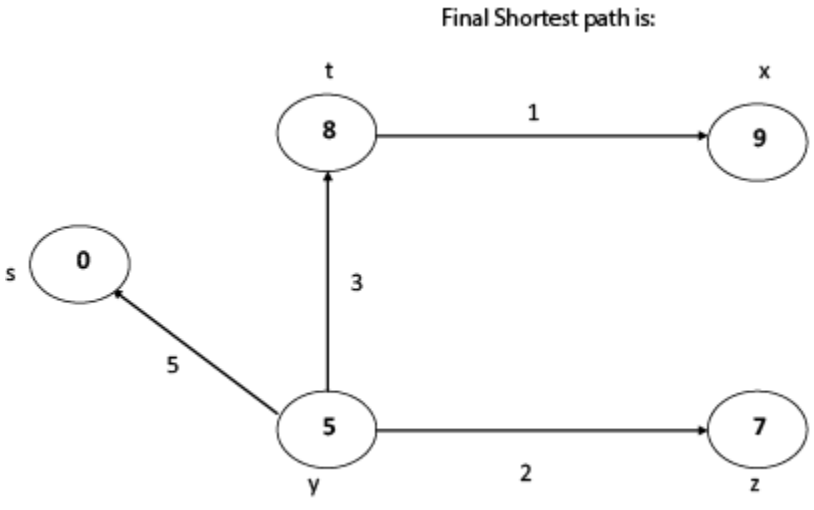


Thus we get all shortest path vertex as

Weight	from	s	to	y	is	5
Weight	from	s	to	z	is	7
Weight	from	s	to	t	is	8

Weight from s to x is 9

These are the shortest distance from the source's' in the given graph.



Disadvantage of Dijkstra's Algorithm:

1. It does a blind search, so wastes a lot of time while processing.
2. It can't handle negative edges.
3. It leads to the acyclic graph and most often cannot obtain the right shortest path.
4. We need to keep track of vertices that have been visited.

Bellman-Ford Algorithm

Solves single shortest path problem in which edge weight may be negative but no negative cycle exists.

This algorithm works correctly when some of the edges of the directed graph G may have negative weight. When there are no cycles of negative weight, then we can find out the shortest path between source and destination.

It is slower than Dijkstra's Algorithm but more versatile, as it capable of handling some of the negative weight edges.

This algorithm detects the negative cycle in a graph and reports their existence.

Based on the "**Principle of Relaxation**" in which more accurate values gradually recovered an approximation to the proper distance by until eventually reaching the optimum solution.

Given a weighted directed graph G = (V, E) with source s and weight function w: E → R, the Bellman-Ford algorithm returns a Boolean value indicating whether or not there is a negative weight cycle that is attainable from the source. If there is such a cycle, the algorithm produces the shortest paths and their weights. The algorithm returns TRUE if and only if a graph contains no negative - weight cycles that are reachable from the source.

Recurrence Relation

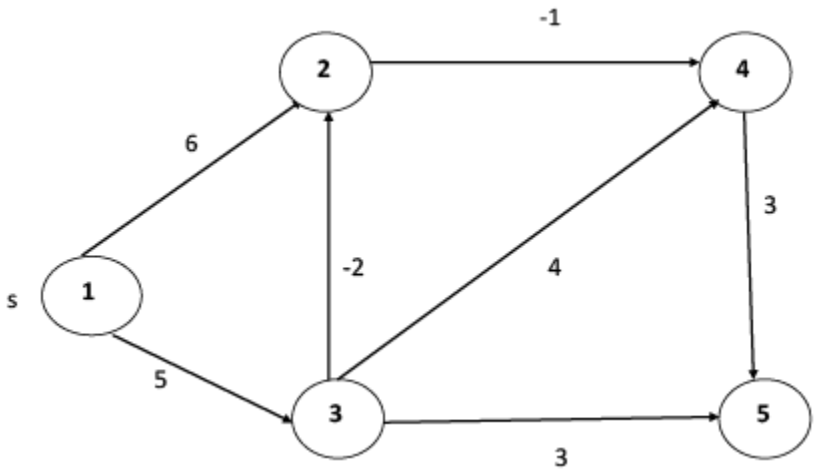
$dist^{k}$ [u] = [min[$dist^{k-1}$ [u],min[$dist^{k-1}$ [i]+cost [i,u]]] as i except u.

k → k is the source vertex
u → u is the destination vertex
i → no of edges to be scanned concerning a vertex.

