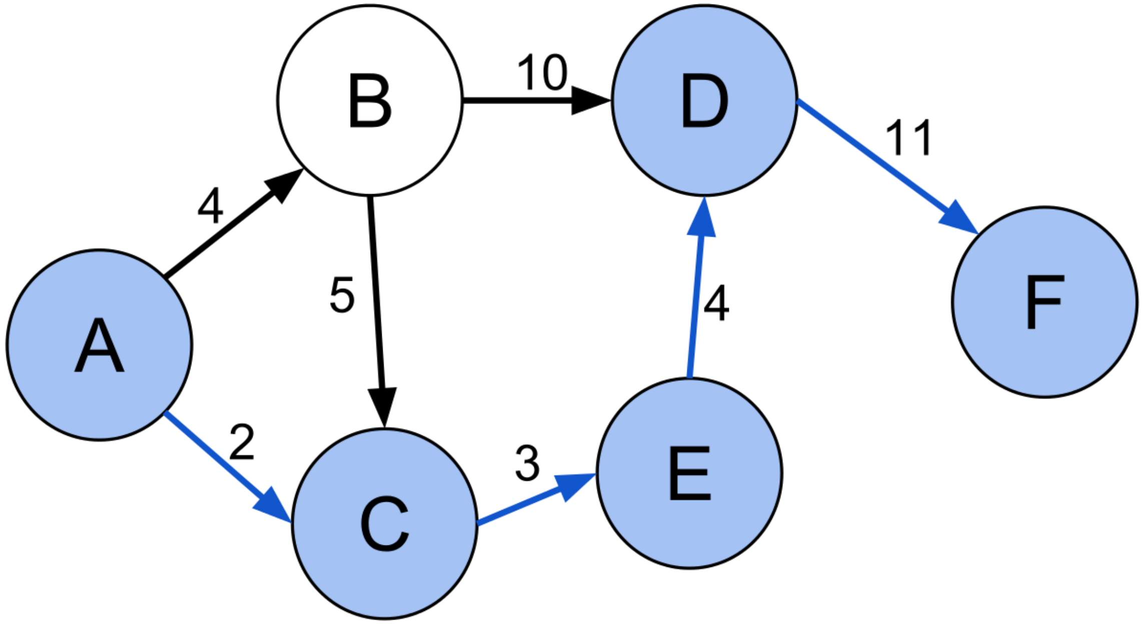
Shortest Path Problem I

SWE2016-44

Problem Statement

Find a path between two vertices in a graph such that the sum of the weights of its constituent edges is minimized.

A graph is a series of nodes connected by edges. Graphs can be weighted and directional.

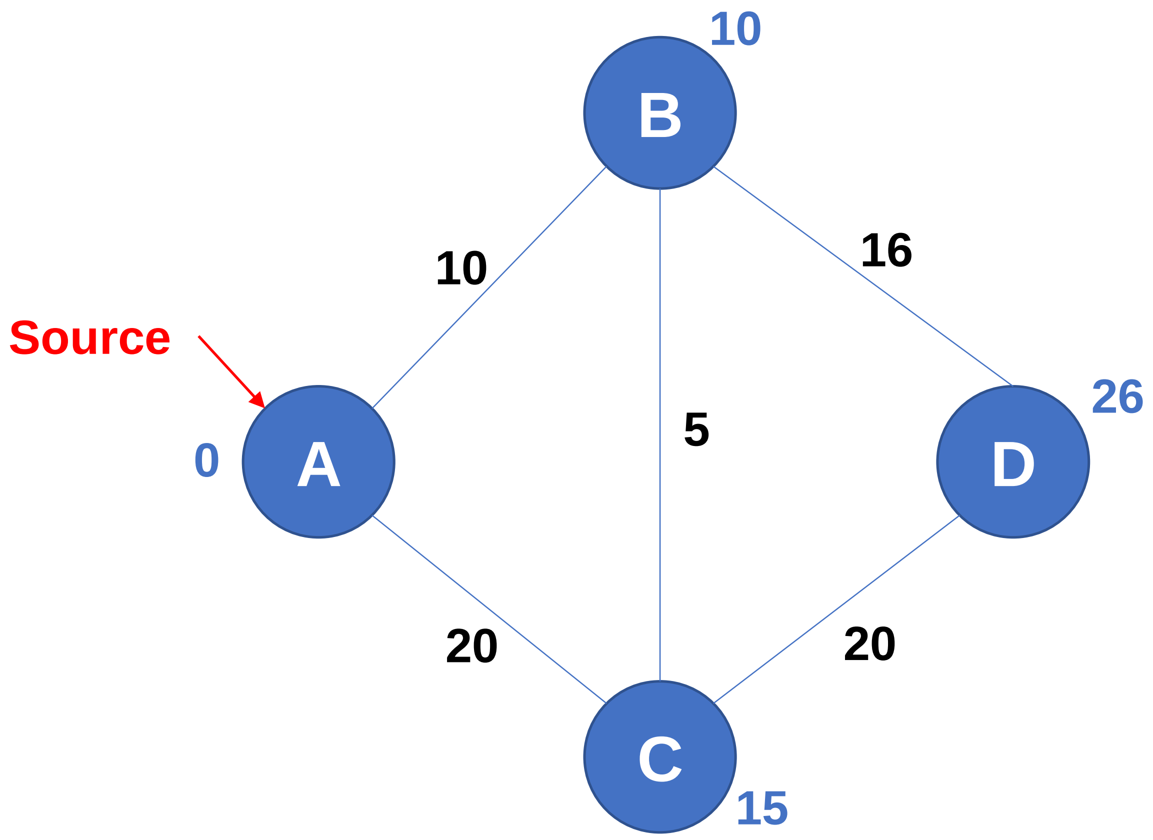


Shortest Path Algorithms

- Dijkstra's algorithm: solves the single-source shortest path problem with non-negative edge weight.
- Bellman–Ford algorithm: solves the single-source problem if edge weights may be negative.
- A* search algorithm: solves for single pair shortest path using heuristics to try to speed up the search.
- Floyd–Warshall algorithm: solves all pairs shortest paths.
- Johnson's algorithm: solves all pairs shortest paths, and may be faster than Floyd–Warshall on sparse graphs.
- Viterbi algorithm: solves the shortest stochastic path problem with an additional probabilistic weight on each node.

Dijkstra's algorithm

Dijkstra's algorithm

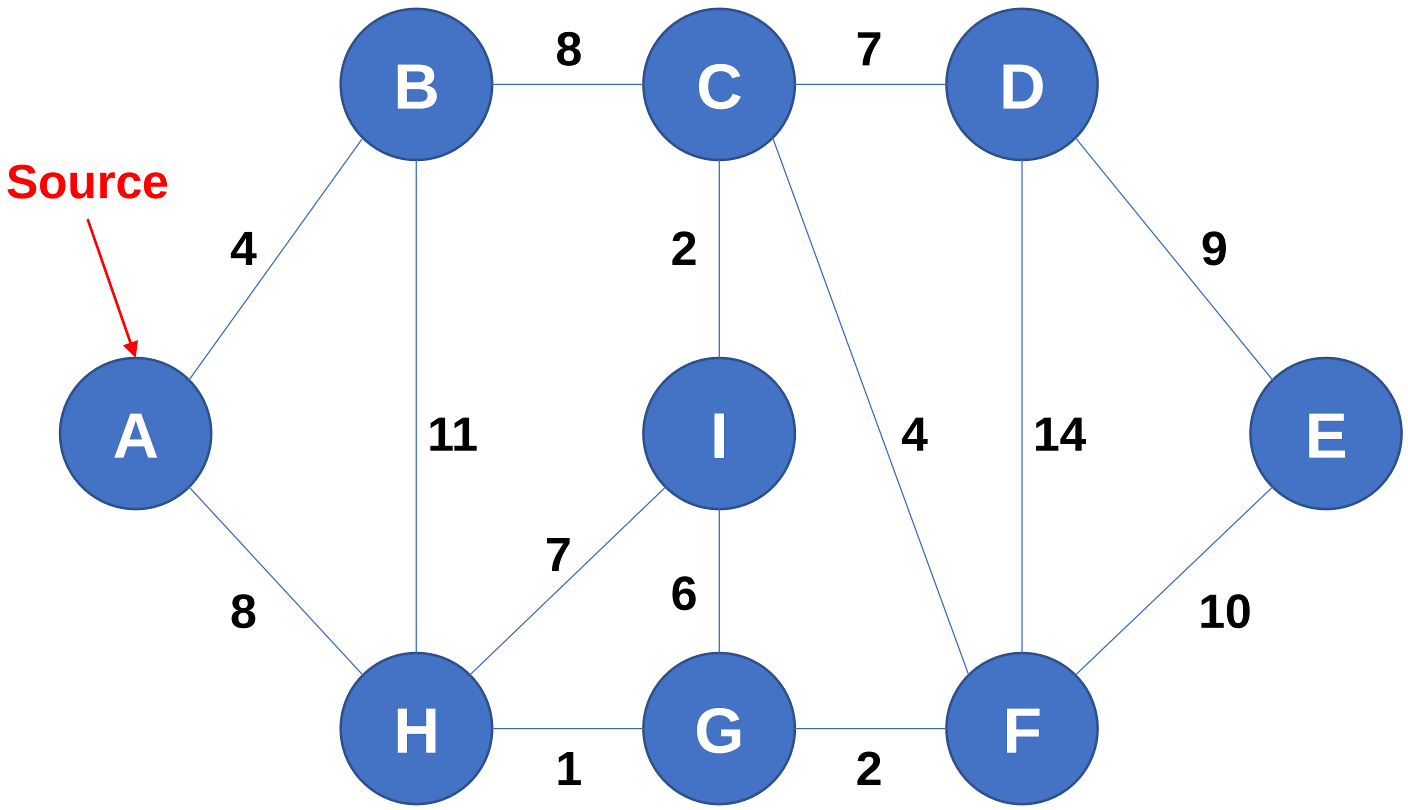


Similar to Prim's Algorithm

Dijkstra's algorithm

1. Create an empty set (*sptSet*) that keeps track of vertices included in shortest path tree
2. Initialize all vertices distances as INFINITE except for the source vertex. Initialize the source distance=0.
3. While *sptSet* doesn’t include all vertices
    1) Pick a vertex u which is not there in *sptSet* and has minimum distance value.
    2) Include u to the set.
    3) Update distance value of all adjacent vertices of u. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if sum of distance value of u and weight of edge u-v, is less than the distance value of v, then update the distance value of v.

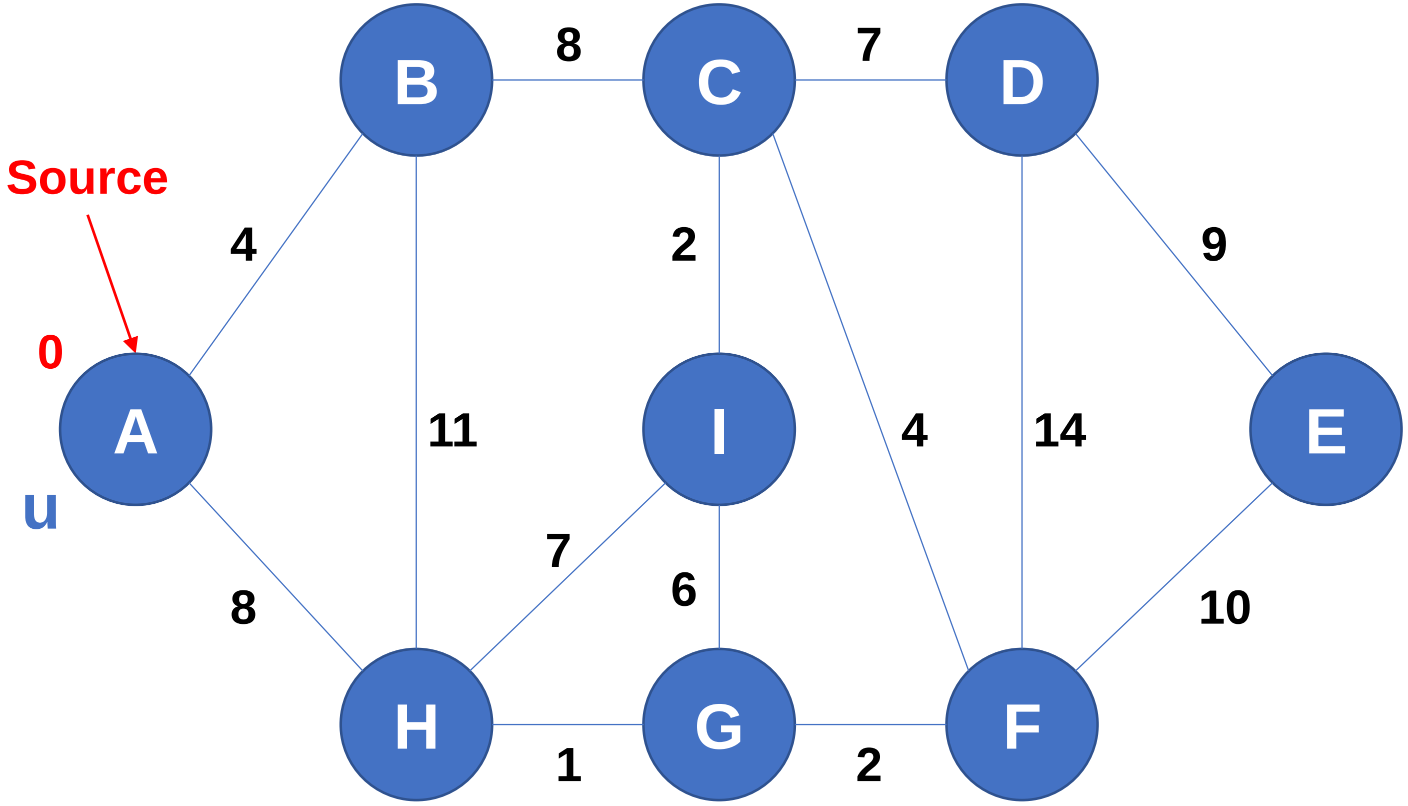
Dijkstra's algorithm



Vertex	d	π
A	0	NIL
B	∞	NIL
C	∞	NIL
D	∞	NIL
E	∞	NIL
F	∞	NIL
G	∞	NIL
H	∞	NIL
I	∞	NIL

distance (d) · parent (π)

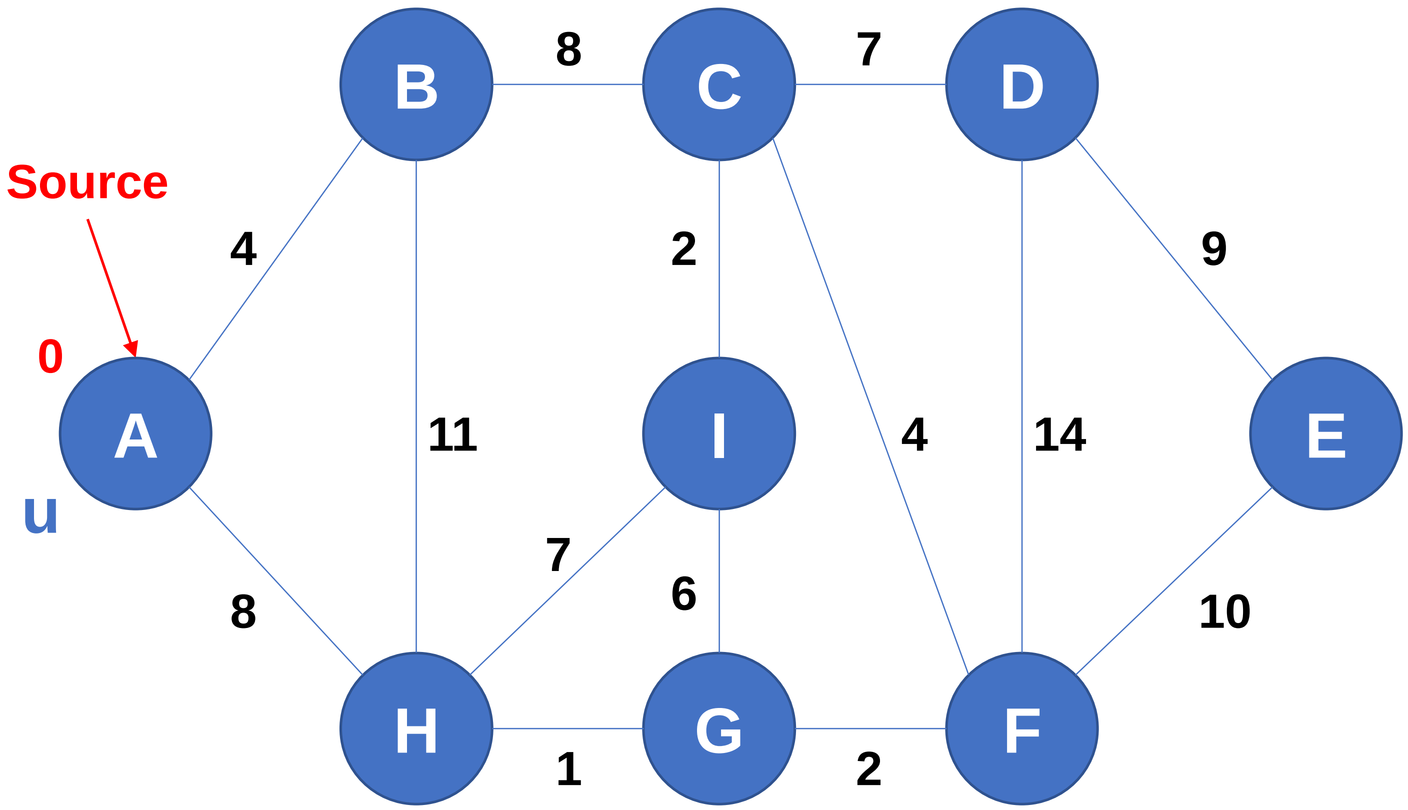
Dijkstra's algorithm



Update B: d[u]+4=4 < ∞
Update H: d[u]+8=8 < ∞

Vertex	d (distance)	π (parent)
~~A~~	0	NIL
B	∞	NIL
C	∞	NIL
D	∞	NIL
E	∞	NIL
F	∞	NIL
G	∞	NIL
H	∞	NIL
I	∞	NIL

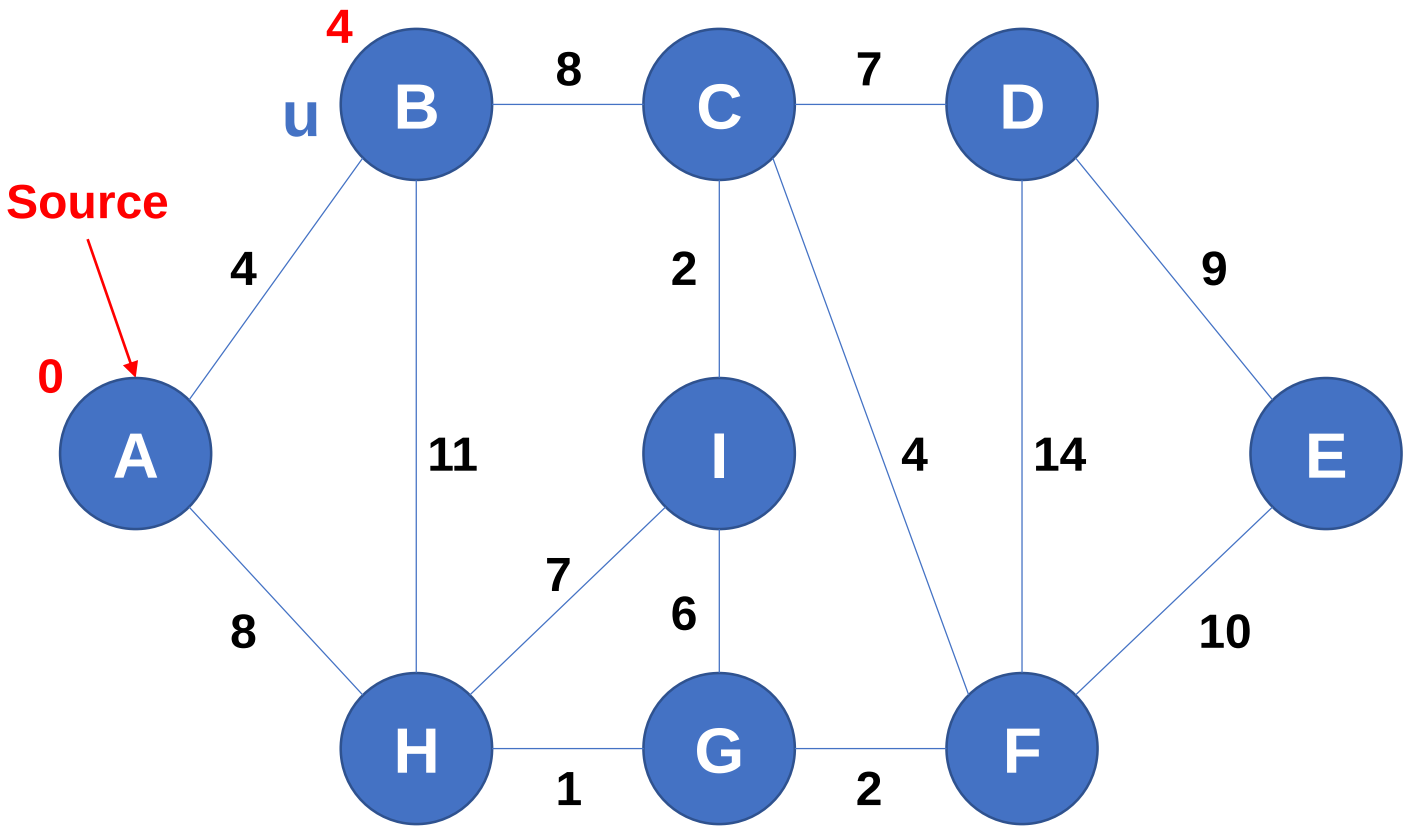
Dijkstra's algorithm



Update B: d[u]+4=4 < ∞
Update H: d[u]+8=8 < ∞

Vertex	d (distance)	π (parent)
~~A~~	0	NIL
B	4	A
C	∞	NIL
D	∞	NIL
E	∞	NIL
F	∞	NIL
G	∞	NIL
H	8	A
I	∞	NIL