BELLMAN -FORD (G, w, s)

```
1. INITIALIZE - SINGLE - SOURCE (G, s)
2. for i ← 1 to |V[G]| - 1
3. do for each edge (u, v) ∈ E [G]
4. do RELAX (u, v, w)
5. for each edge (u, v) ∈ E [G]
6. do if d [v] > d [u] + w (u, v)
7. then return FALSE.
8. return TRUE.
```

Example: Here first we list all the edges and their weights.



Solution:

$dist^k$ [u] = [min[$dist^{k-1}$ [u],min[$dist^{k-1}$ [i]+cost [i,u]]] as i ≠ u.

Vertices

No of Edges Traversed

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

$dist^2$ [2]=min[$dist^1$ [2],min[$dist^1$ [1]+cost[1,2],$dist^1$ [3]+cost[3,2],$dist^1$ [4]+cost[4,2],$dist^1$ [5]+cost[5,2]]]

Min = [6, 0 + 6, 5 + (-2), ∞ + ∞ , ∞ +∞] = 3

$dist^2$ [3]=min[$dist^1$ [3],min[$dist^1$ [1]+cost[1,3],$dist^1$ [2]+cost[2,3],$dist^1$ [4]+cost[4,3],$dist^1$ [5]+cost[5,3]]]

Min = [5, 0 +∞, 6 +∞, ∞ + ∞ , ∞ + ∞] = 5

$dist^2$ [4]=min[$dist^1$ [4],min[$dist^1$ [1]+cost[1,4],$dist^1$ [2]+cost[2,4],$dist^1$ [3]+cost[3,4],$dist^1$ [5]+cost[5,4]]]

Min = [∞, 0 +∞, 6 + (-1), 5 + 4, ∞ +∞] = 5

$dist^2$ [5]=min[$dist^1$ [5],min[$dist^1$ [1]+cost[1,5],$dist^1$ [2]+cost[2,5],$dist^1$ [3]+cost[3,5],$dist^1$ [4]+cost[4,5]]]

Min = [∞, 0 + ∞,6 + ∞,5 + 3, ∞ + 3] = 8

$dist^3$ [2]=min[$dist^2$ [2],min[$dist^2$ [1]+cost[1,2],$dist^2$ [3]+cost[3,2],$dist^2$ [4]+cost[4,2],$dist^2$ [5]+cost[5,2]]]

Min = [3, 0 + 6, 5 + (-2), 5 + ∞ , 8 + ∞] = 3

$dist^3$ [3]=min[$dist^2$ [3],min[$dist^2$ [1]+cost[1,3],$dist^2$ [2]+cost[2,3],$dist^2$ [4]+cost[4,3],$dist^2$ [5]+cost[5,3]]]

Min = [5, 0 + ∞, 3 + ∞, 5 + ∞,8 + ∞] = 5

$dist^3$ [4]=min[$dist^2$ [4],min[$dist^2$ [1]+cost[1,4],$dist^2$ [2]+cost[2,4],$dist^2$ [3]+cost[3,4],$dist^2$ [5]+cost[5,4]]]

Min = [5, 0 + ∞, 3 + (-1), 5 + 4, 8 + ∞] = 2

$dist^3$ [5]=min[$dist^2$ [5],min[$dist^2$ [1]+cost[1,5],$dist^2$ [2]+cost[2,5],$dist^2$ [3]+cost[3,5],$dist^2$ [4]+cost[4,5]]]

Min = [8, 0 + ∞, 3 + ∞, 5 + 3, 5 + 3] = 8

$dist^4$ [2]=min[$dist^3$ [2],min[$dist^3$ [1]+cost[1,2],$dist^3$ [3]+cost[3,2],$dist^3$ [4]+cost[4,2],$dist^3$ [5]+cost[5,2]]]