Dijkstra's algorithm

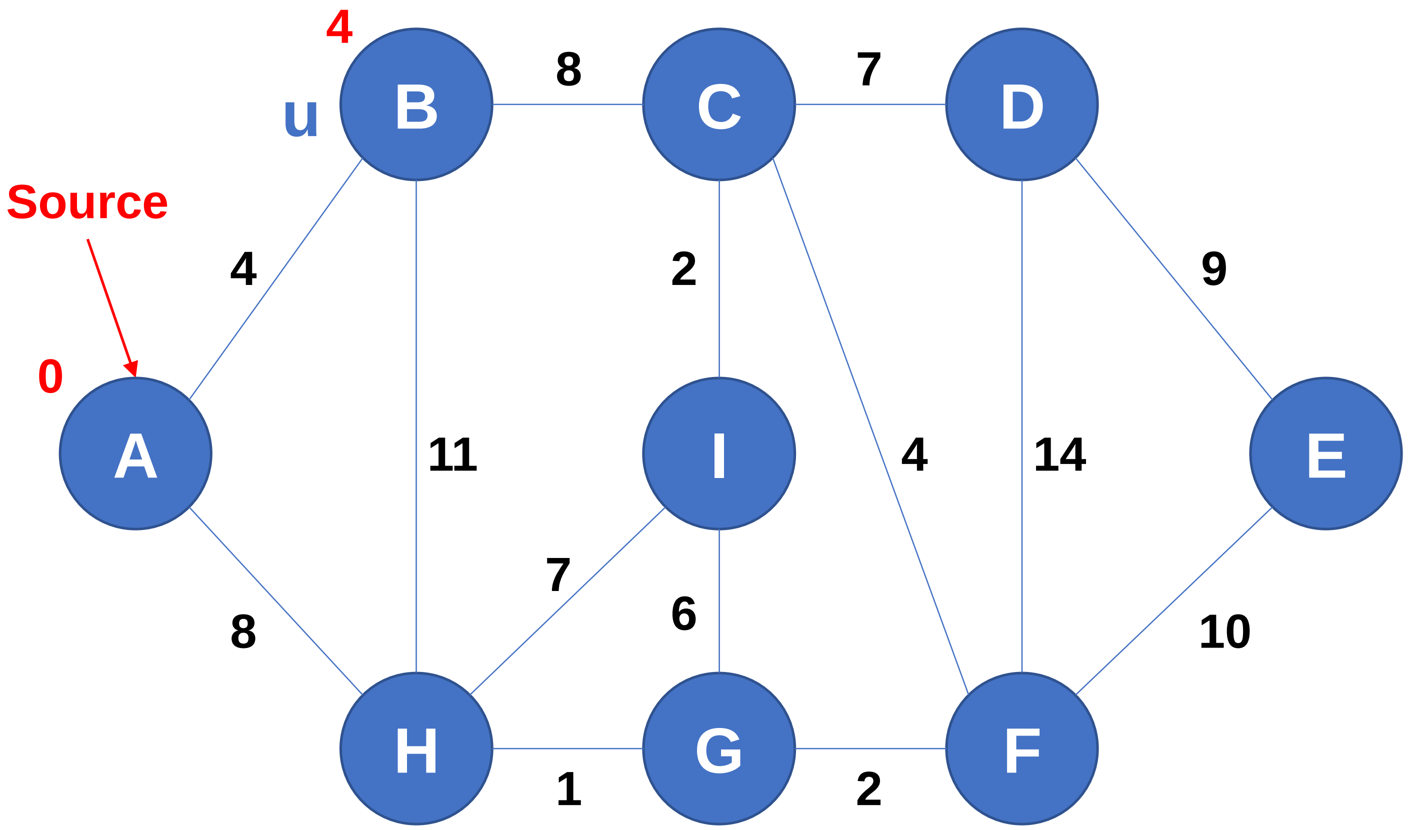


Update C: d[u]+8=12 < ∞
Update H: d[u]+11=15 > 8

distance → d, parent → π

Vertex	d	π
~~A~~	0	NIL
~~B~~	4	A
C	∞	NIL
D	∞	NIL
E	∞	NIL
F	∞	NIL
G	∞	NIL
H	8	A
I	∞	NIL

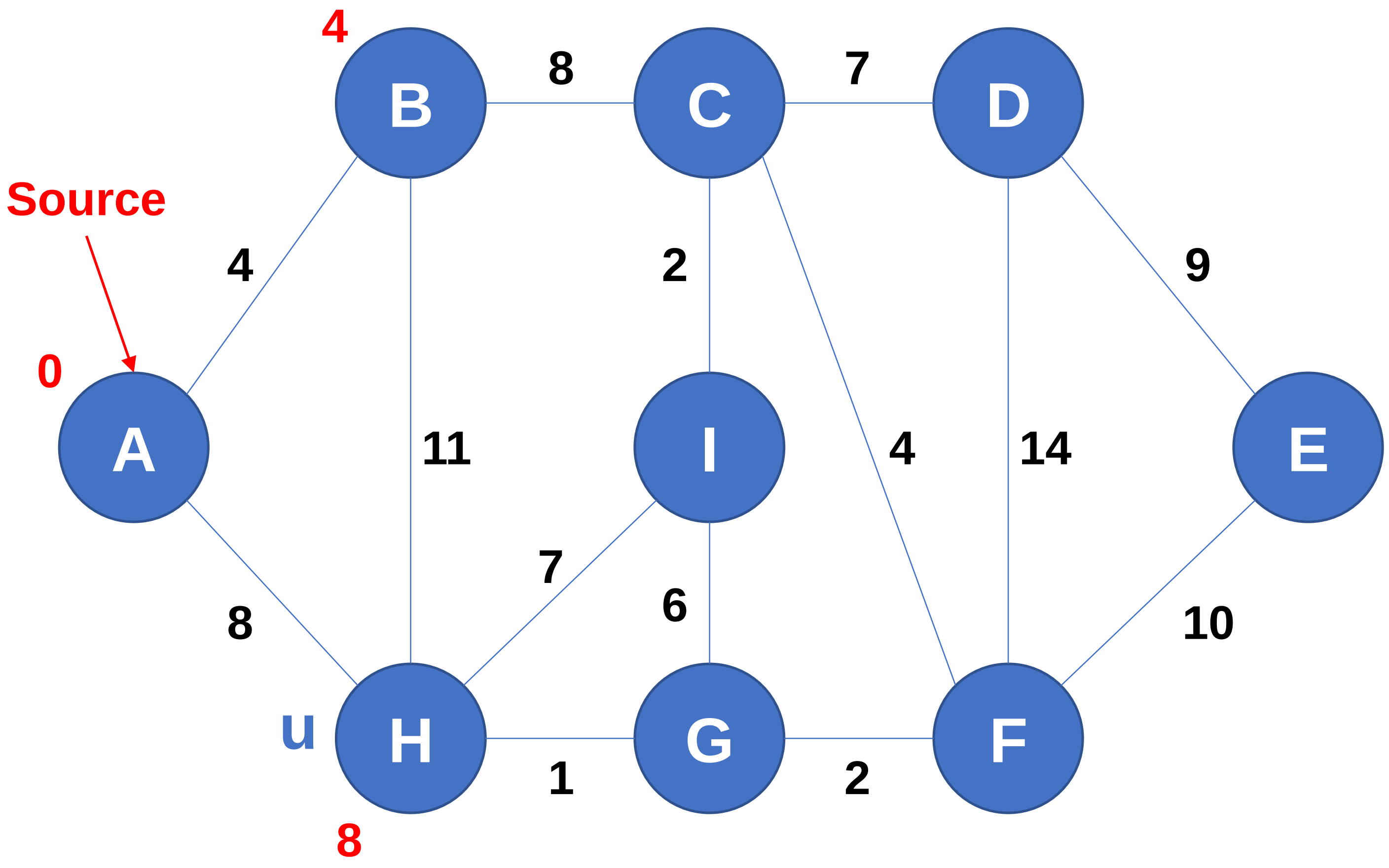
Dijkstra's algorithm



Update C: d[u]+8=12 < ∞
Update H: d[u]+11=15 > 8

Vertex	d (distance)	π (parent)
~~A~~	0	NIL
~~B~~	4	A
C	12	B
D	∞	NIL
E	∞	NIL
F	∞	NIL
G	∞	NIL
H	8	A
I	∞	NIL

Dijkstra's algorithm

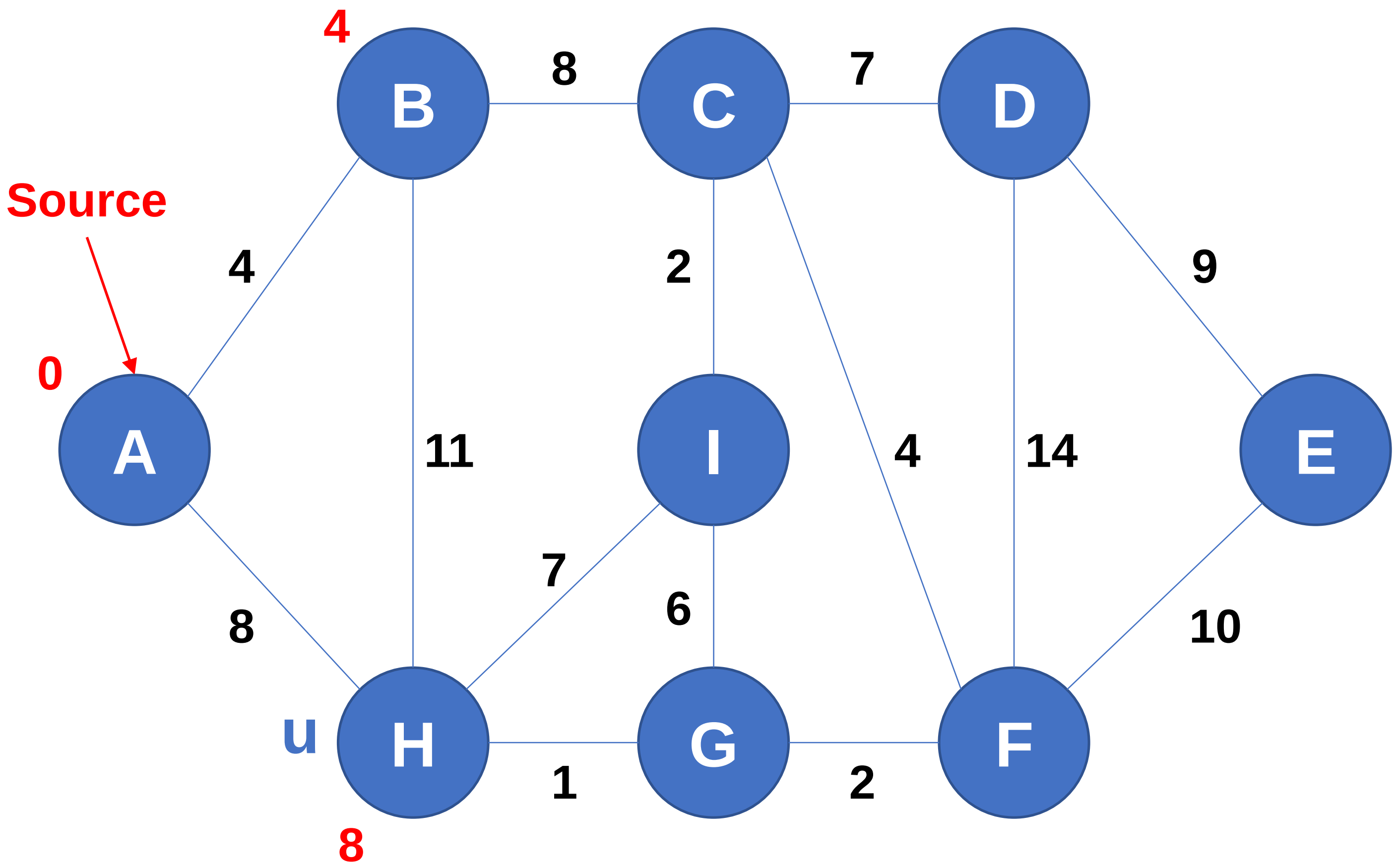


Update G: d[u]+1=9 < ∞

Update I: d[u]+7=15 < ∞

Vertex	d (distance)	π (parent)
~~A~~	0	NIL
~~B~~	4	A
C	12	B
D	∞	NIL
E	∞	NIL
F	∞	NIL
G	∞	NIL
~~H~~	8	A
I	∞	NIL

Dijkstra's algorithm

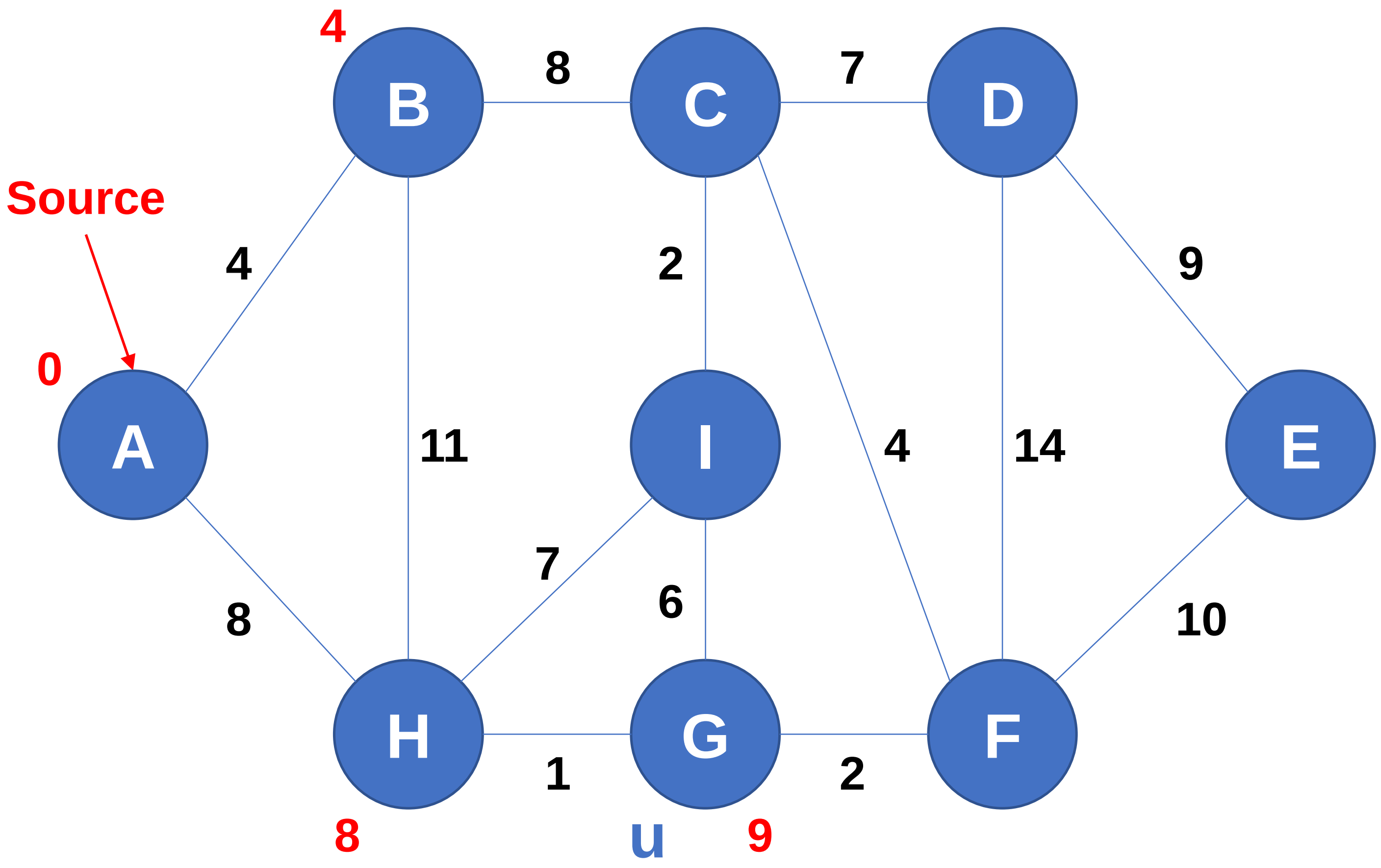


Update G: d[u]+1=9 < ∞

Update I: d[u]+7=15 < ∞

distance	parent	
Vertex	d	π
~~A~~	0	NIL
~~B~~	4	A
C	12	B
D	∞	NIL
E	∞	NIL
F	∞	NIL
G	9	H
~~H~~	8	A
I	15	H

Dijkstra's algorithm

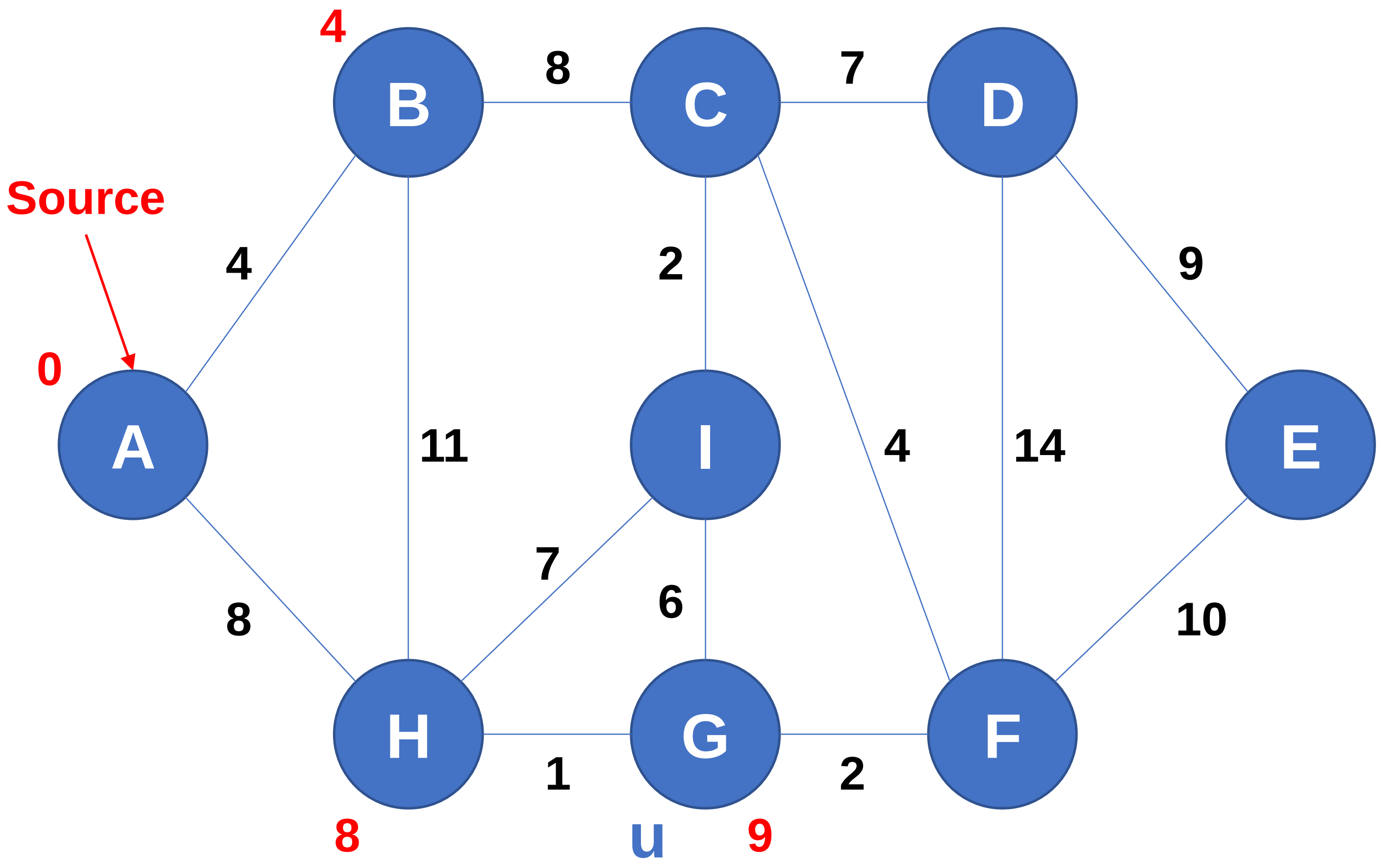


Update F: d[u]+2=11 < ∞
Update I: d[u]+6=15 = 15

distance parent

Vertex	d	π
~~A~~	0	NIL
~~B~~	4	A
C	12	B
D	∞	NIL
E	∞	NIL
F	∞	NIL
~~G~~	9	H
~~H~~	8	A
I	15	H

Dijkstra's algorithm



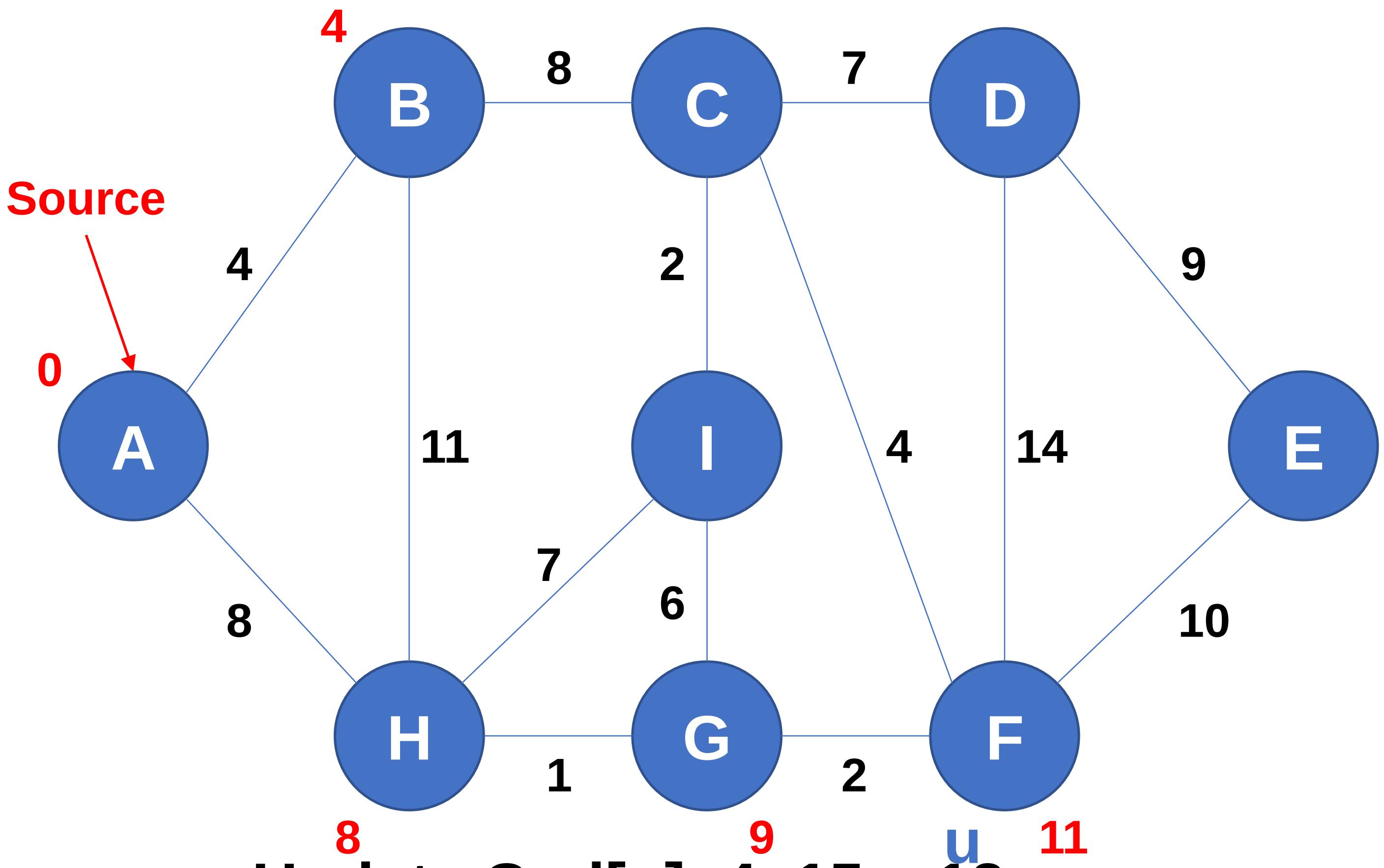
Update F: d[u]+2=11 < ∞

Update I: d[u]+6=15 = 15

distance
parent

Vertex	d	π
~~A~~	0	NIL
~~B~~	4	A
C	12	B
D	∞	NIL
E	∞	NIL
F	11	G
~~G~~	9	H
~~H~~	8	A
I	15	H

Dijkstra's algorithm

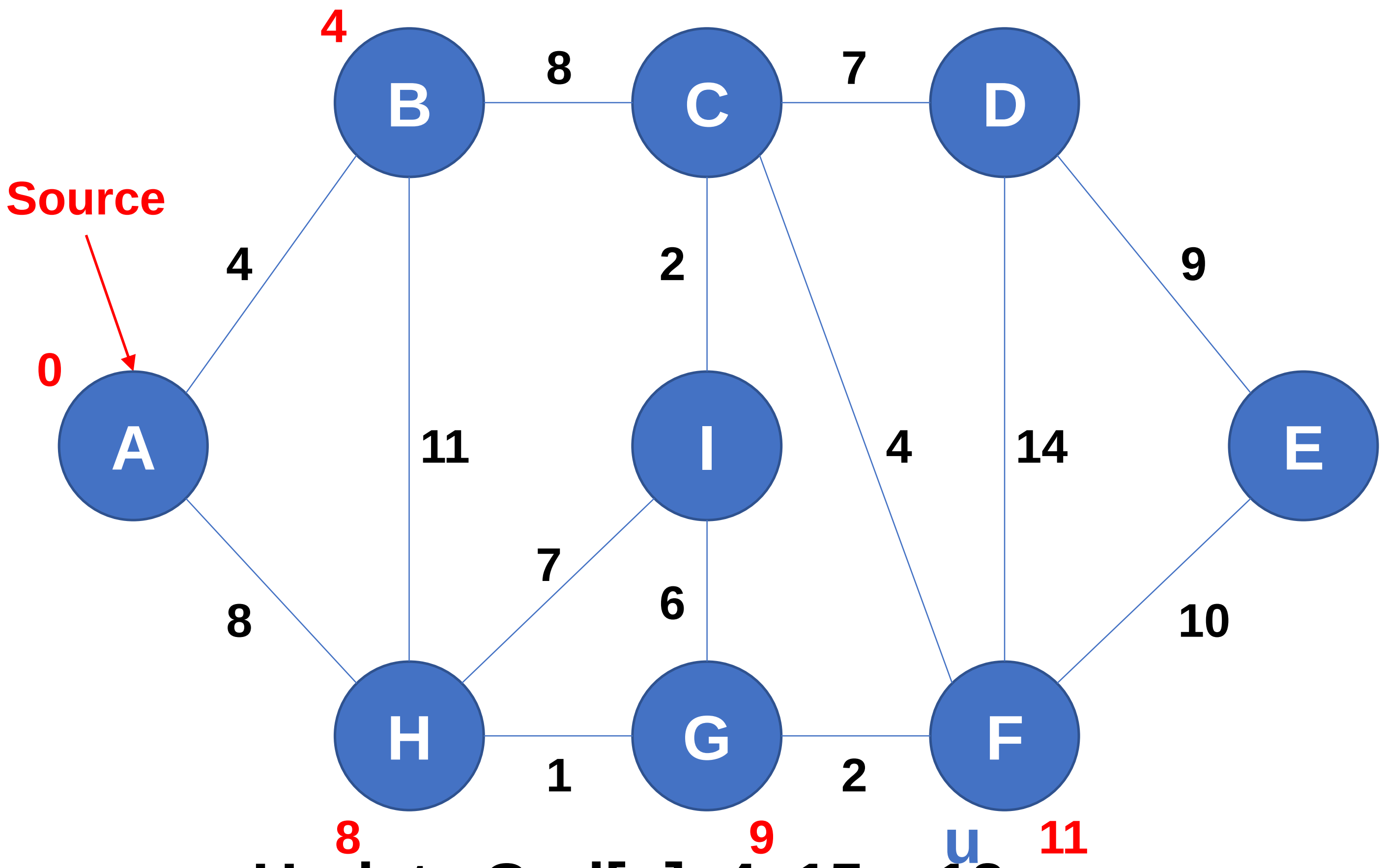


Update C: d[u]+4=15 > 12
Update D: d[u]+14=25 < ∞
Update E: d[u]+10=21 < ∞

distance → d, parent → π

Vertex	d	π
~~A~~	0	NIL
~~B~~	4	A
C	12	B
D	∞	NIL
E	∞	NIL
~~F~~	11	G
~~G~~	9	H
~~H~~	8	A
I	15	H

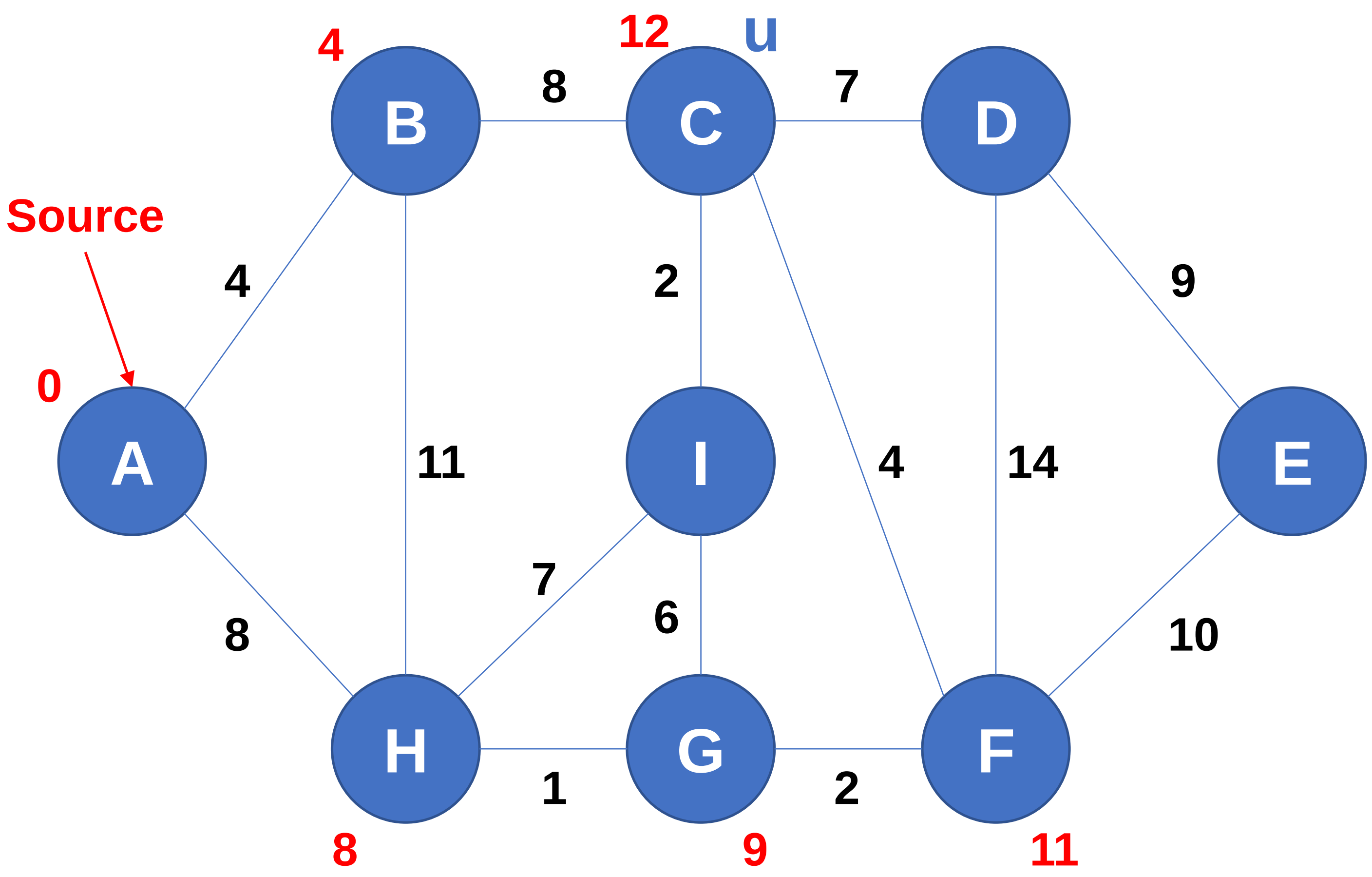
Dijkstra's algorithm



Update C: d[u]+4=15 > 12
Update D: d[u]+14=25 < ∞
Update E: d[u]+10=21 < ∞

Vertex	d (distance)	π (parent)
~~A~~	0	NIL
~~B~~	4	A
C	12	B
D	25	F
E	21	F
~~F~~	11	G
~~G~~	9	H
~~H~~	8	A
I	15	H

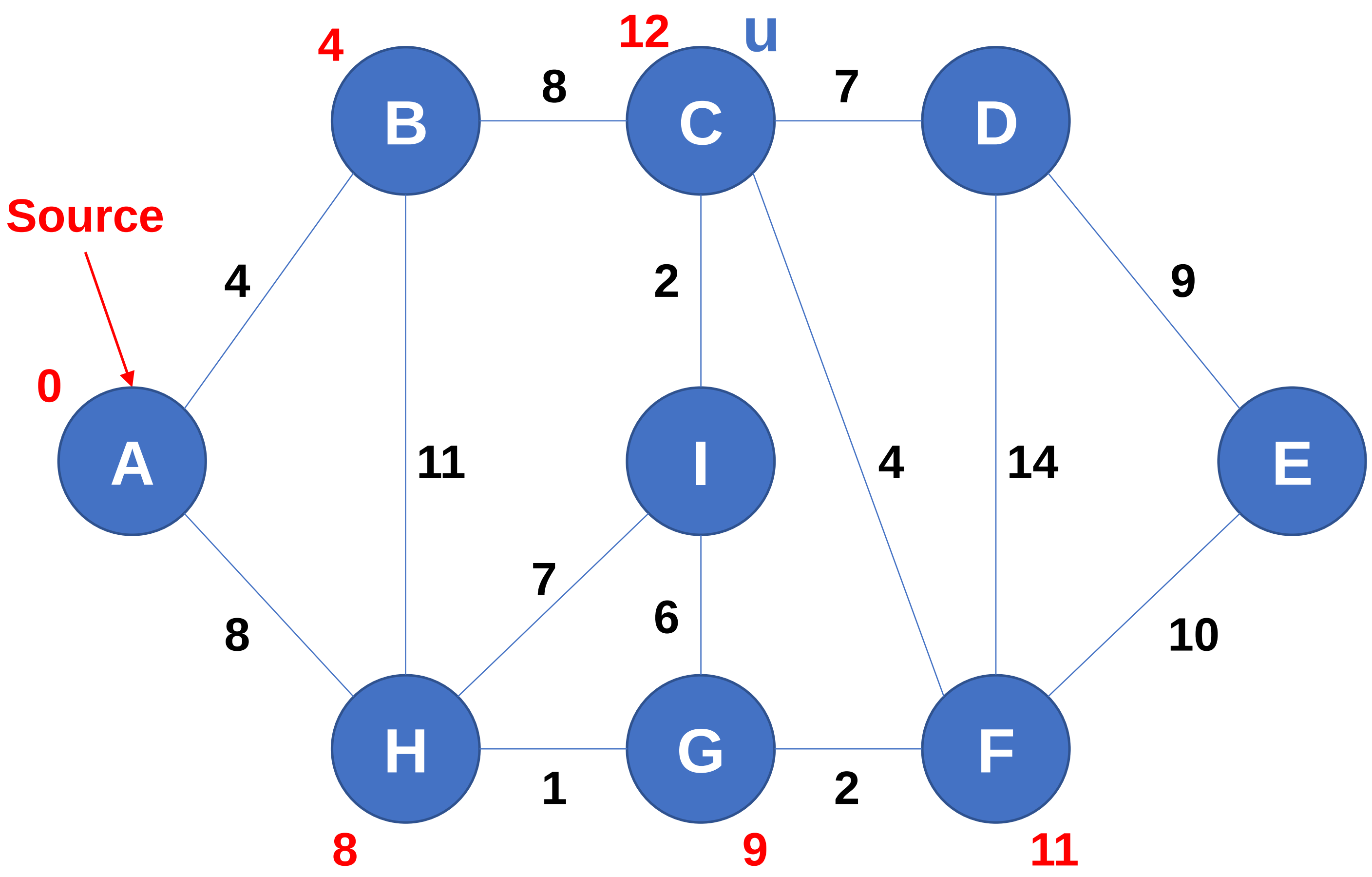
Dijkstra's algorithm



Update D: d[u]+7=19 < 25
Update I: d[u]+2=14 < 15

Vertex	d (distance)	π (parent)
~~A~~	0	NIL
~~B~~	4	A
~~C~~	12	B
D	25	F
E	21	F
~~F~~	11	G
~~G~~	9	H
~~H~~	8	A
I	15	H

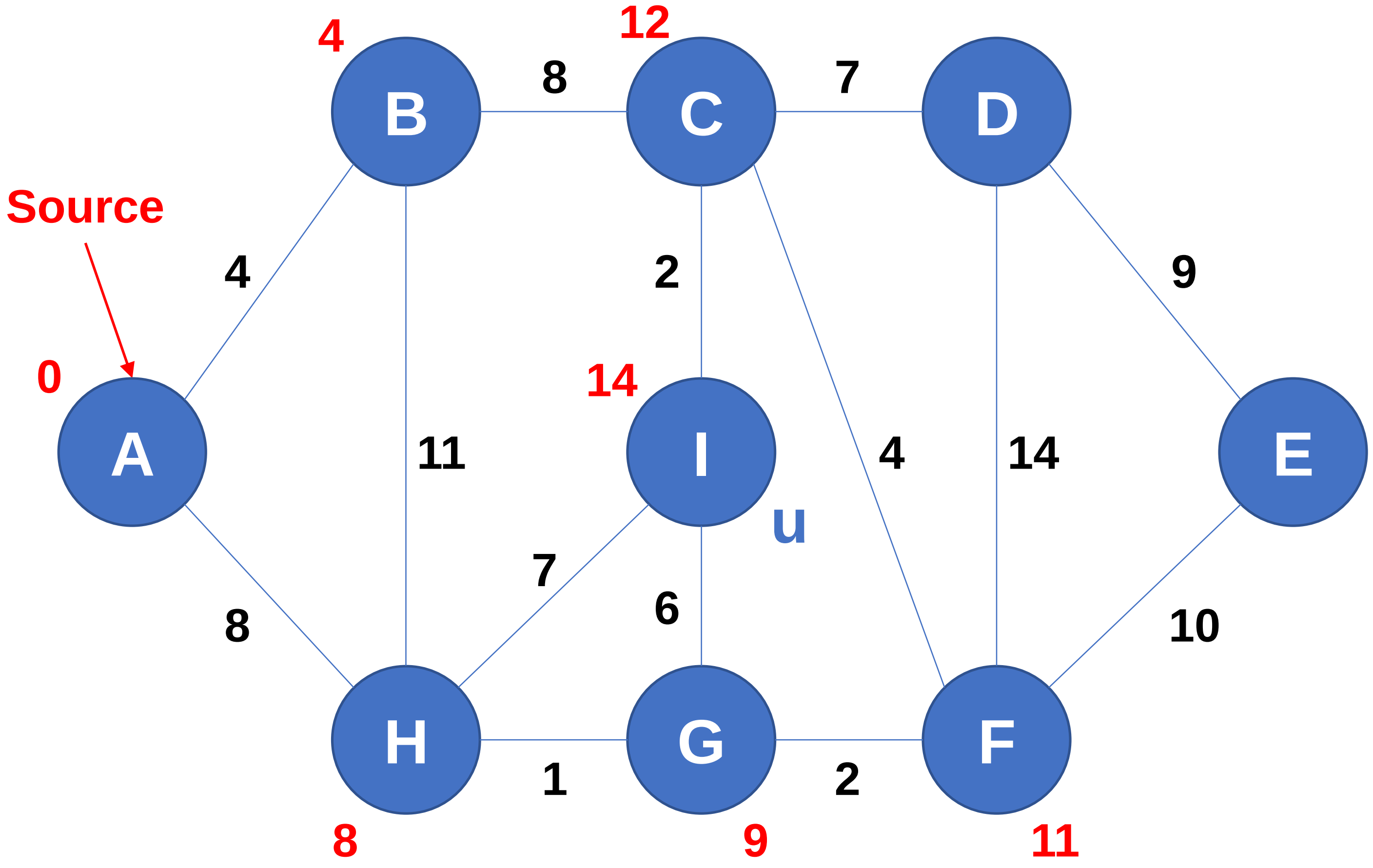
Dijkstra's algorithm



Update D: d[u]+7=19 < 25
Update I: d[u]+2=14 < 15

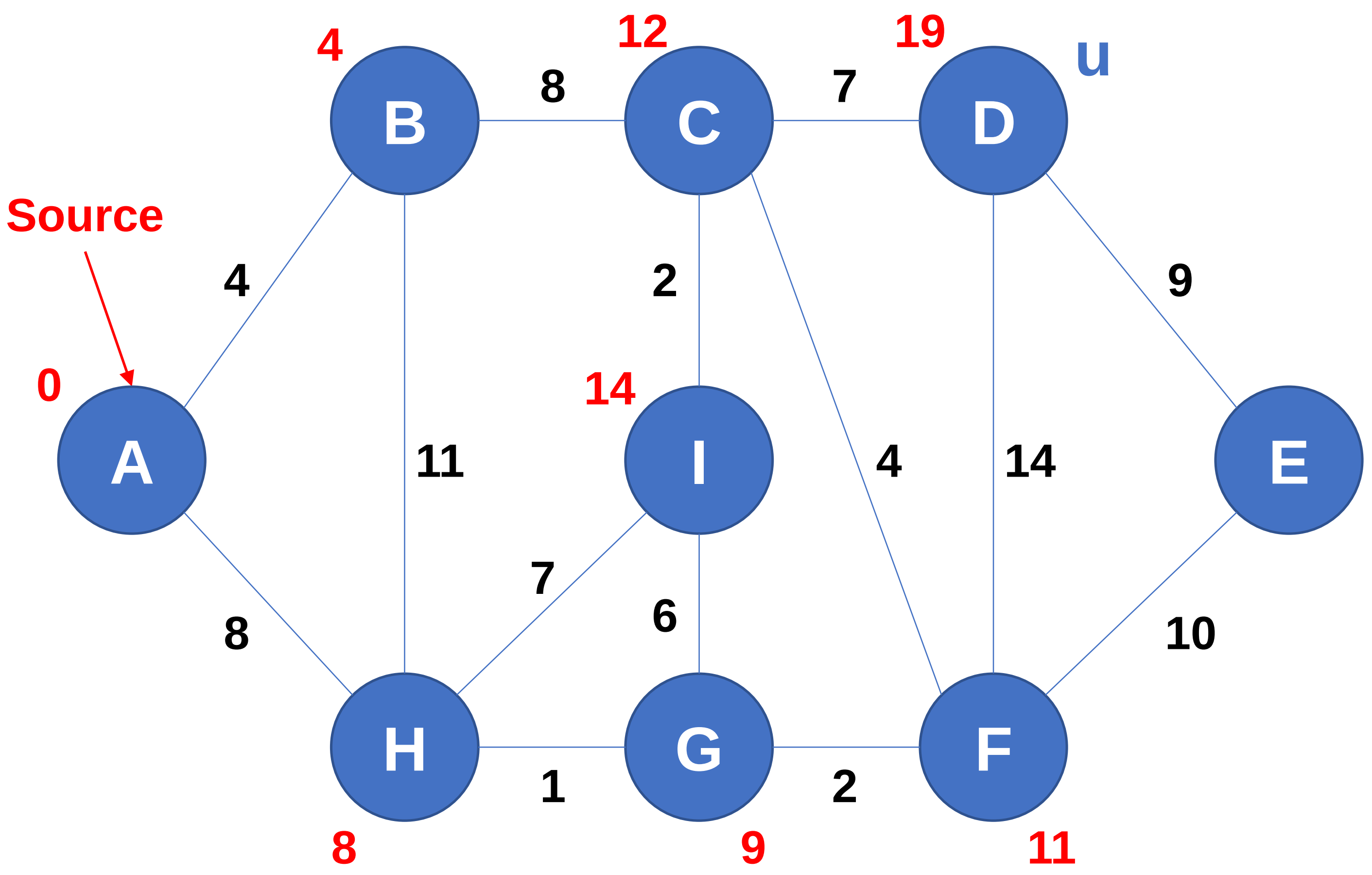
Vertex	d (distance)	π (parent)
~~A~~	0	NIL
~~B~~	4	A
~~C~~	12	B
D	19	C
E	21	F
~~F~~	11	G
~~G~~	9	H
~~H~~	8	A
I	14	C

Dijkstra's algorithm



Vertex	d (distance)	π (parent)
~~A~~	0	NIL
~~B~~	4	A
~~C~~	12	B
D	19	C
E	21	F
~~F~~	11	G
~~G~~	9	H
~~H~~	8	A
~~I~~	14	C

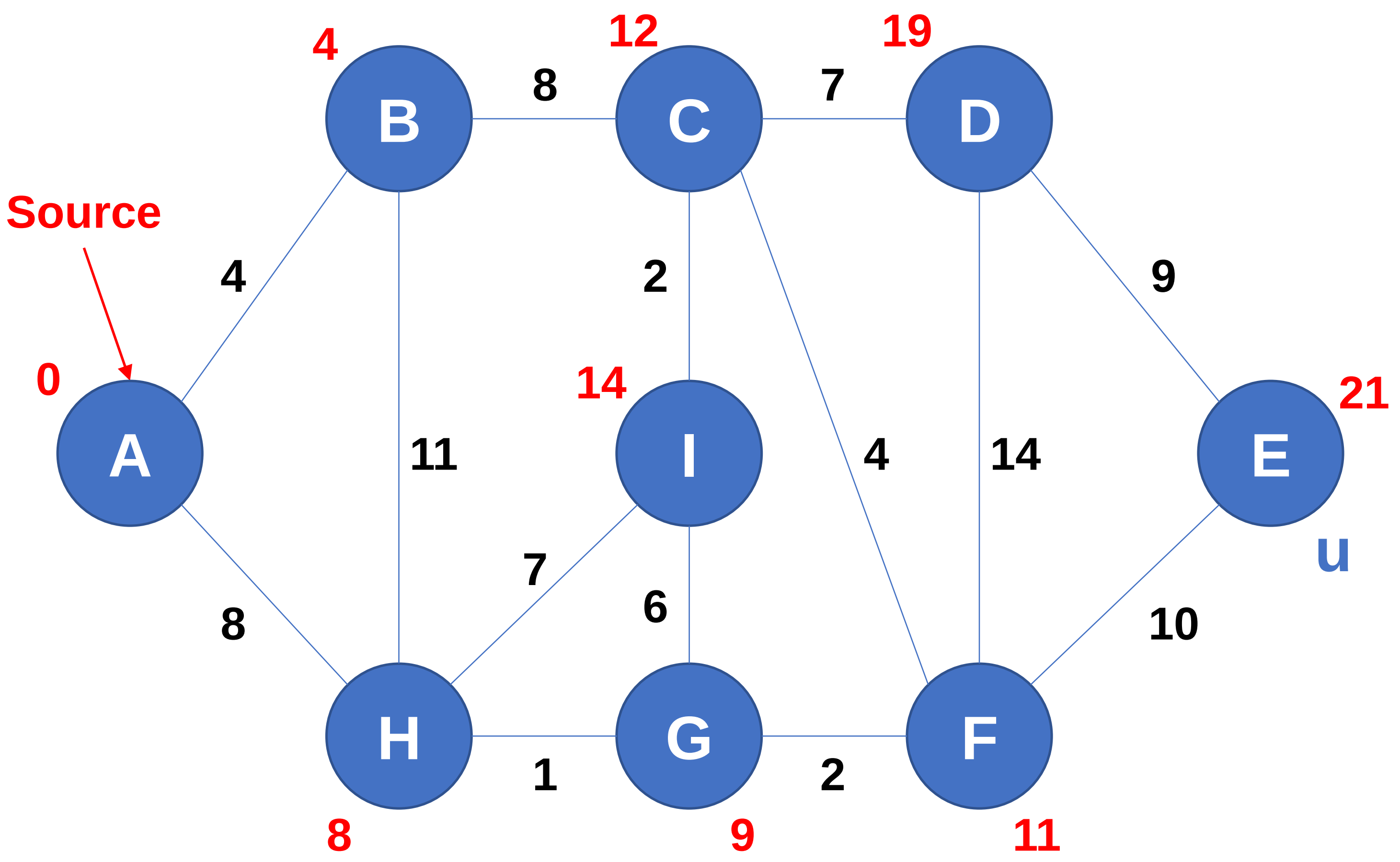
Dijkstra's algorithm



Vertex	d (distance)	π (parent)
~~A~~	0	NIL
~~B~~	4	A
~~C~~	12	B
~~D~~	19	C
E	21	F
~~F~~	11	G
~~G~~	9	H
~~H~~	8	A
~~I~~	14	C

Update E: d[u]+9=28 > 21

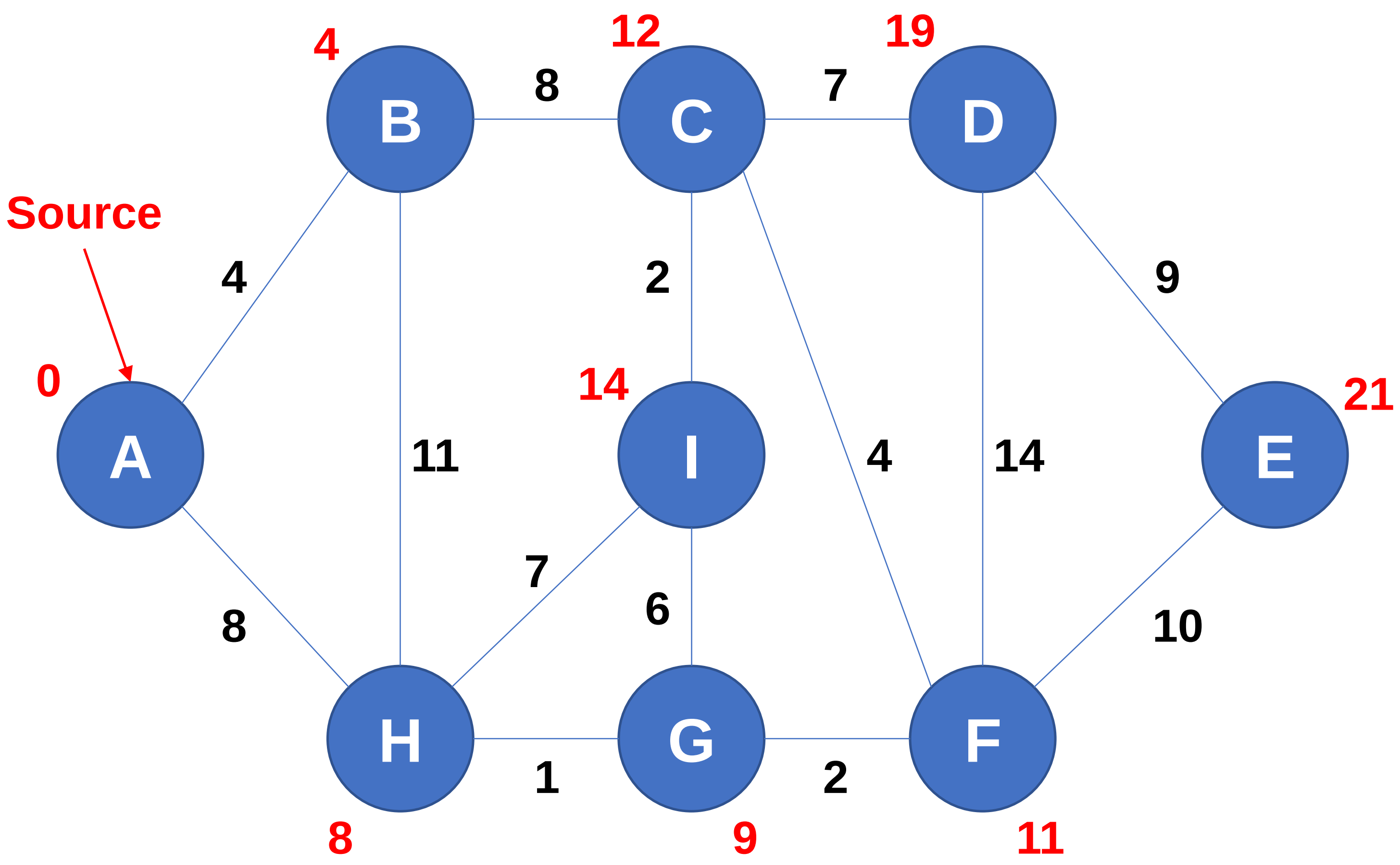
Dijkstra's algorithm



distance parent

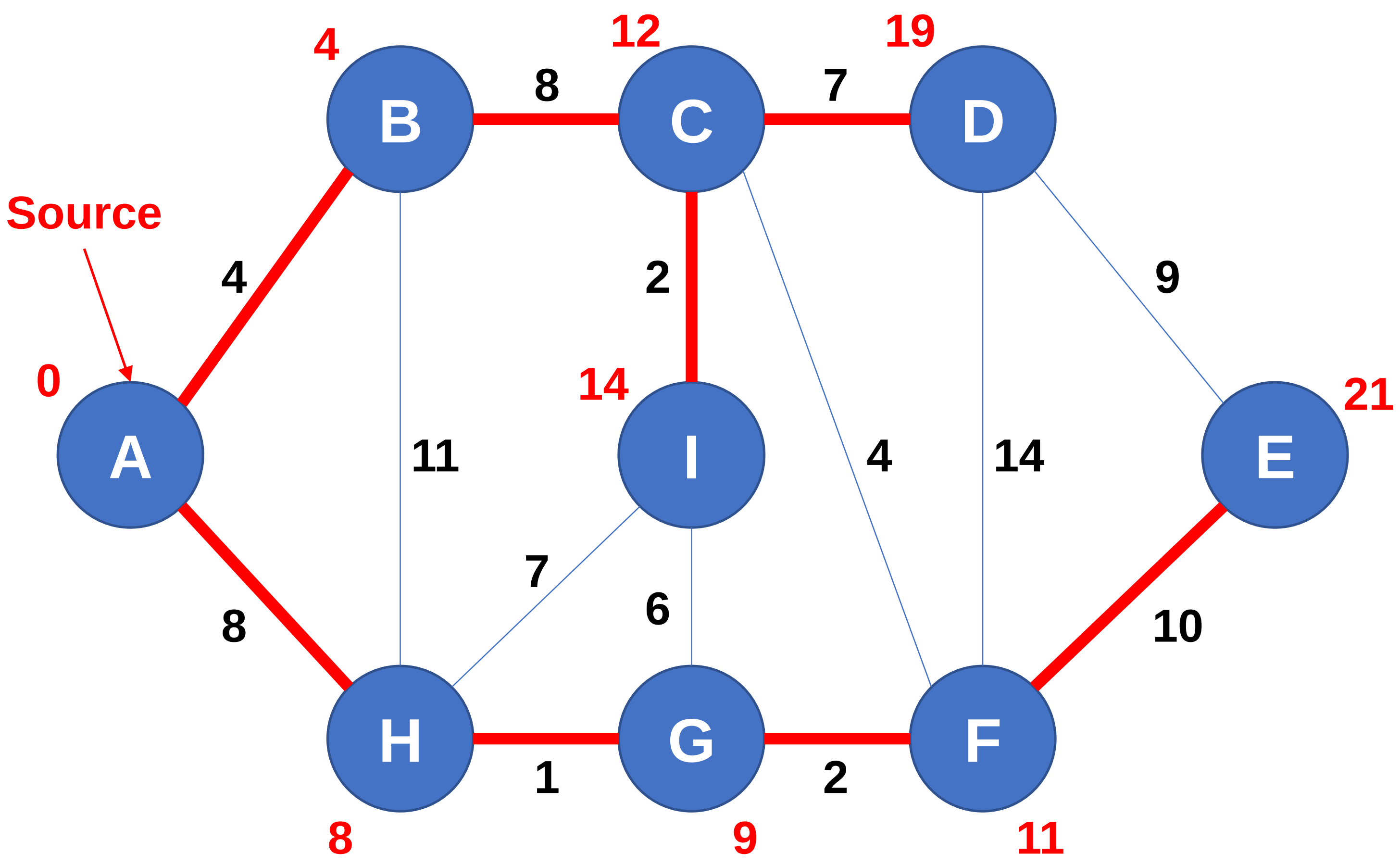
Vertex	d	π
~~A~~	0	NIL
~~B~~	4	A
~~C~~	12	B
~~D~~	19	C
~~E~~	21	F
~~F~~	11	G
~~G~~	9	H
~~H~~	8	A
~~I~~	14	C