Min = [3, 0 + 6, 5 + (-2), 2 + ∞, 8 + ∞] =3

$dist^4$ [3]=min[$dist^3$ [3],min[$dist^3$ [1]+cost[1,3],$dist^3$ [2]+cost[2,3],$dist^3$ [4]+cost[4,3],$dist^3$ [5]+cost[5,3]]]

Min = 5, 0 + ∞, 3 + ∞, 2 + ∞, 8 + ∞] =5

$dist^4$ [4]=min[$dist^3$ [4],min[$dist^3$ [1]+cost[1,4],$dist^3$ [2]+cost[2,4],$dist^3$ [3]+cost[3,4],$dist^3$ [5]+cost[5,4]]]

Min = [2, 0 + ∞, 3 + (-1), 5 + 4, 8 + ∞] = 2

$dist^4$ [5]=min[$dist^3$ [5],min[$dist^3$ [1]+cost[1,5],$dist^3$ [2]+cost[2,5],$dist^3$ [3]+cost[3,5],$dist^3$ [5]+cost[4,5]]]

Min = [8, 0 +∞, 3 + ∞, 8, 5] = 5

Single Source Shortest Path in a directed Acyclic Graphs

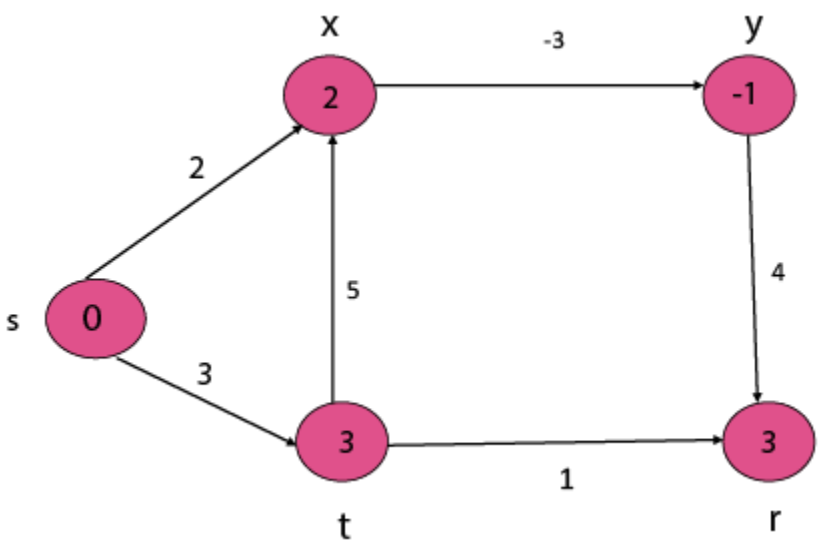
By relaxing the edges of a weighted DAG (Directed Acyclic Graph) G = (V, E) according to a topological sort of its vertices, we can figure out shortest paths from a single source in Ø(V+E) time. Shortest paths are always well described in a dag, since even if there are negative-weight edges, no negative-weight cycles can exist.

DAG - SHORTEST - PATHS (G, w, s)