Dijkstra's algorithm



Vertex	d (distance)	π (parent)
~~A~~	0	NIL
~~B~~	4	A
~~C~~	12	B
~~D~~	19	C
~~E~~	21	F
~~F~~	11	G
~~G~~	9	H
~~H~~	8	A
~~I~~	14	C

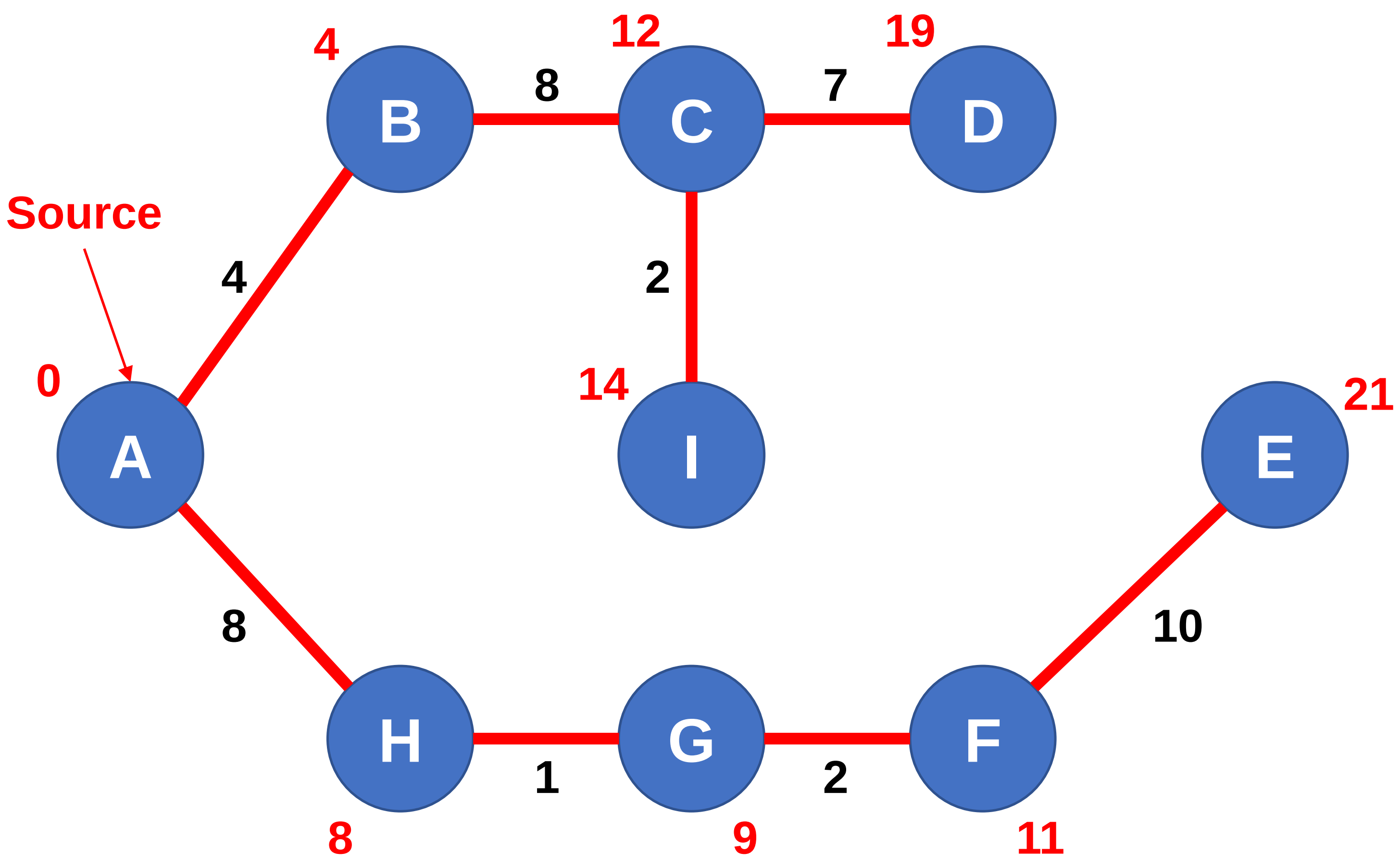
Dijkstra's algorithm



Vertex	d	π
~~A~~	0	NIL
~~B~~	4	A
~~C~~	12	B
~~D~~	19	C
~~E~~	21	F
~~F~~	11	G
~~G~~	9	H
~~H~~	8	A
~~I~~	14	C

distance (d), parent (π)

Dijkstra's algorithm



distance　parent

Vertex	d	π
~~A~~	0	NIL
~~B~~	4	A
~~C~~	12	B
~~D~~	19	C
~~E~~	21	F
~~F~~	11	G
~~G~~	9	H
~~H~~	8	A
~~I~~	14	C

Complexity

Time Complexity: $O(V^2)$

If the input graph is represented using adjacency list, it can be reduced to $O(E \log V)$ with the help of binary heap.

Pseudo-Code

$d[s] \leftarrow 0$
for each $v \in V - \{s\}$:
    do $d[v] \leftarrow \infty$
$S \leftarrow \emptyset$
$Q \leftarrow V$
while $Q \neq \emptyset$:
    do $u \leftarrow \mathrm{Extract} - \mathrm{Min}(Q)$:
        $S \leftarrow S \cup \{u\}$
        for each $v \in Adj[u]$:
            if $d[v] > d[u] + w(u, v)$:
                $d[v] = d[u] + w(u, v)$

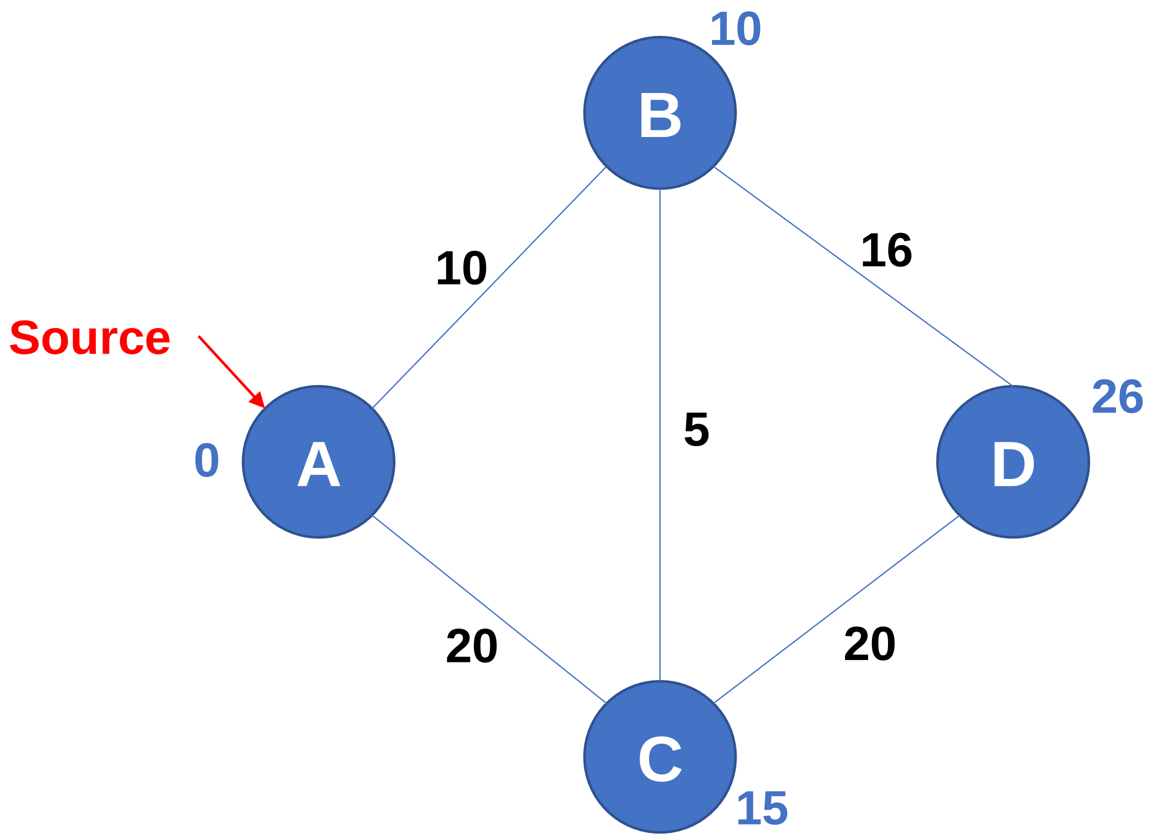
Implementation

```
int minDistance(int dist[], bool sptSet[])
{
    // Initialize min value
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++)
        if (sptSet[v] == false && dist[v] <= min)
            min = dist[v], min_index = v;

    return min_index;
}

int main()
{
    /* Let us create the example graph discussed above */
    int graph[V][V] = { { 0, 4, 0, 0, 0, 0, 0, 8, 0 },
                        { 4, 0, 8, 0, 0, 0, 0, 11, 0 },
                        { 0, 8, 0, 7, 0, 4, 0, 0, 2 },
                        { 0, 0, 7, 0, 9, 14, 0, 0, 0 },
                        { 0, 0, 0, 9, 0, 10, 0, 0, 0 },
                        { 0, 0, 4, 14, 10, 0, 2, 0, 0 },
                        { 0, 0, 0, 0, 0, 2, 0, 1, 6 },
                        { 8, 11, 0, 0, 0, 0, 1, 0, 7 },
                        { 0, 0, 2, 0, 0, 0, 6, 7, 0 } };

    dijkstra(graph, 0);

    return 0;
}
```

```
// Function that implements Dijkstra's single source shortest path algorithm
// for a graph represented using adjacency matrix representation
void dijkstra(int graph[V][V], int src)
{
    int dist[V]; // The output array.  dist[i] will hold the shortest
    // distance from src to i

    bool sptSet[V]; // sptSet[i] will be true if vertex i is included in shortest
    // path tree or shortest distance from src to i is finalized

    // Initialize all distances as INFINITE and stpSet[] as false
    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX, sptSet[i] = false;

    // Distance of source vertex from itself is always 0
    dist[src] = 0;

    // Find shortest path for all vertices
    for (int count = 0; count < V - 1; count++) {
        // Pick the minimum distance vertex from the set of vertices not
        // yet processed. u is always equal to src in the first iteration.
        int u = minDistance(dist, sptSet);

        // Mark the picked vertex as processed
        sptSet[u] = true;

        // Update dist value of the adjacent vertices of the picked vertex.
        for (int v = 0; v < V; v++)

            // Update dist[v] only if is not in sptSet, there is an edge from
            // u to v, and total weight of path from src to  v through u is
            // smaller than current value of dist[v]
            if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX
                && dist[u] + graph[u][v] < dist[v])
                dist[v] = dist[u] + graph[u][v];
    }

    // print the constructed distance array
    printSolution(dist);
}
```

Bellman Ford's algorithm

Bellman Ford's algorithm

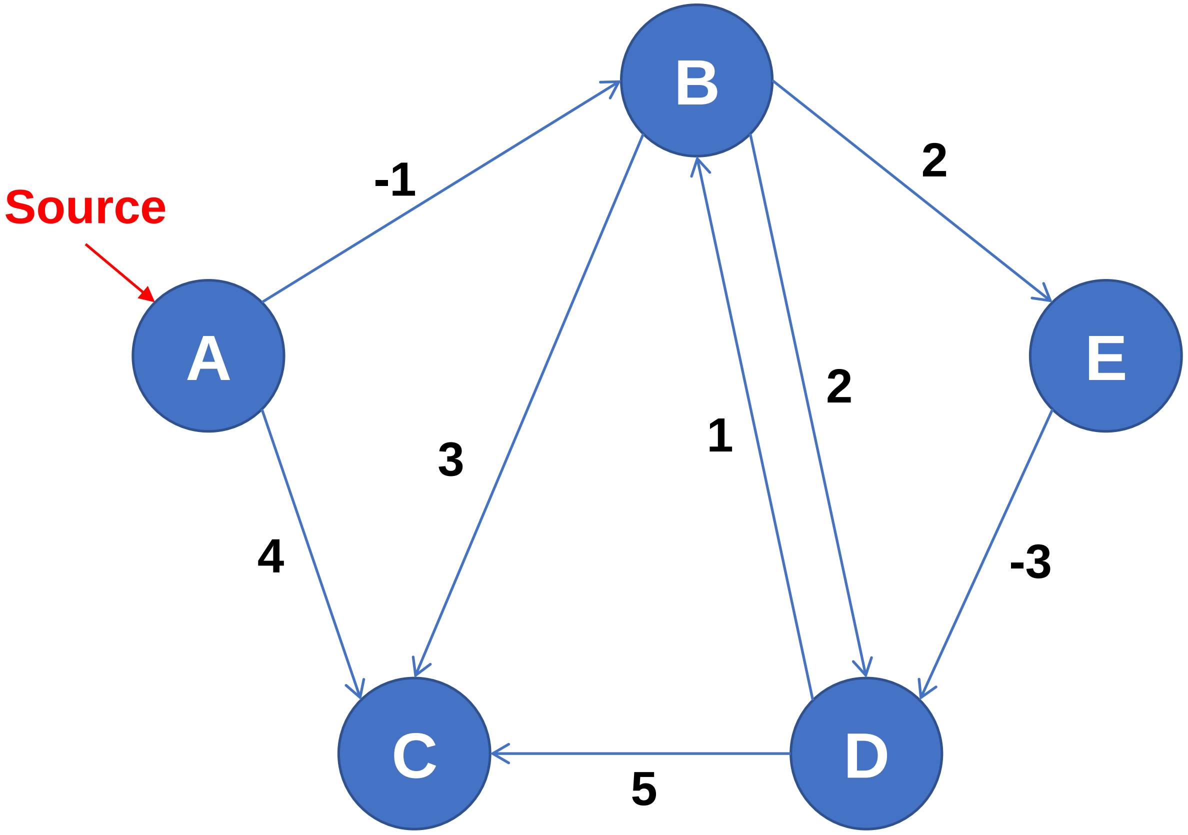


Similar to Prim's Algorithm

Bellman Ford's algorithm

1. Initialize distances from source to all vertices as infinite and distance to source itself as 0. Create an array dist[] of size |V| with all values as infinite except dist[src] where src is source vertex.
2. Calculate shortest distances. Do following |V|-1 times for each edge u-v:
   1) If dist[v] > dist[u]+weight of edge uv, then update dist[v] (=dist[u]+weight of edge uv).
3. This step reports if there is a negative weight cycle in graph. Do following for each edge u-v
   1) If dist[v] > dist[u]+weight of edge uv, then “Graph contains negative weight cycle”

Bellman Ford's algorithm



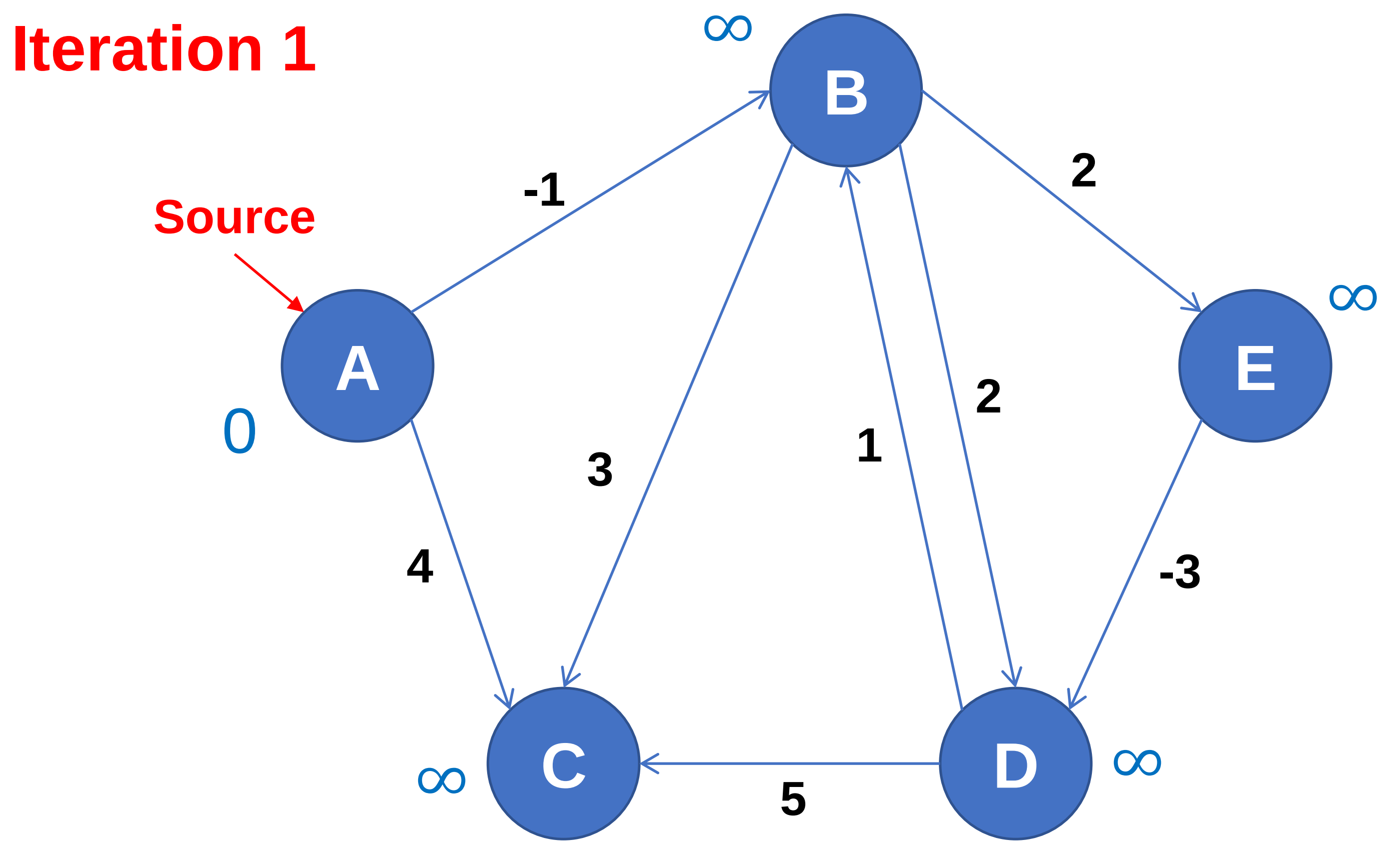
distance ↑ d   parent ↑ π

Vertex	d	π
A	0	NIL
B	∞	NIL
C	∞	NIL
D	∞	NIL
E	∞	NIL

Let all edges are processed in following order:
(B, E), (D, B), (B, D), (A, B), (A, C), (D, C), (B, C), (E, D)

Bellman Ford's algorithm

Iteration 1



distance ↑ d | parent ↑ π

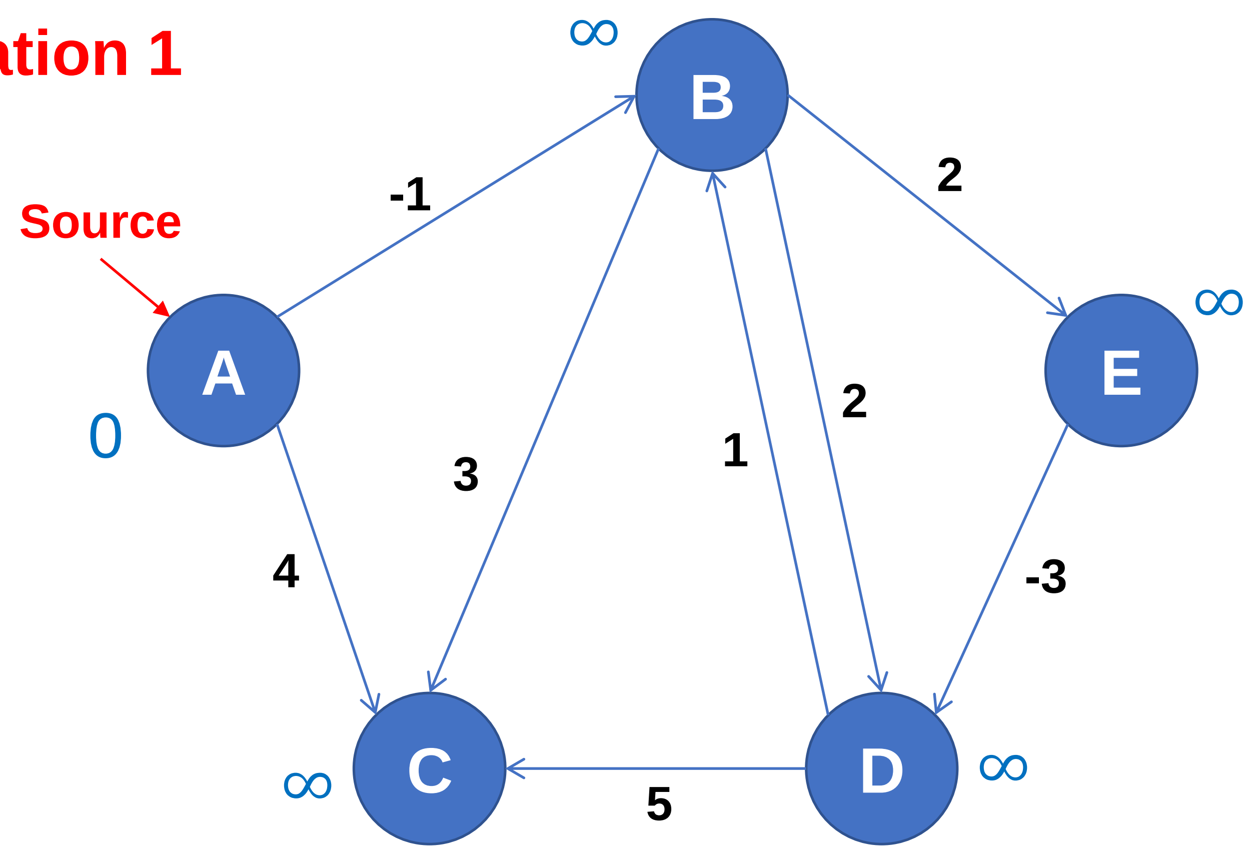
Vertex	d	π
A	0	NIL
B	∞	NIL
C	∞	NIL
D	∞	NIL
E	∞	NIL

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

(B, E), (D, B), (B, D): d[u]+edge(u,v)=∞ = ∞

Bellman Ford's algorithm

Iteration 1



distance ↑ parent ↑

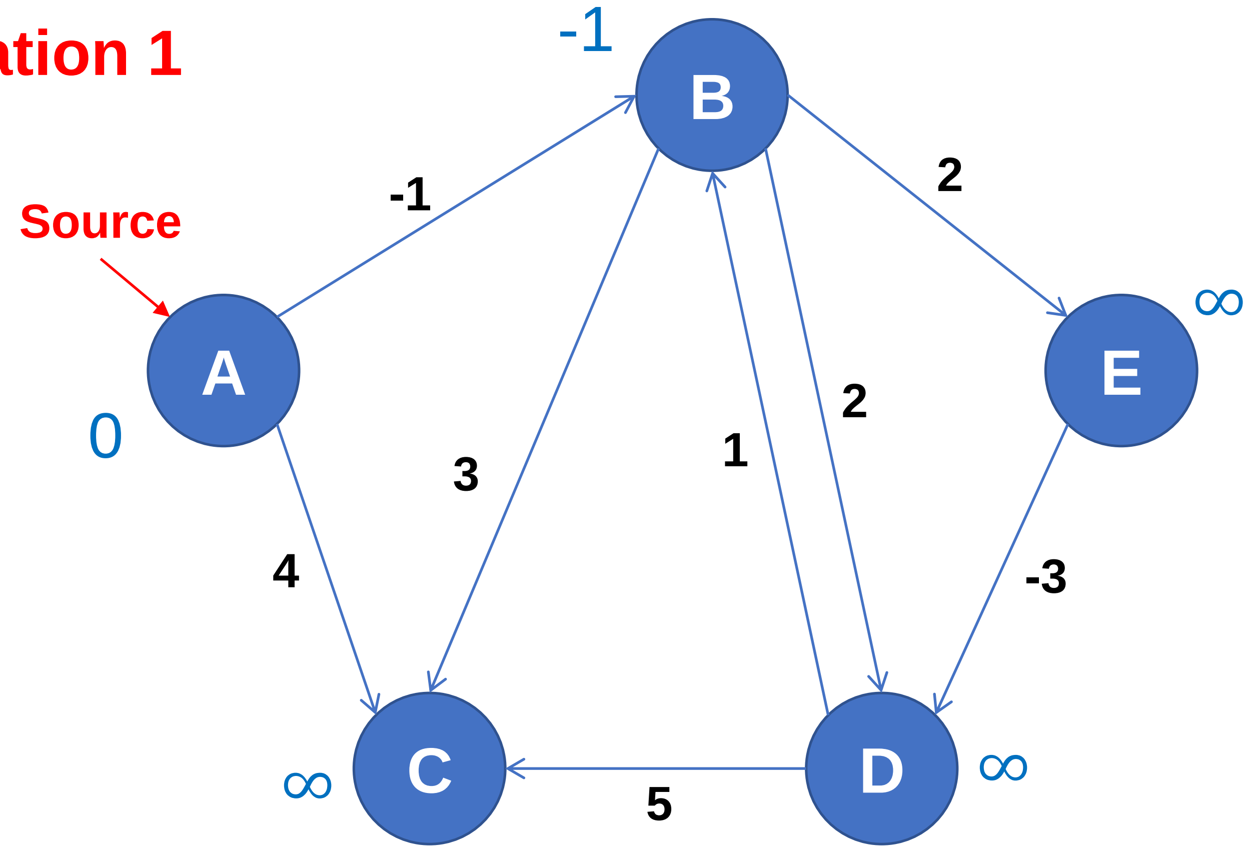
Vertex	d	π
A	0	NIL
B	∞	NIL
C	∞	NIL
D	∞	NIL
E	∞	NIL

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

(A, B): d[u]+edge(u,v)=0+(-1) < ∞

Bellman Ford's algorithm

Iteration 1



Vertex	d	π
A	0	NIL
B	-1	A
C	∞	NIL
D	∞	NIL
E	∞	NIL

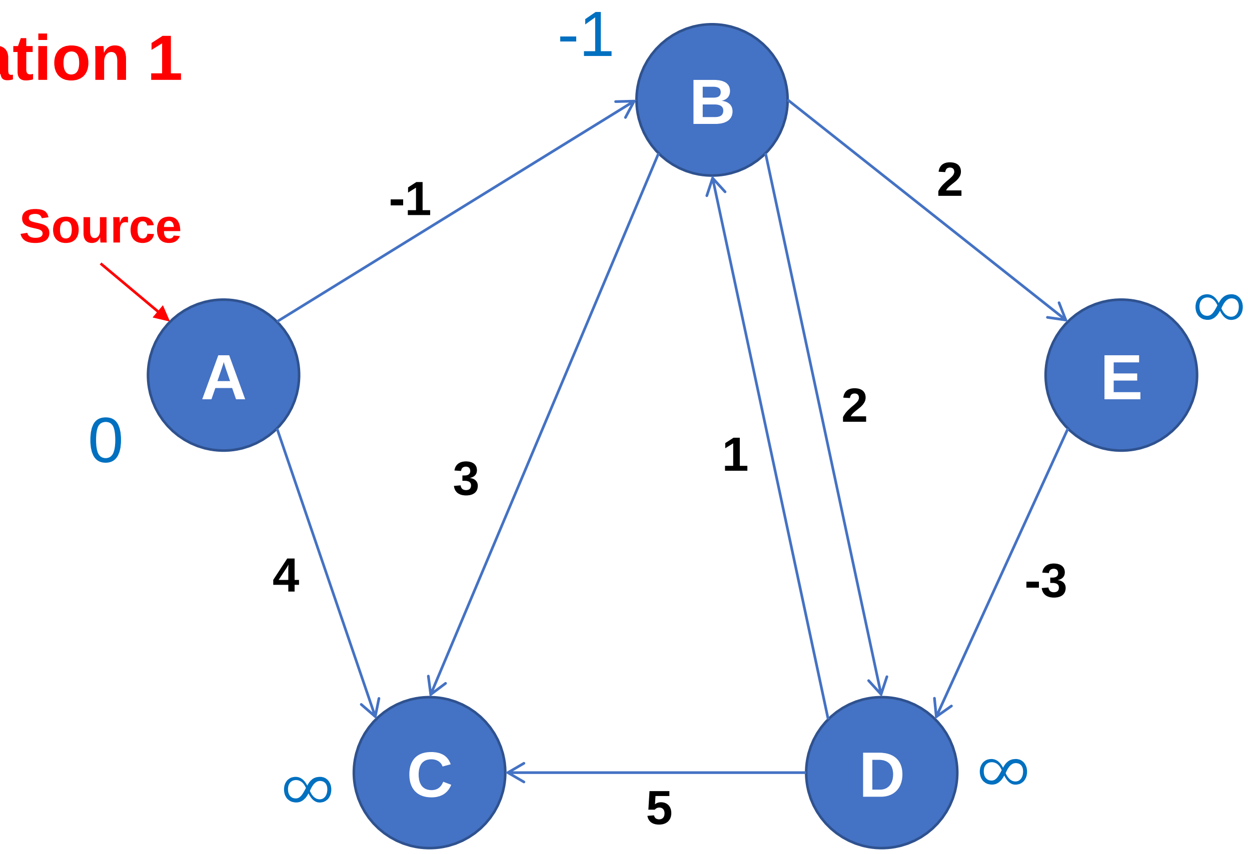
distance ↑ d; parent ↑ π

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

(A, B): d[u]+edge(u,v)=0+(-1) < ∞

Bellman Ford's algorithm

Iteration 1



Vertex	d	π
A	0	NIL
B	-1	A
C	∞	NIL
D	∞	NIL
E	∞	NIL

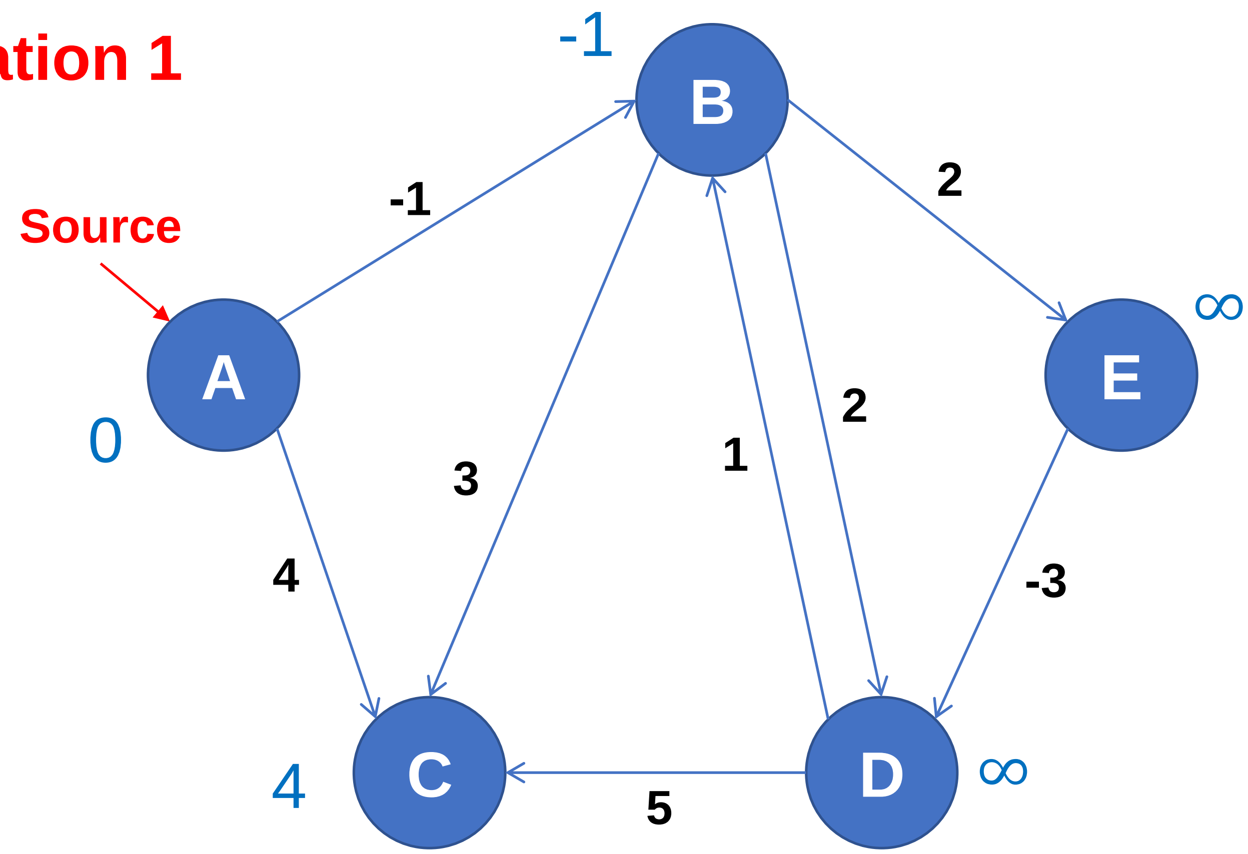
distance: d; parent: π

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

(A, C): d[u]+edge(u,v)=0+4 < ∞

Bellman Ford's algorithm

Iteration 1



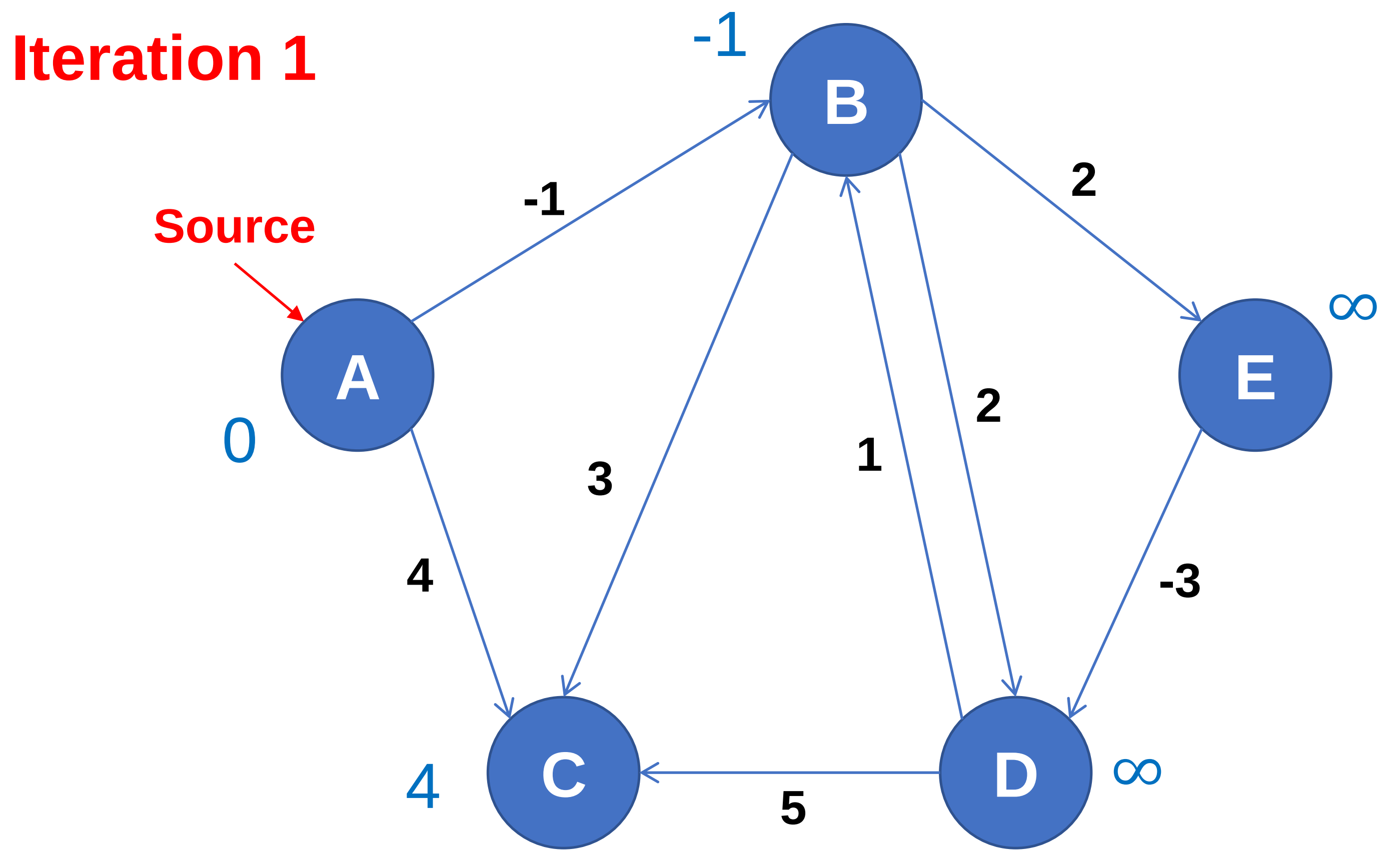
distance ↑ d; parent ↑ π

Vertex	d	π
A	0	NIL
B	-1	A
C	4	A
D	∞	NIL
E	∞	NIL

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

(A, C): d[u]+edge(u,v)=0+4 < ∞

Bellman Ford's algorithm



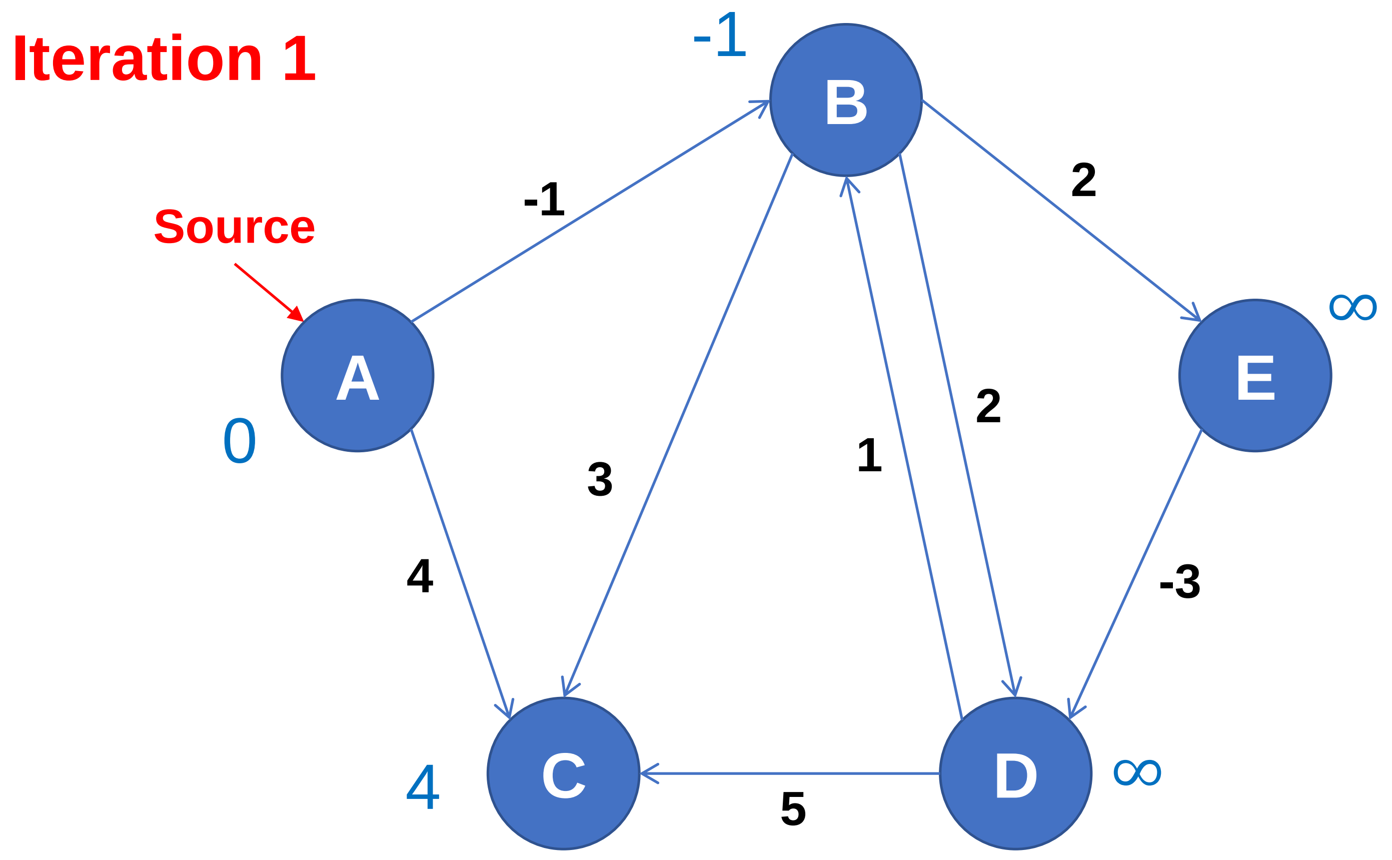
distance ↑ d, parent ↑ π

Vertex	d	π
A	0	NIL
B	-1	A
C	4	A
D	∞	NIL
E	∞	NIL

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

(D, C): $d[u]+edge(u,v)=\infty > 4$

Bellman Ford's algorithm

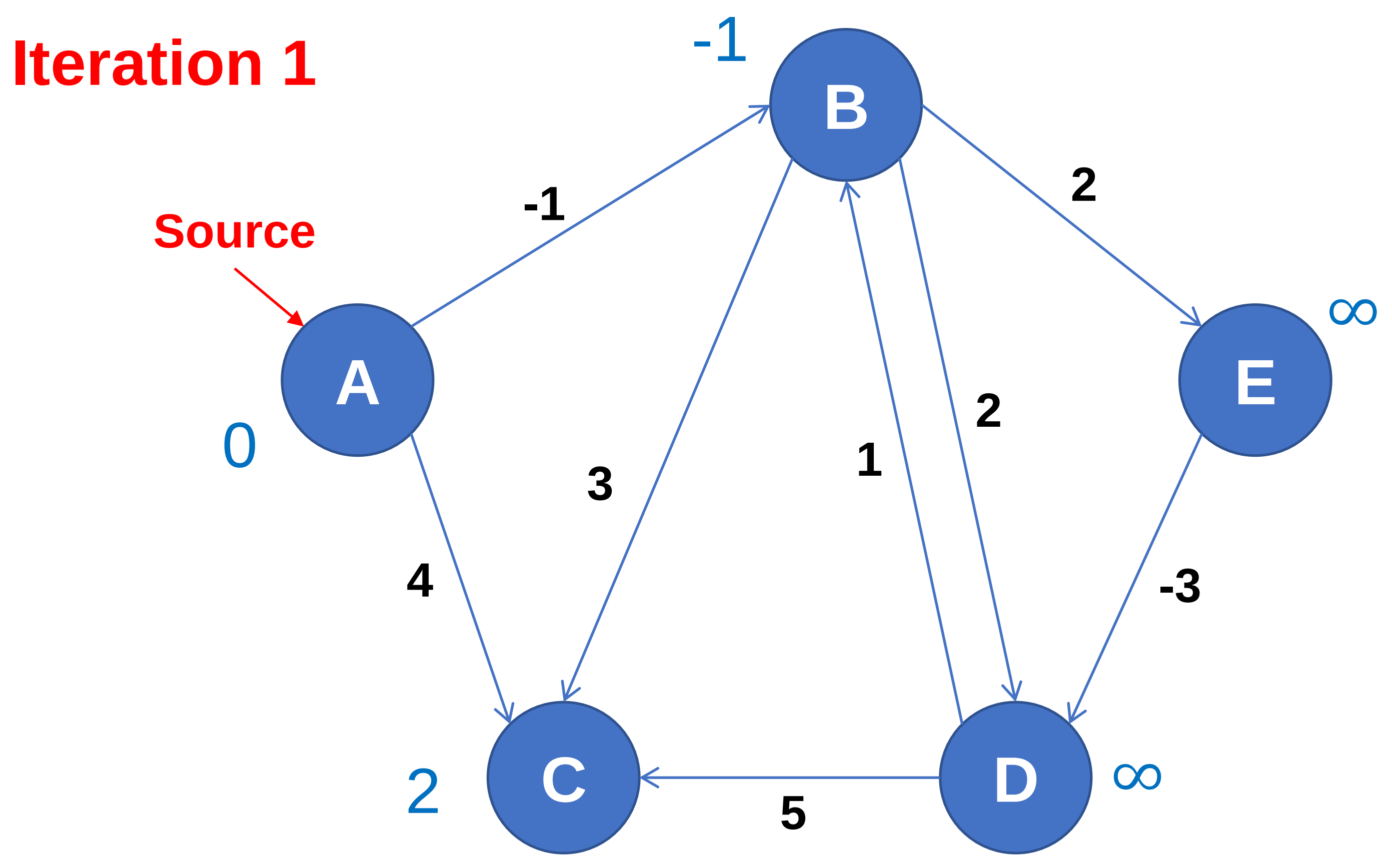


Vertex	d (distance)	π (parent)
A	0	NIL
B	-1	A
C	4	A
D	∞	NIL
E	∞	NIL

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

(B, C): d[u]+edge(u,v)=(-1)+3 < 4

Bellman Ford's algorithm



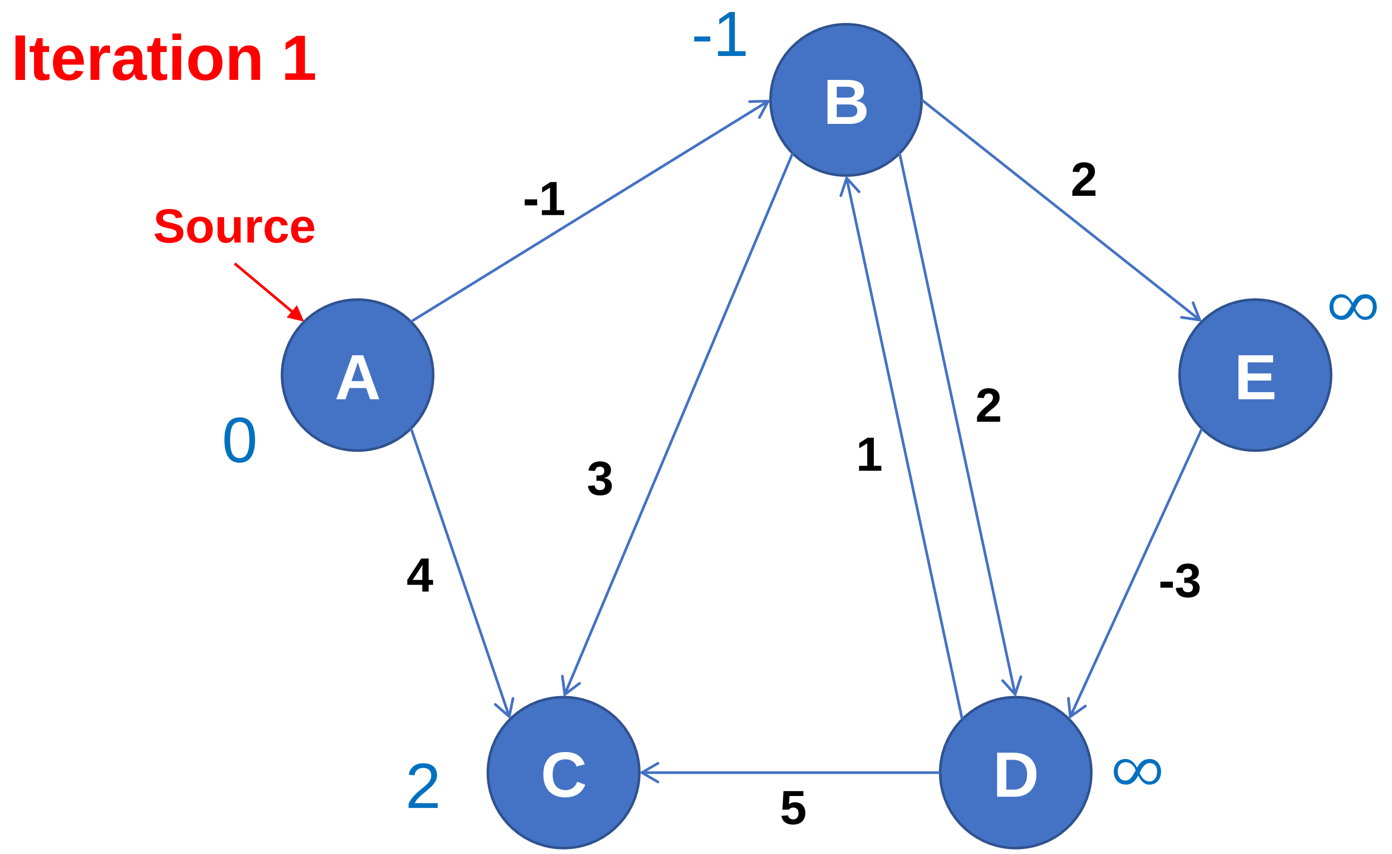
distance ↑ d; parent ↑ π

Vertex	d	π
A	0	NIL
B	-1	A
C	2	B
D	∞	NIL
E	∞	NIL

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

(B, C): d[u]+edge(u,v)=(-1)+3=2 < 4

Bellman Ford's algorithm



distance parent

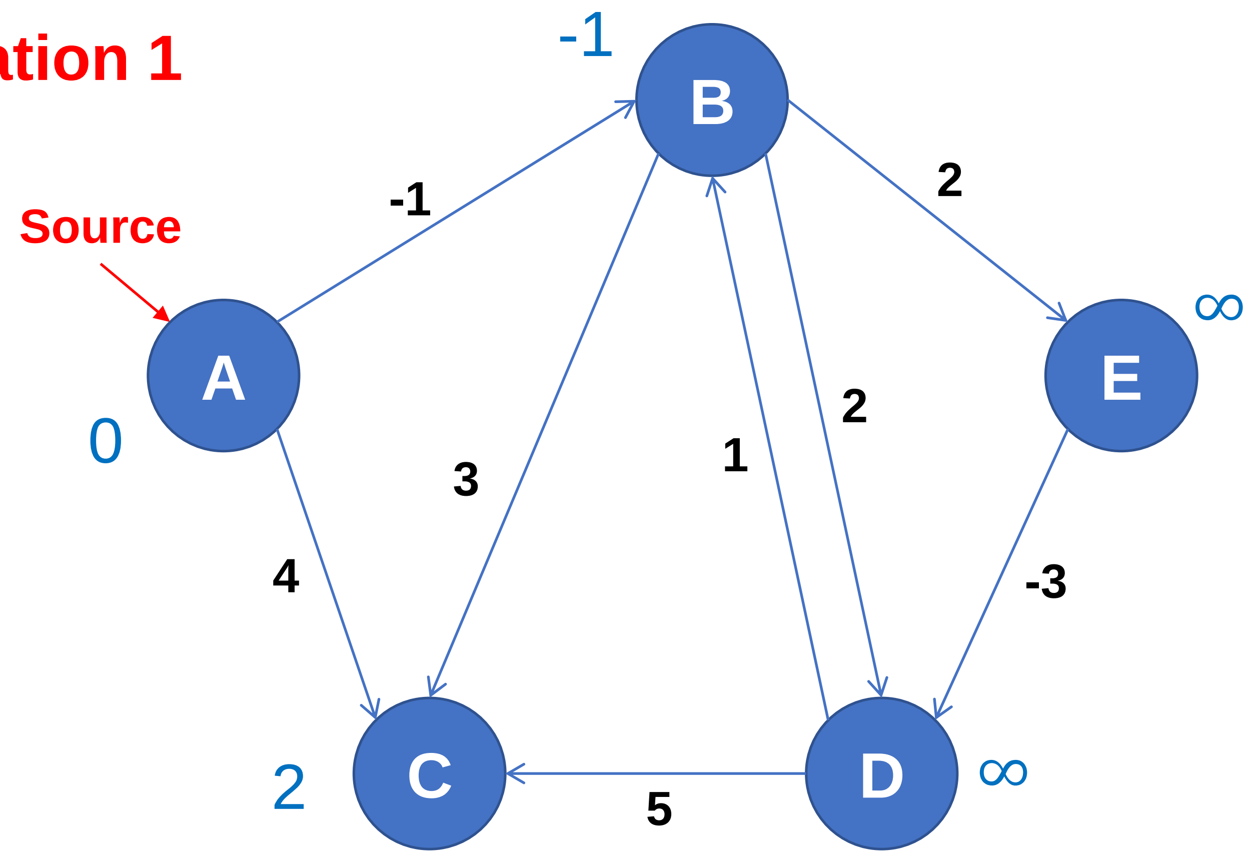
Vertex	d	π
A	0	NIL
B	-1	A
C	2	B
D	∞	NIL
E	∞	NIL