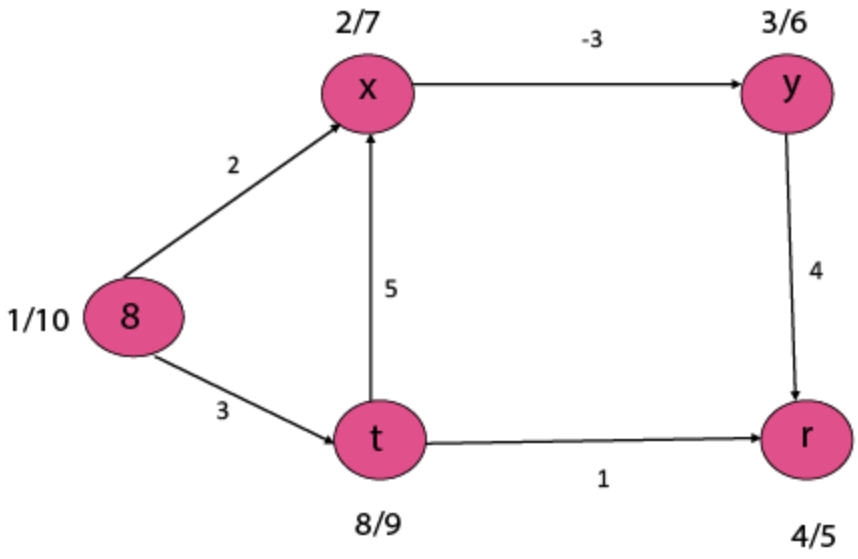
```
1. Topologically sort the vertices of G.
2. INITIALIZE - SINGLE- SOURCE (G, s)
3. for each vertex u taken in topologically sorted order
4. do for each vertex v ∈ Adj [u]
5. do RELAX (u, v, w)
```

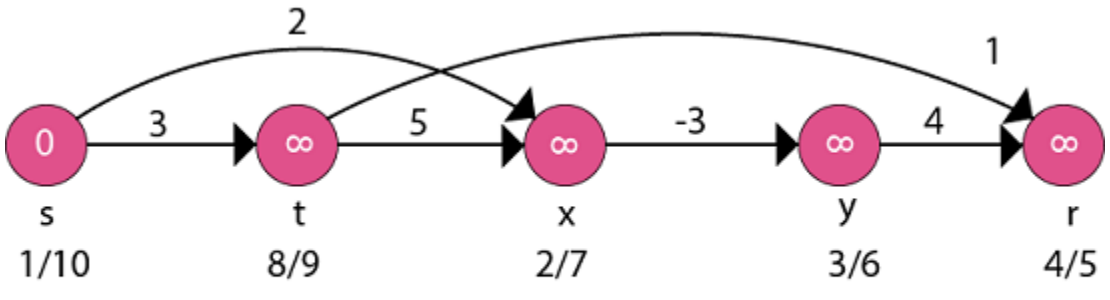
The running time of this data is determined by line 1 and by the for loop of lines 3 - 5. The topological sort can be implemented in Ø (V + E) time. In the for loop of lines 3 - 5, as in Dijkstra's algorithm, there is one repetition per vertex. For each vertex, the edges that leave the vertex are each examined exactly once. Unlike Dijkstra's algorithm, we use only O (1) time per edge. The running time is thus Ø (V + E), which is linear in the size of an adjacency list depiction of the graph.

Example:



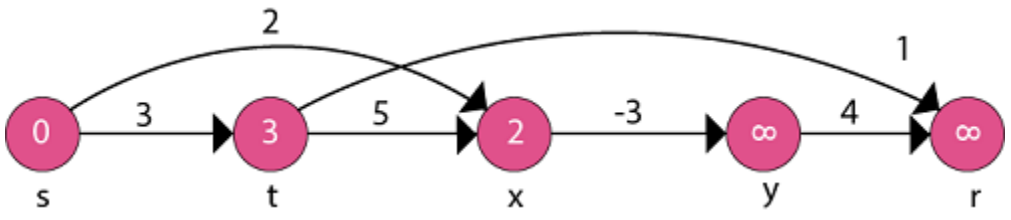
Step1: To topologically sort vertices apply **DFS (Depth First Search)** and then arrange vertices in linear order by decreasing order of finish time.