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

(E, D): d[u]+edge(u,v)= ∞ = ∞

Bellman Ford's algorithm

Iteration 1



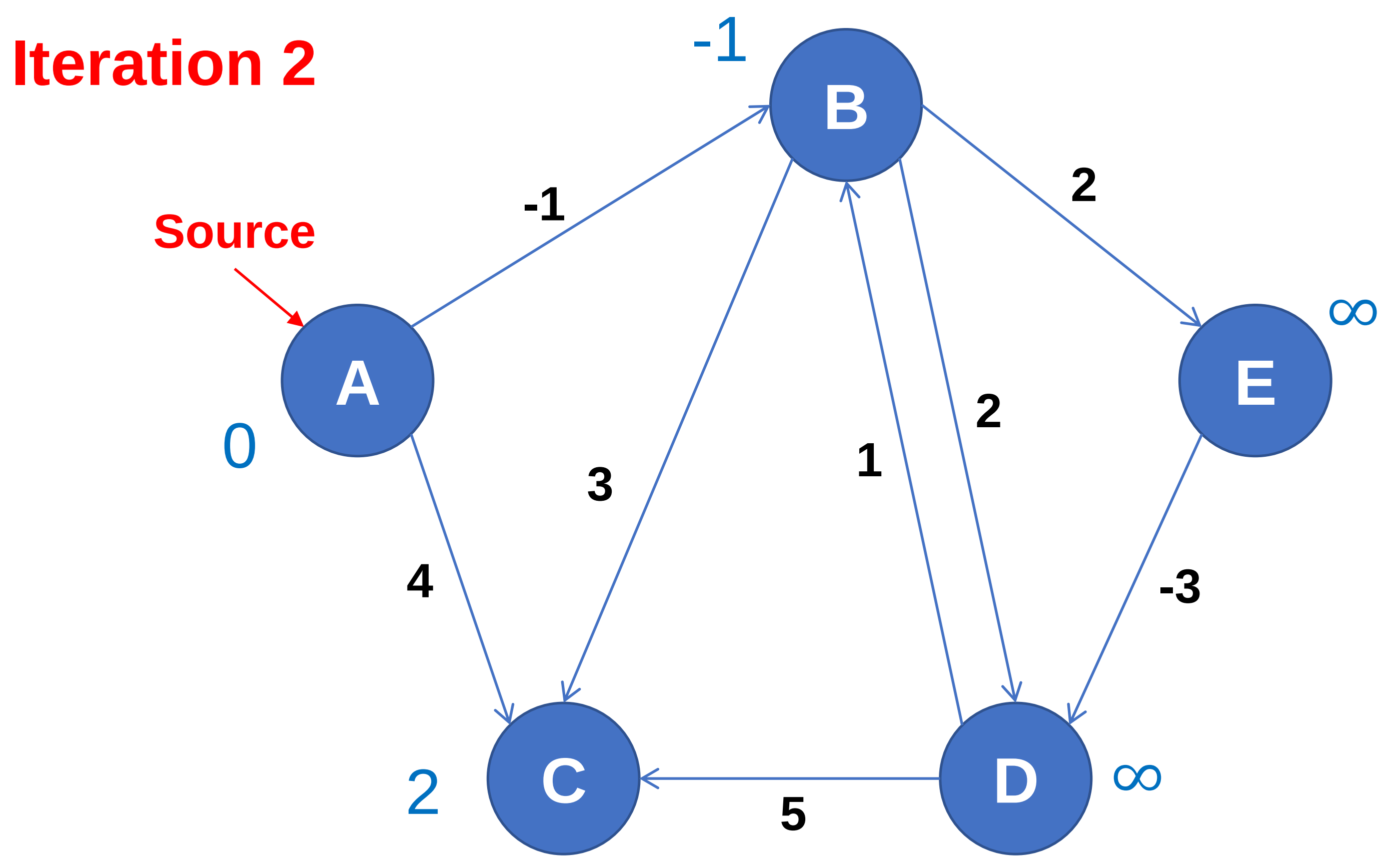
Vertex	d	π
A	0	NIL
B	-1	A
C	2	B
D	∞	NIL
E	∞	NIL

distance ↑ d; parent ↑ π

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

Bellman Ford's algorithm

Iteration 2



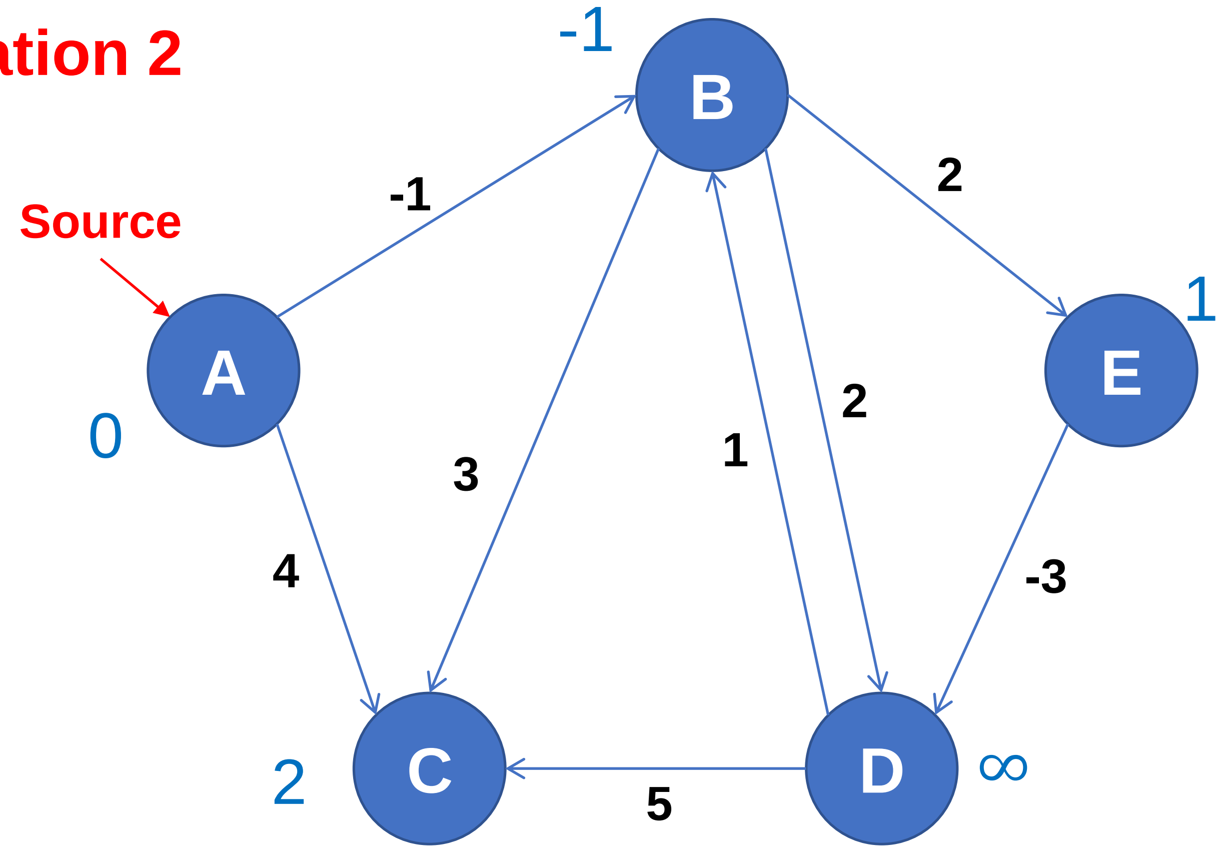
Vertex	d	π
A	0	NIL
B	-1	A
C	2	B
D	∞	NIL
E	∞	NIL

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

(B, E): $d[u]+edge(u,v)=(-1)+2=1 < \infty$

Bellman Ford's algorithm

Iteration 2

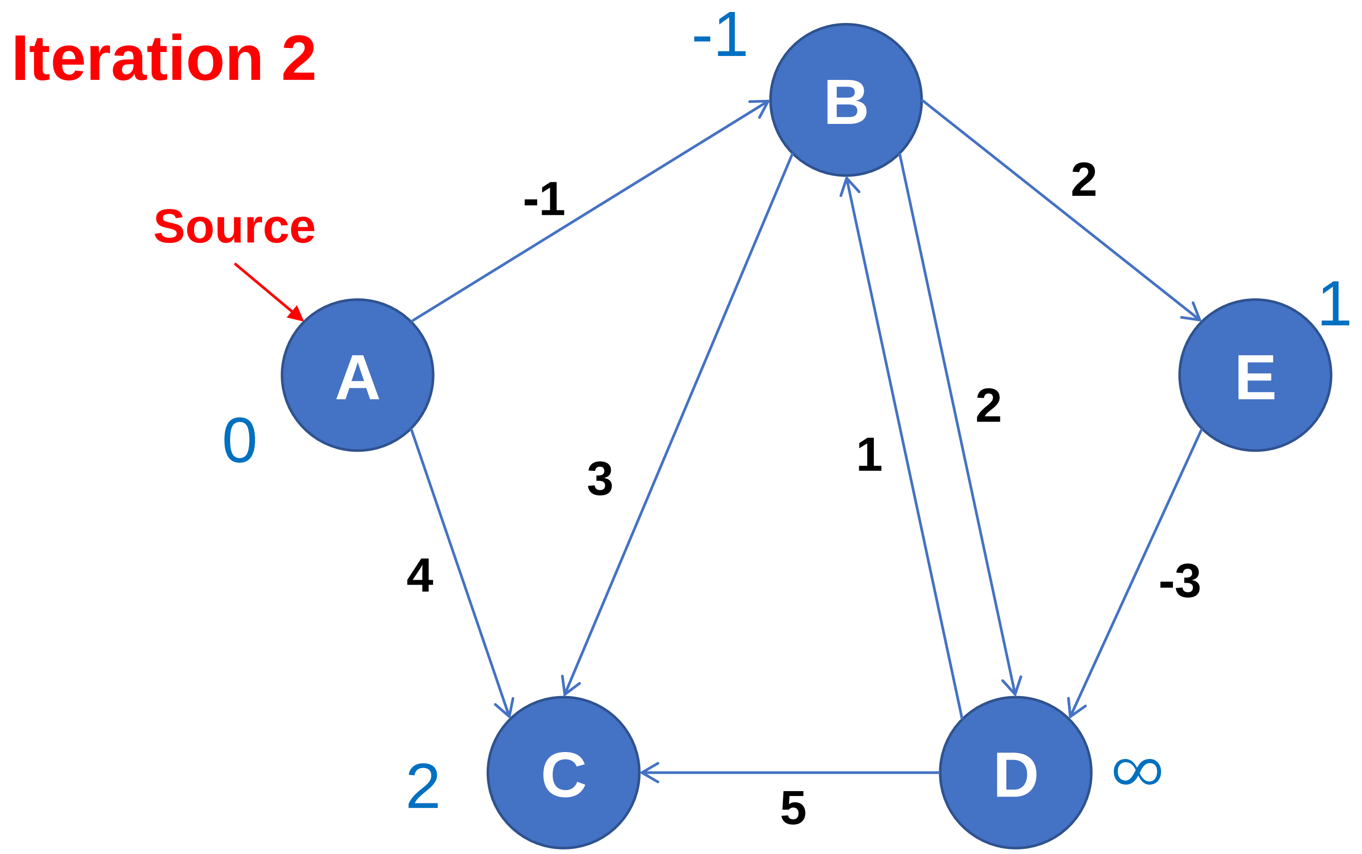


Vertex	d (distance)	π (parent)
A	0	NIL
B	-1	A
C	2	B
D	∞	NIL
E	1	B

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

(B, E): d[u]+edge(u,v)=(-1)+2=1 < ∞

Bellman Ford's algorithm



Vertex	d	π
A	0	NIL
B	-1	A
C	2	B
D	∞	NIL
E	1	B

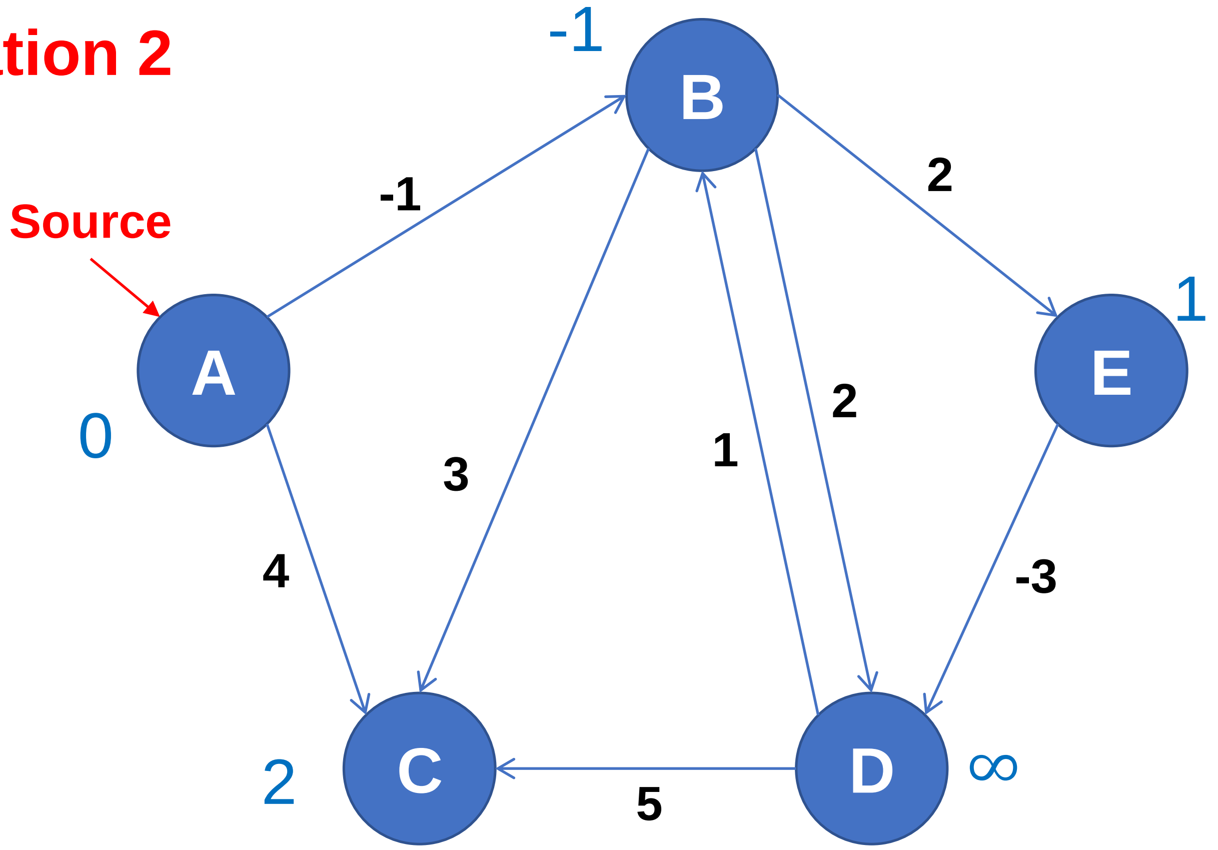
distance (d), parent (π)

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

(D, B): d[u]+edge(u,v)=∞ = ∞

Bellman Ford's algorithm

Iteration 2



Vertex	d	π
A	0	NIL
B	-1	A
C	2	B
D	∞	NIL
E	1	B

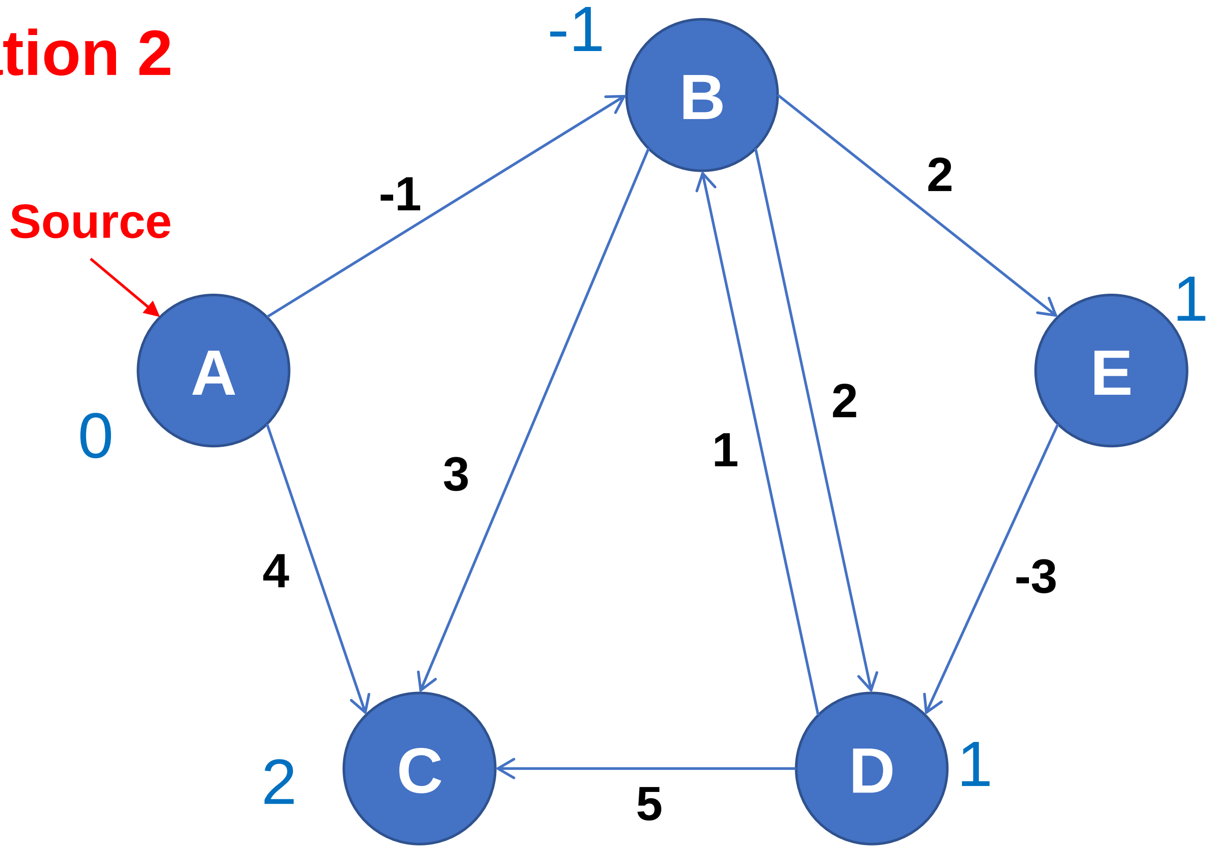
(distance ↑ d, parent ↑ π)

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

(B, D): d[u]+edge(u,v)=(-1)+2=1 < ∞

Bellman Ford's algorithm

Iteration 2



distance parent

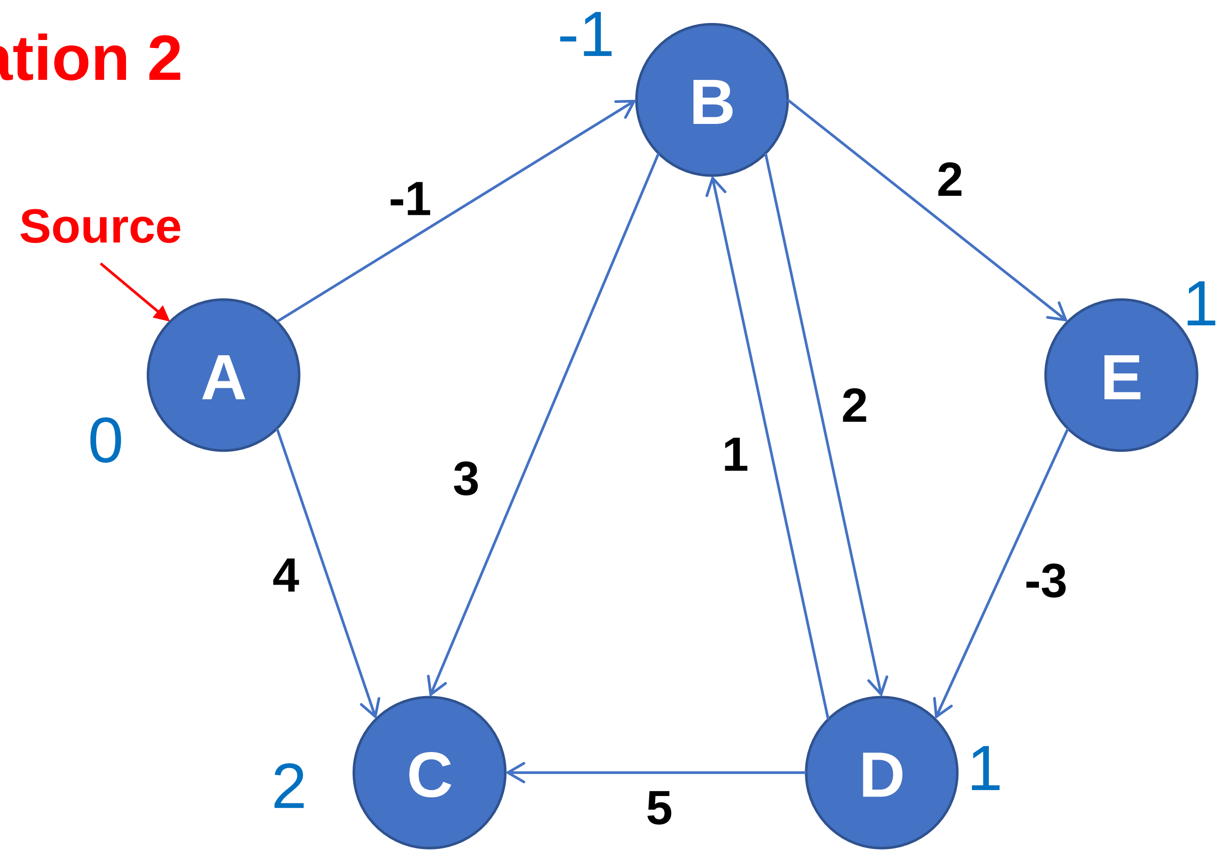
Vertex	d	π
A	0	NIL
B	-1	A
C	2	B
D	1	B
E	1	B

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

(B, D): d[u]+edge(u,v)=(-1)+2=1 < ∞

Bellman Ford's algorithm

Iteration 2



Vertex	d	π
A	0	NIL
B	-1	A
C	2	B
D	1	B
E	1	B

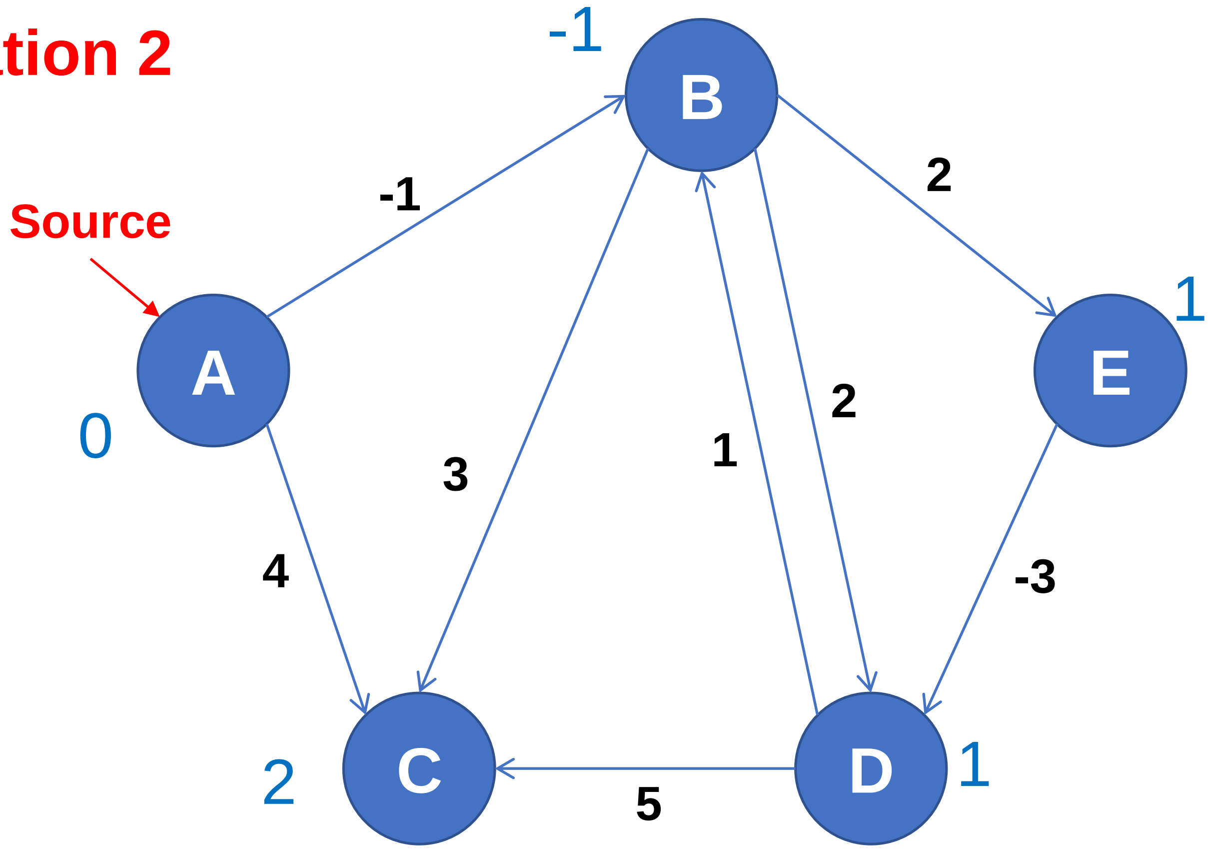
distance → d; parent → π

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

(A, B): d[u]+edge(u,v)=0+(-1)=-1 = -1

Bellman Ford's algorithm

Iteration 2



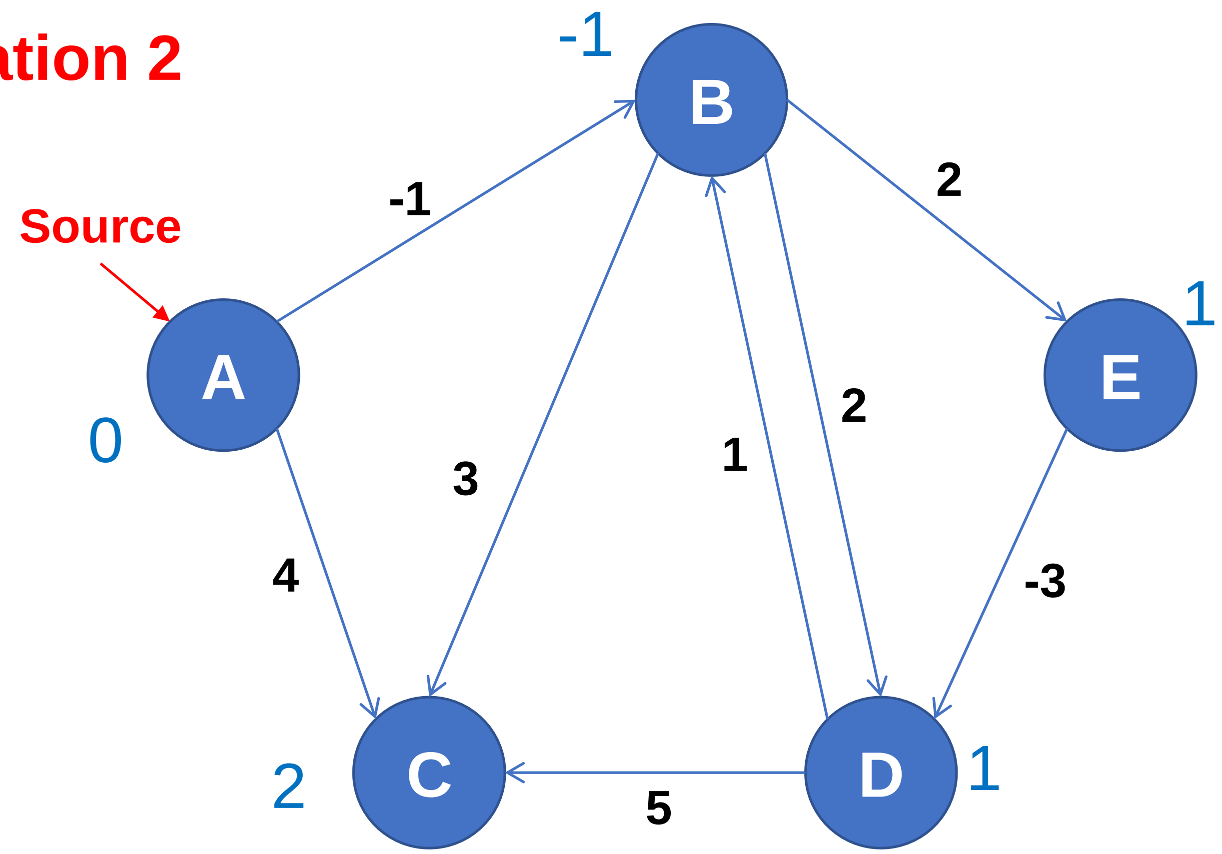
Vertex	d (distance)	π (parent)
A	0	NIL
B	-1	A
C	2	B
D	1	B
E	1	B

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

(A, C): d[u]+edge(u,v)=0+4=4 > 2

Bellman Ford's algorithm

Iteration 2



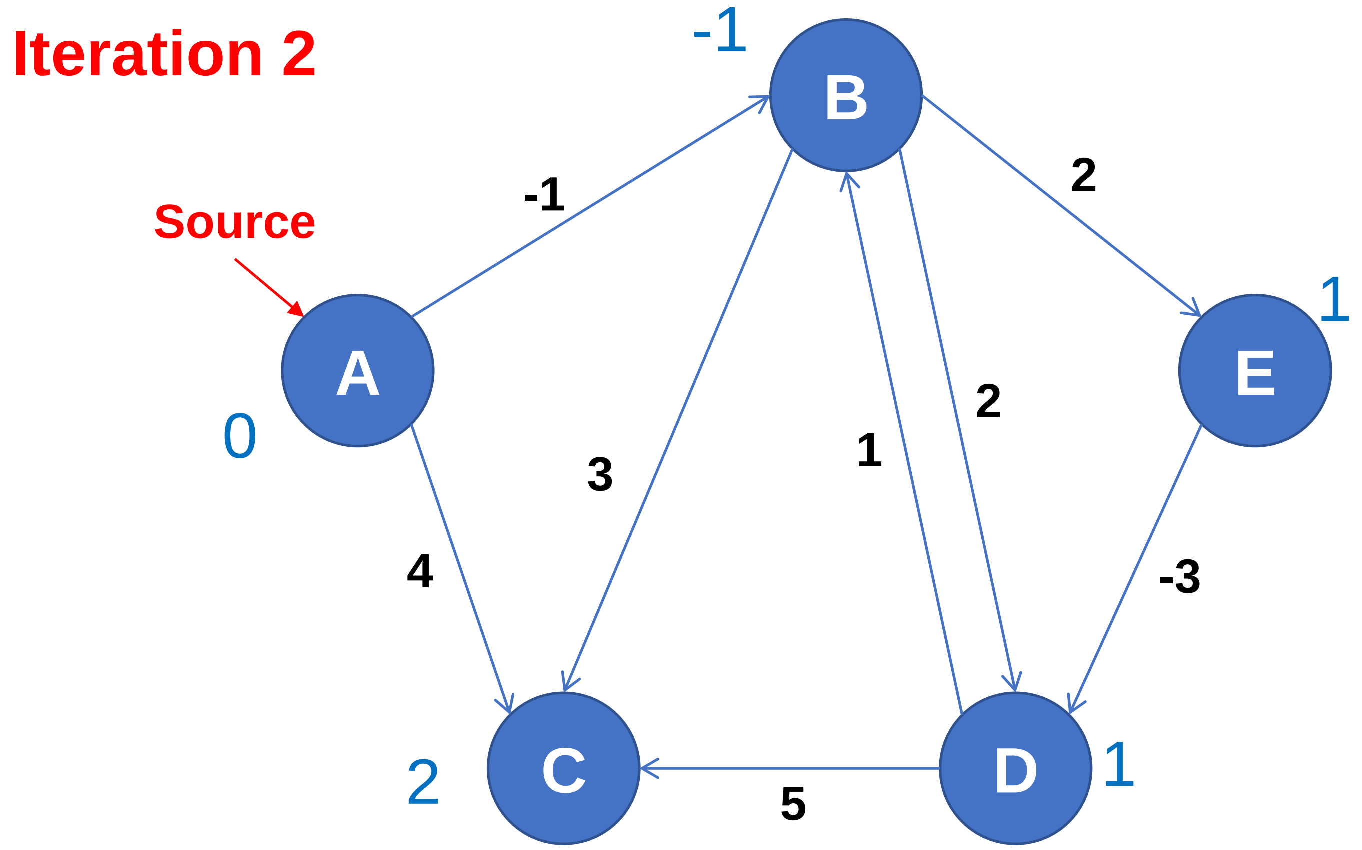
distance parent

Vertex	d	π
A	0	NIL
B	-1	A
C	2	B
D	1	B
E	1	B

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

(D, C): d[u]+edge(u,v)=1+5=6 > 2

Bellman Ford's algorithm



distance parent

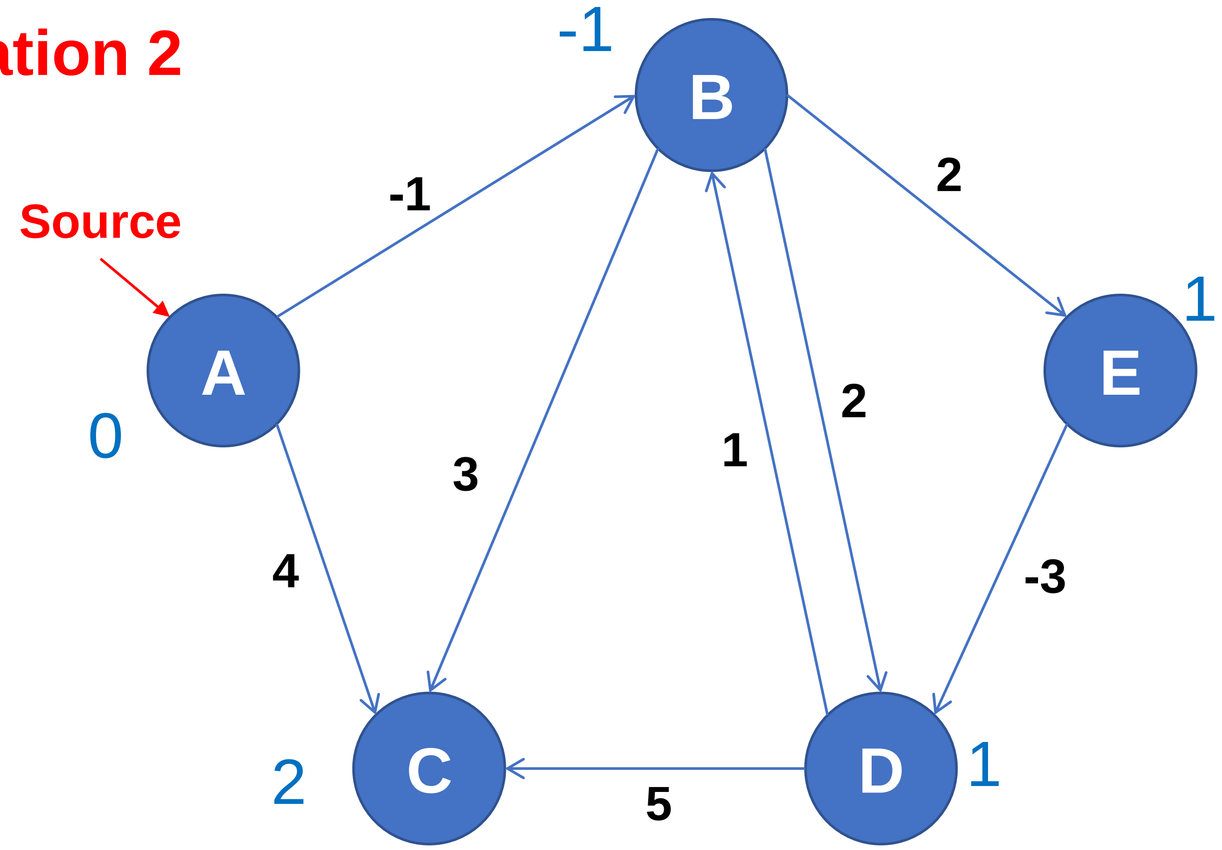
Vertex	d	π
A	0	NIL
B	-1	A
C	2	B
D	1	B
E	1	B

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

(B, C): d[u]+edge(u,v)=(-1)+3=2 > 2

Bellman Ford's algorithm

Iteration 2



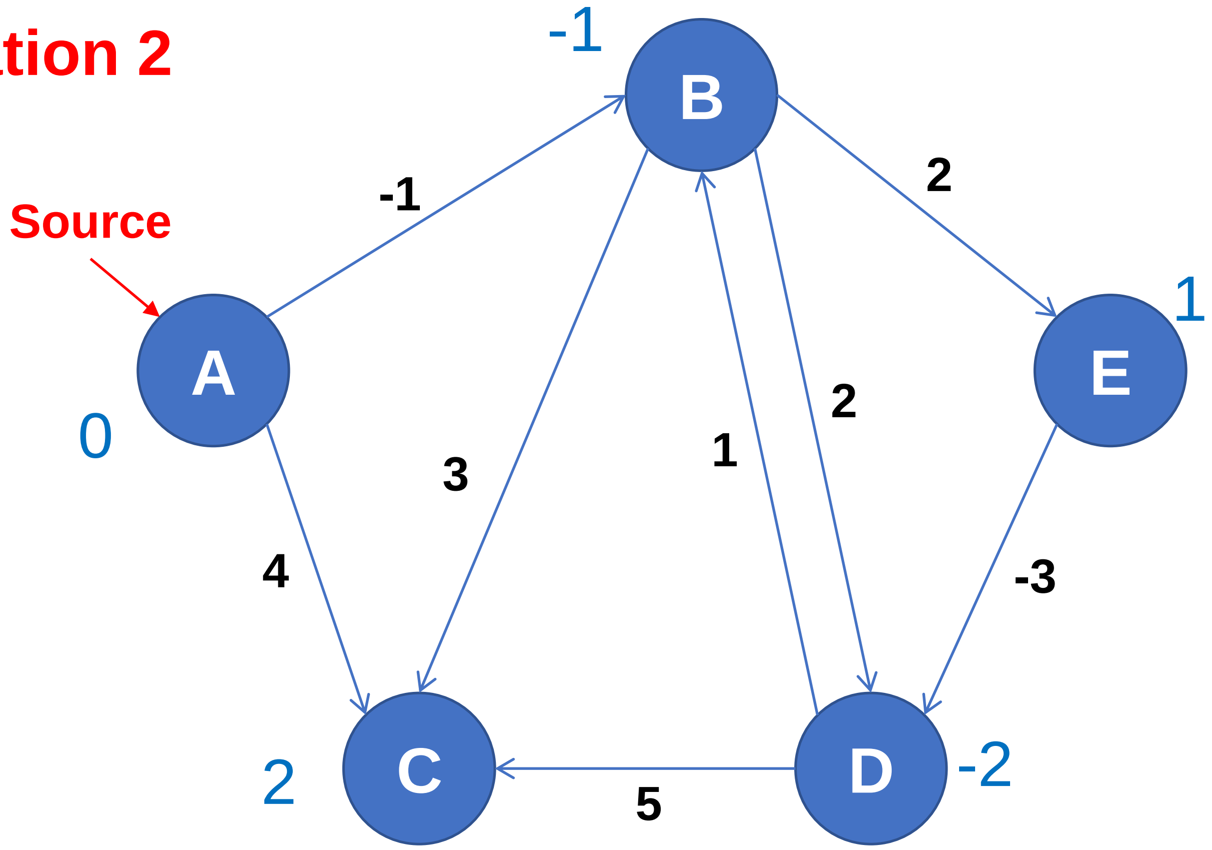
Vertex	d (distance)	π (parent)
A	0	NIL
B	-1	A
C	2	B
D	1	B
E	1	B

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

(E, D): d[u]+edge(u,v)=1+(-3)=-2 < 1

Bellman Ford's algorithm

Iteration 2



distance ↑ d; parent ↑ π

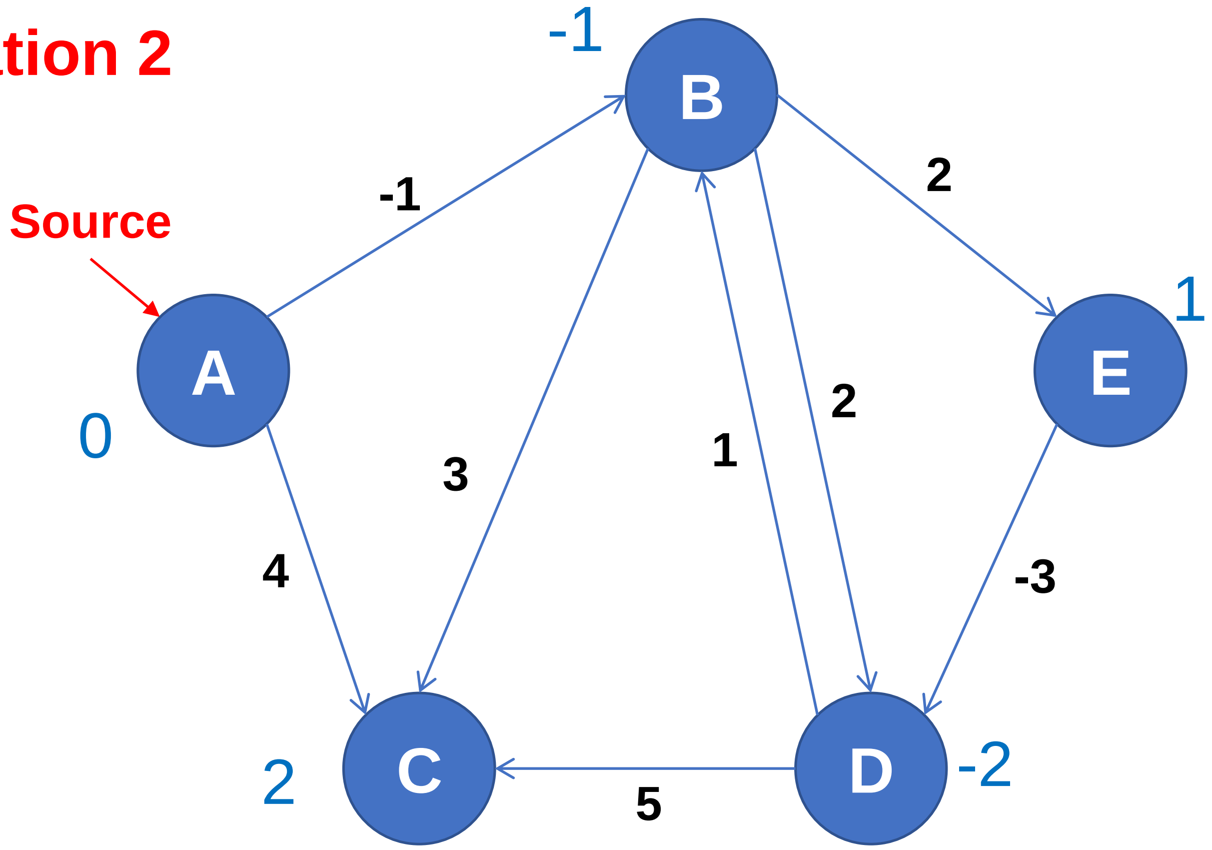
Vertex	d	π
A	0	NIL
B	-1	A
C	2	B
D	-2	E
E	1	B

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

(E, D): d[u]+edge(u,v)=1+(-3)=-2 < 1

Bellman Ford's algorithm

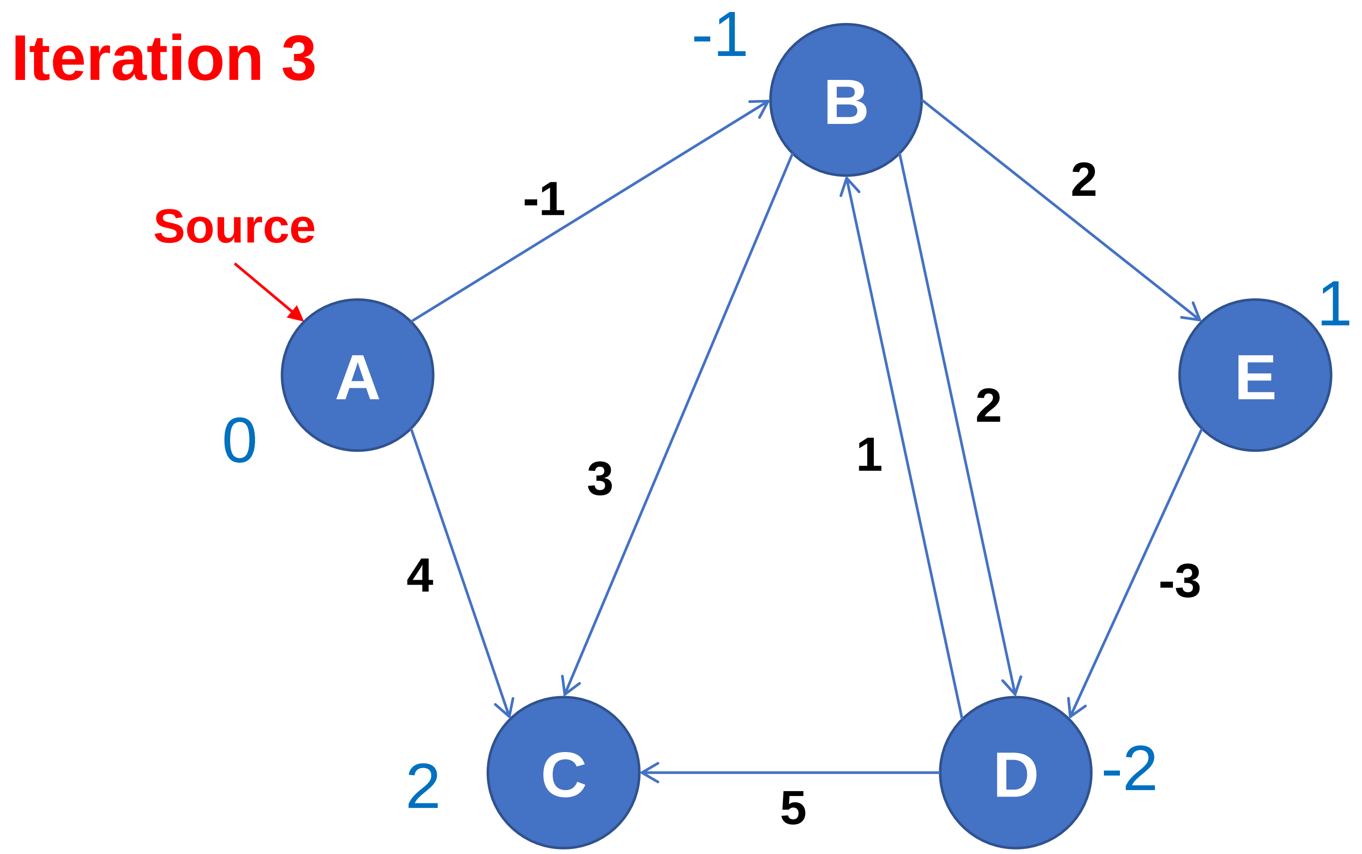
Iteration 2



Vertex	d (distance)	π (parent)
A	0	NIL
B	-1	A
C	2	B
D	-2	E
E	1	B

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

Bellman Ford's algorithm



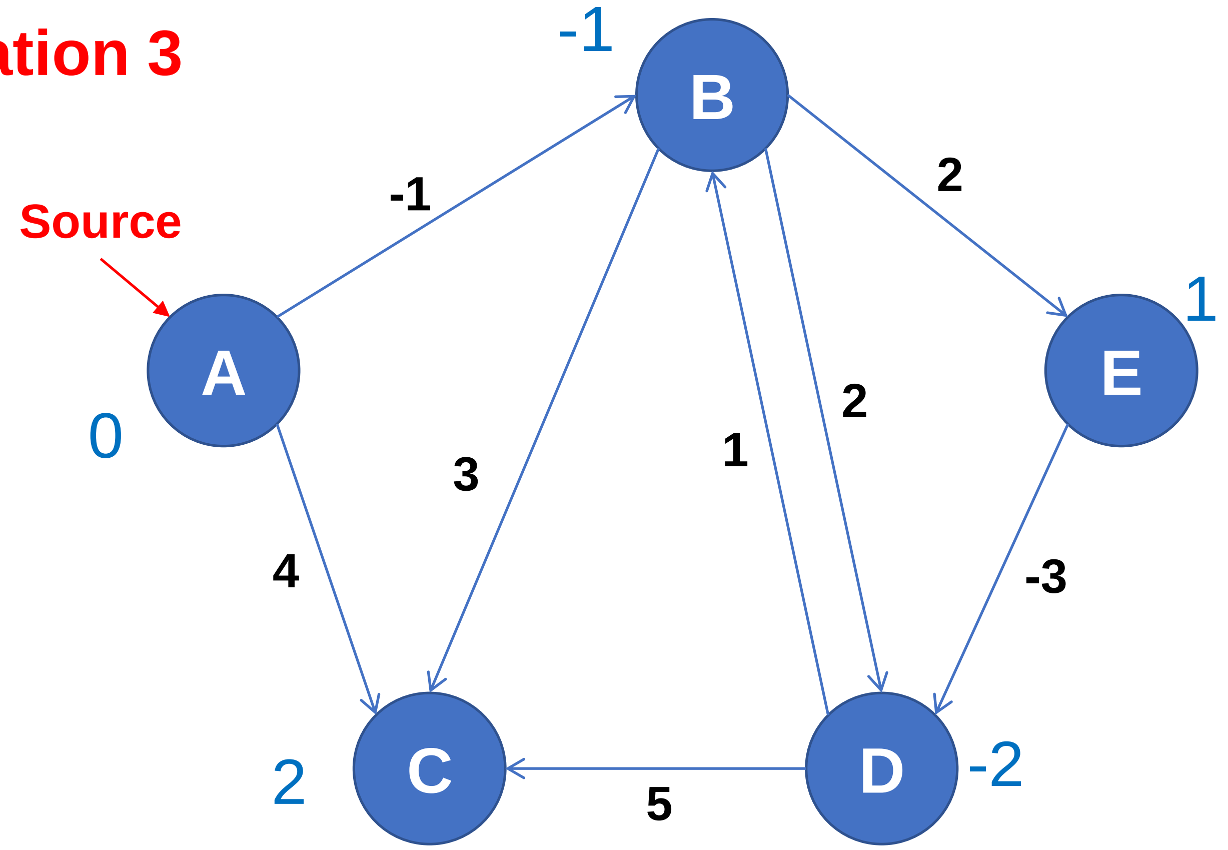
Vertex	d (distance)	π (parent)
A	0	NIL
B	-1	A
C	2	B
D	-2	E
E	1	B