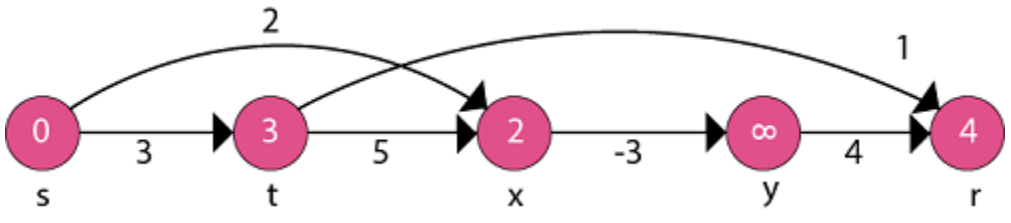


Initialize Single Source

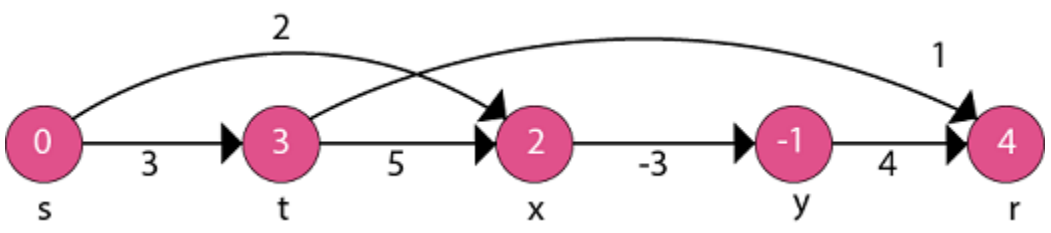
Now, take each vertex in topologically sorted order and relax each edge.



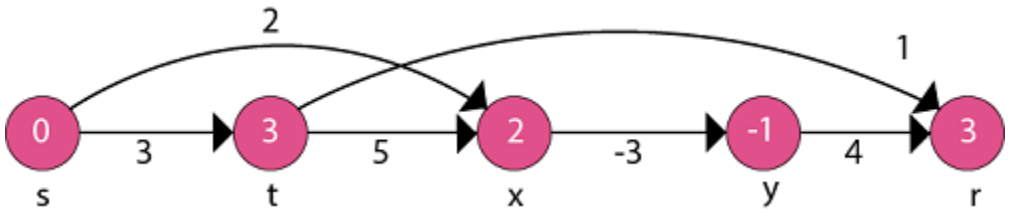
1. adj [s] →t, x
2. 0 + 3 < ∞
3. d [t] ← 3
4. 0 + 2 < ∞
5. d [x] ← 2



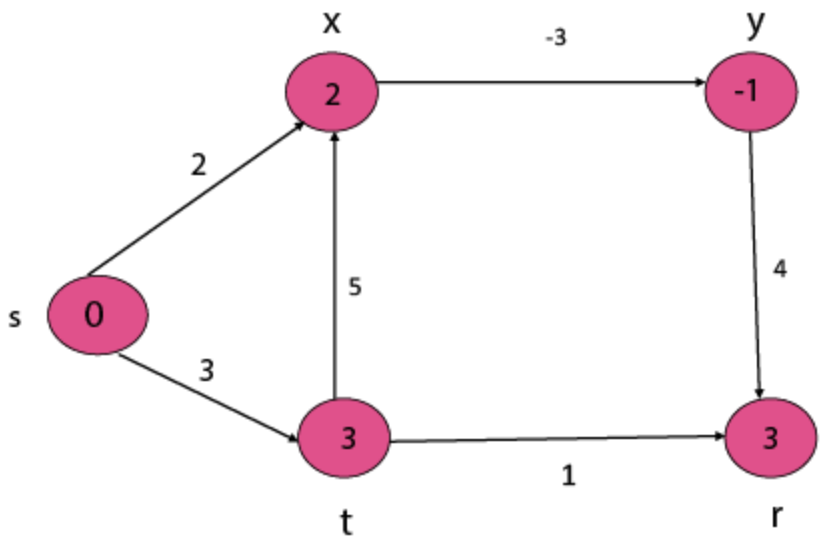
1. adj [t] → r, x
2. 3 + 1 < ∞
3. d [r] ← 4
4. 3 + 5 ≤ 2



1. adj [x] → y
2. 2 - 3 < ∞
3. d [y] ← -1



1. adj [y] → r
2. -1 + 4 < 4
3. 3 <4
4. d [r] ← 3



Thus the Shortest Path is:

1. s to x is 2
2. s to y is -1
3. s to t is 3
4. s to r is 3