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

(B, E): d[u]+edge(u,v)=(-1)+2=1 = 1

Bellman Ford's algorithm

Iteration 3



distance | parent

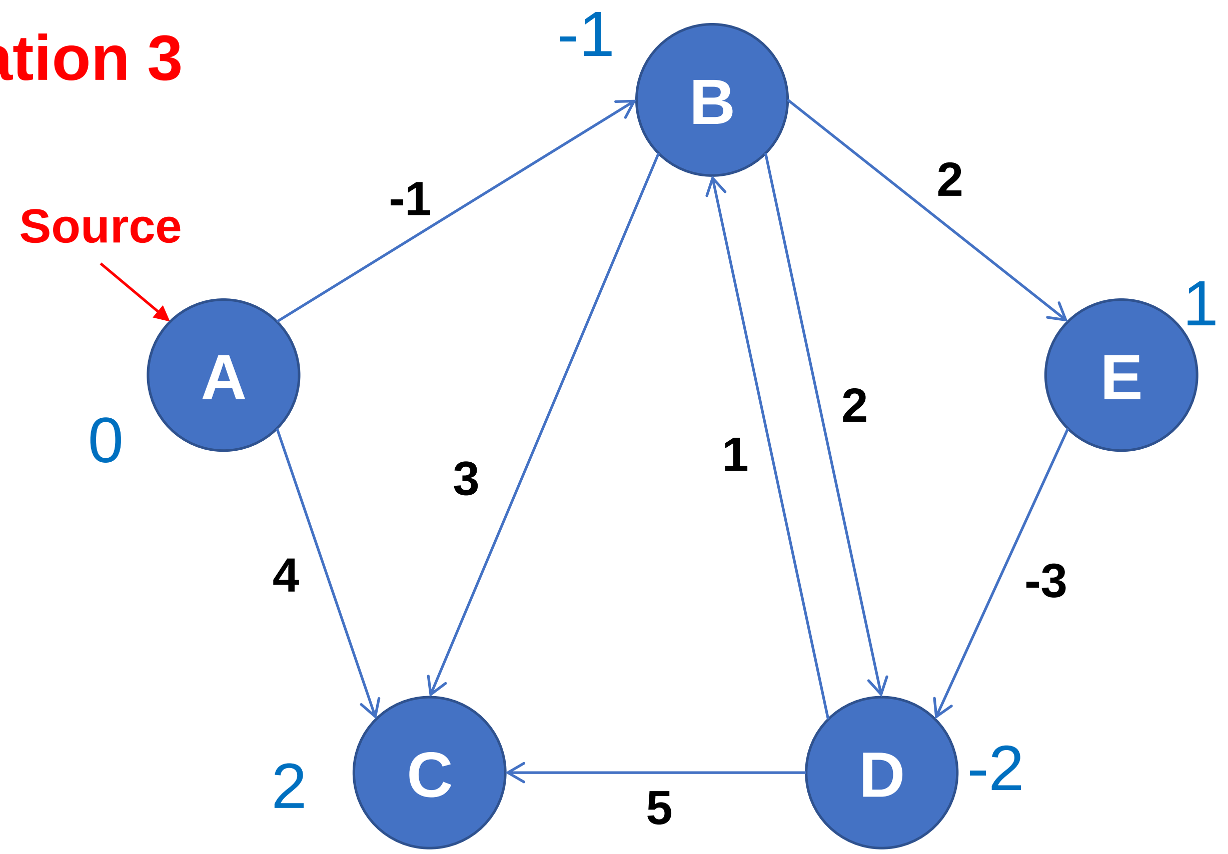
Vertex	d	π
A	0	NIL
B	-1	A
C	2	B
D	-2	E
E	1	B

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

(D, B): d[u]+edge(u,v)=(-2)+1=-1 = -1

Bellman Ford's algorithm

Iteration 3



Vertex	d	π
A	0	NIL
B	-1	A
C	2	B
D	-2	E
E	1	B

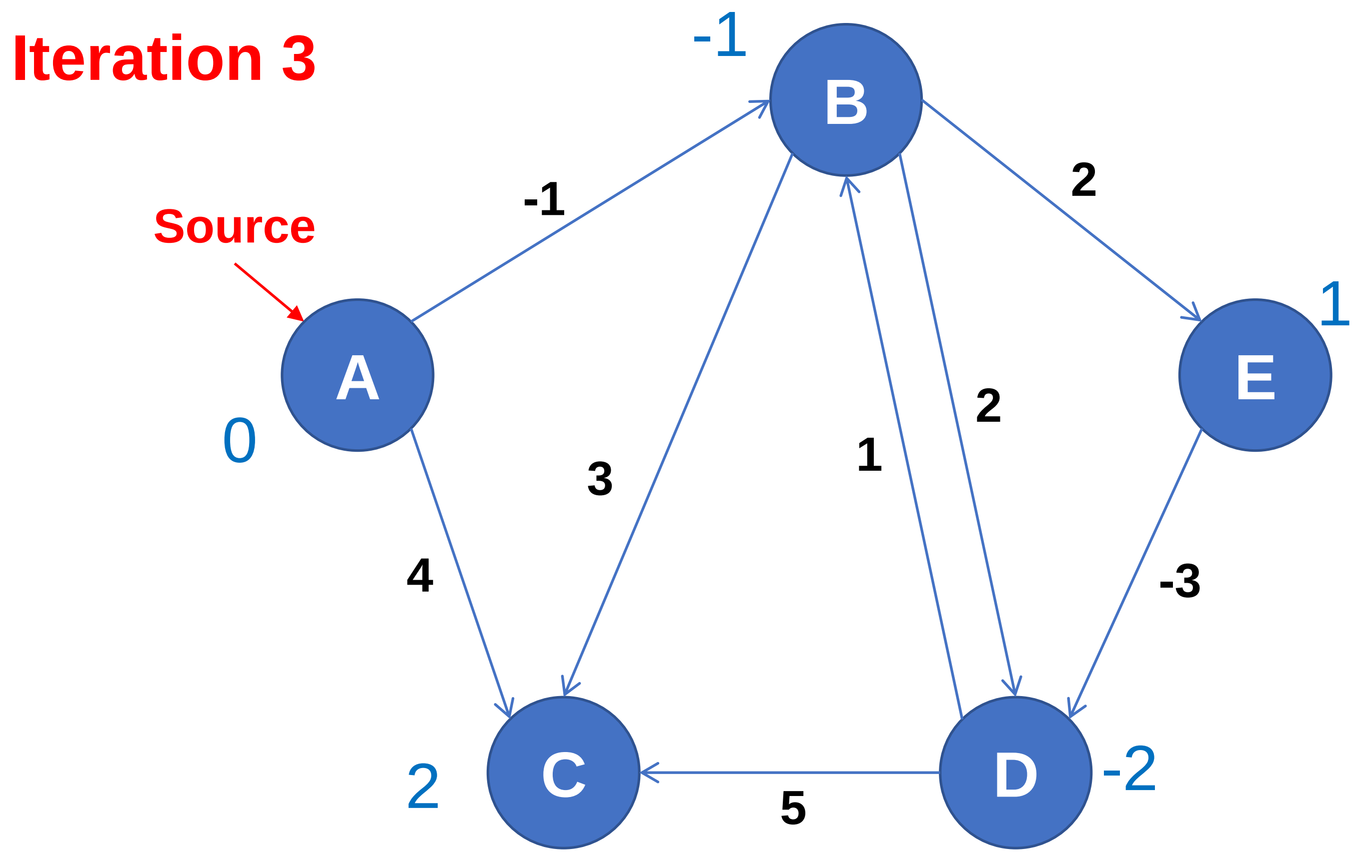
distance (d), parent (π)

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

(B, D): d[u]+edge(u,v)=(-1)+2=1 > -2

Bellman Ford's algorithm

Iteration 3



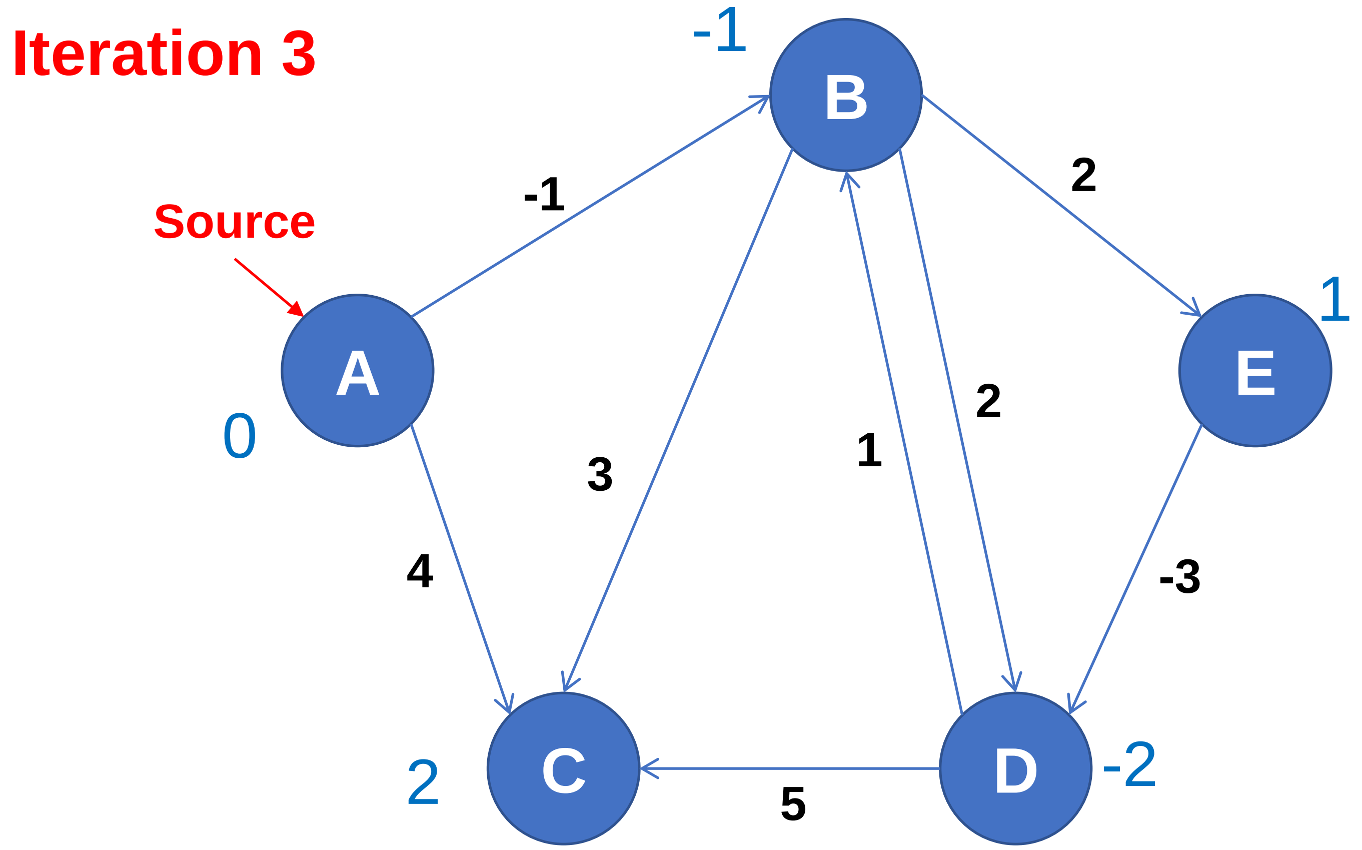
Vertex	d	π
A	0	NIL
B	-1	A
C	2	B
D	-2	E
E	1	B

distance ↑ d, parent ↑ π

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

(A, B): d[u]+edge(u,v)=0+(-1)=-1 = -1

Bellman Ford's algorithm



distance parent

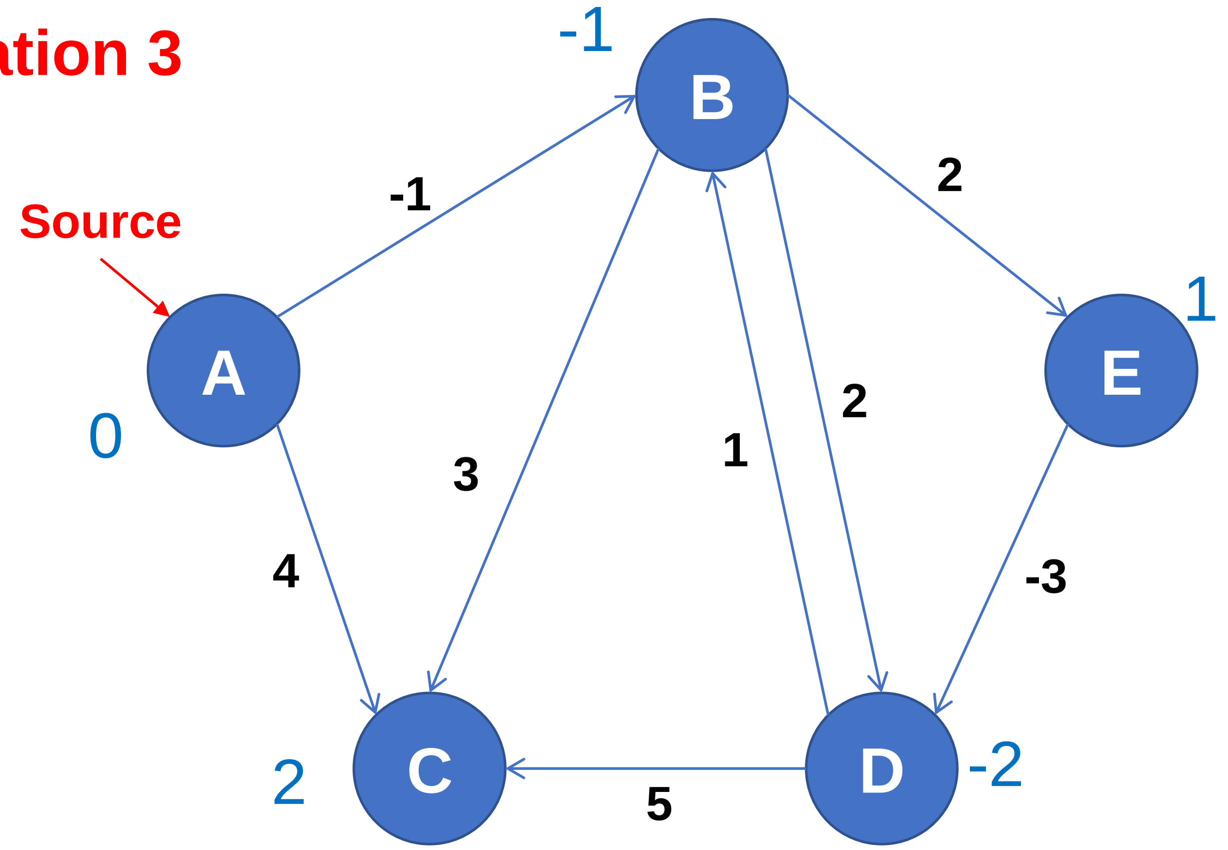
Vertex	d	π
A	0	NIL
B	-1	A
C	2	B
D	-2	E
E	1	B

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

(A, C): d[u]+edge(u,v)=0+2=2 = 2

Bellman Ford's algorithm

Iteration 3



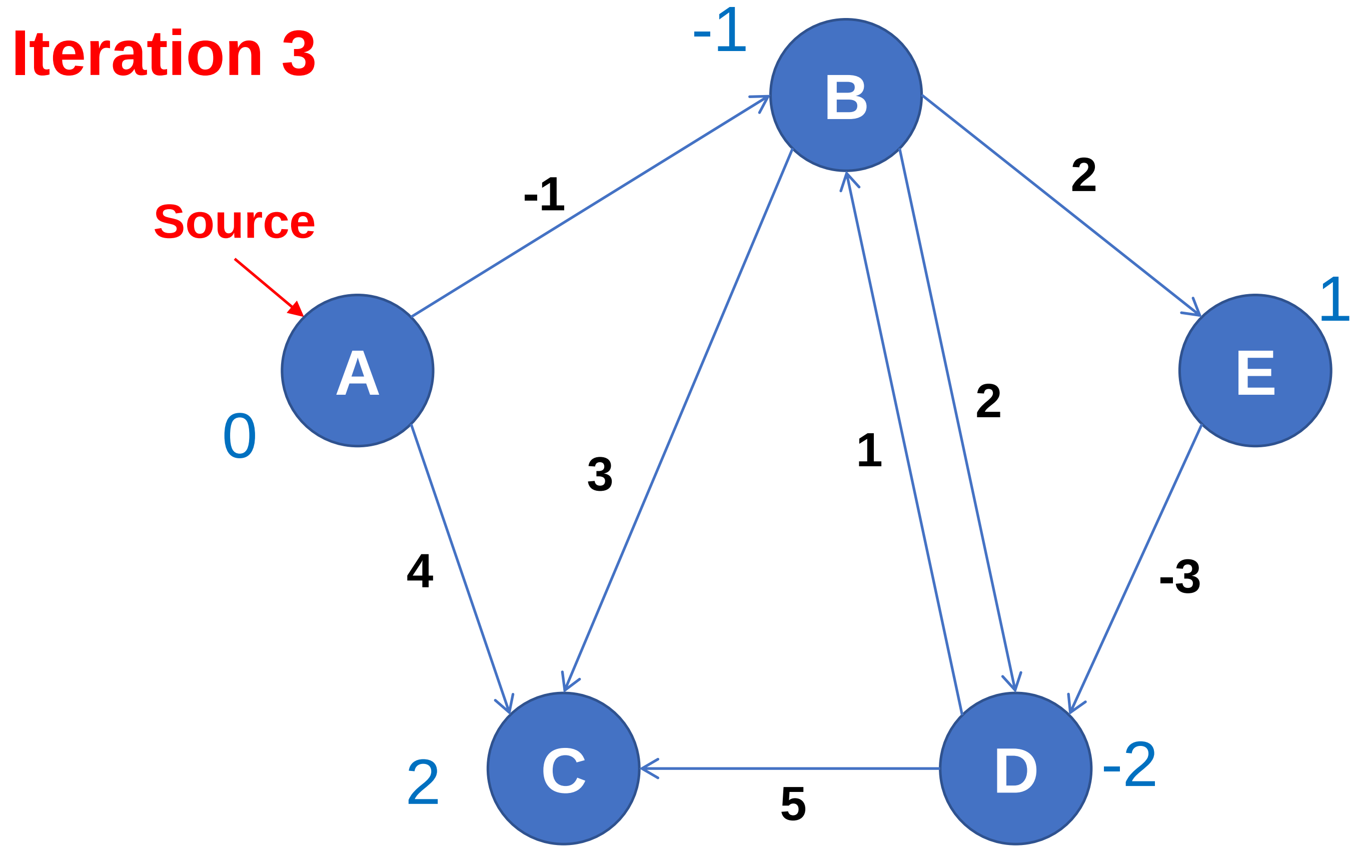
Vertex	d	π
A	0	NIL
B	-1	A
C	2	B
D	-2	E
E	1	B

distance → d; parent → π

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

(D, C): d[u]+edge(u,v)=(-2)+5=3 > 2

Bellman Ford's algorithm



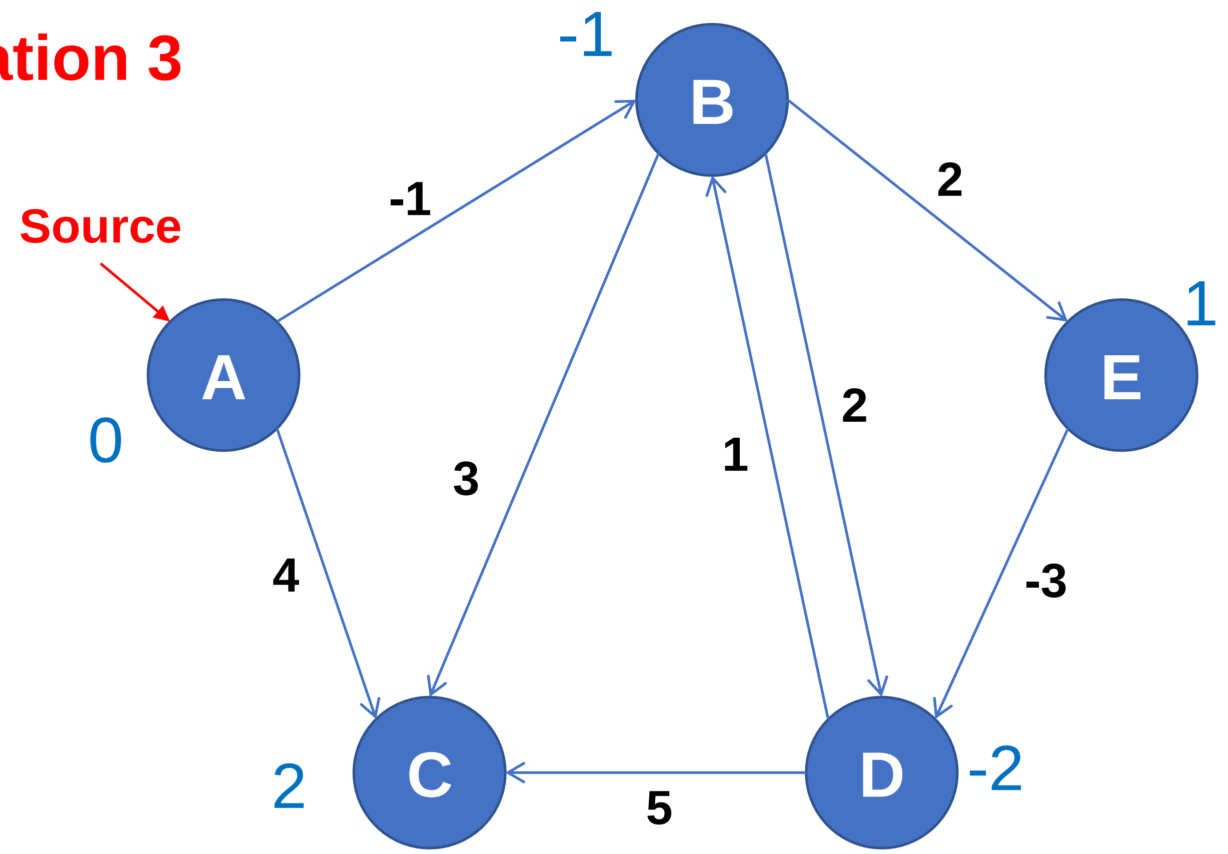
Vertex	d (distance)	π (parent)
A	0	NIL
B	-1	A
C	2	B
D	-2	E
E	1	B

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

(B, C): d[u]+edge(u,v)=(-1)+3=2 = 2

Bellman Ford's algorithm

Iteration 3



distance | parent

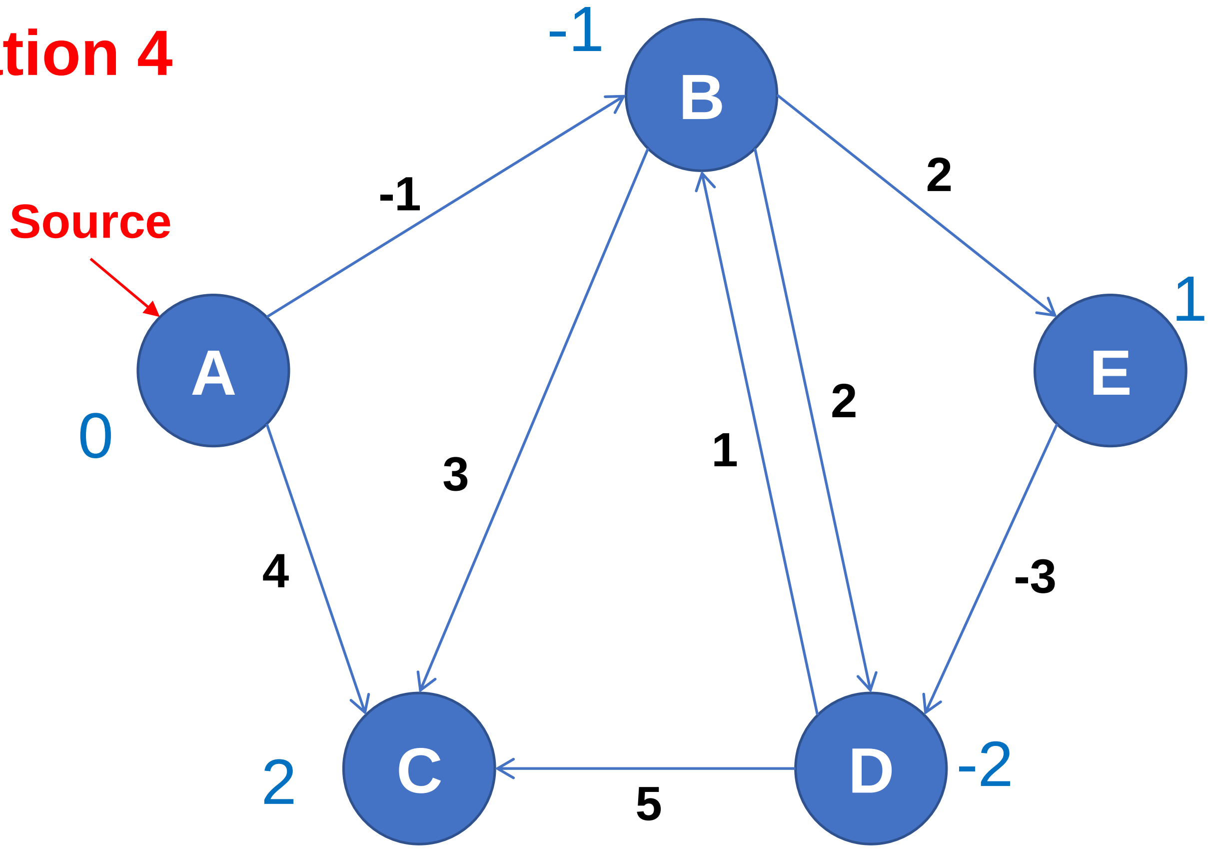
Vertex	d	π
A	0	NIL
B	-1	A
C	2	B
D	-2	E
E	1	B

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

(E, D): d[u]+edge(u,v)=1+(-3)=-2 = -2

Bellman Ford's algorithm

Iteration 4

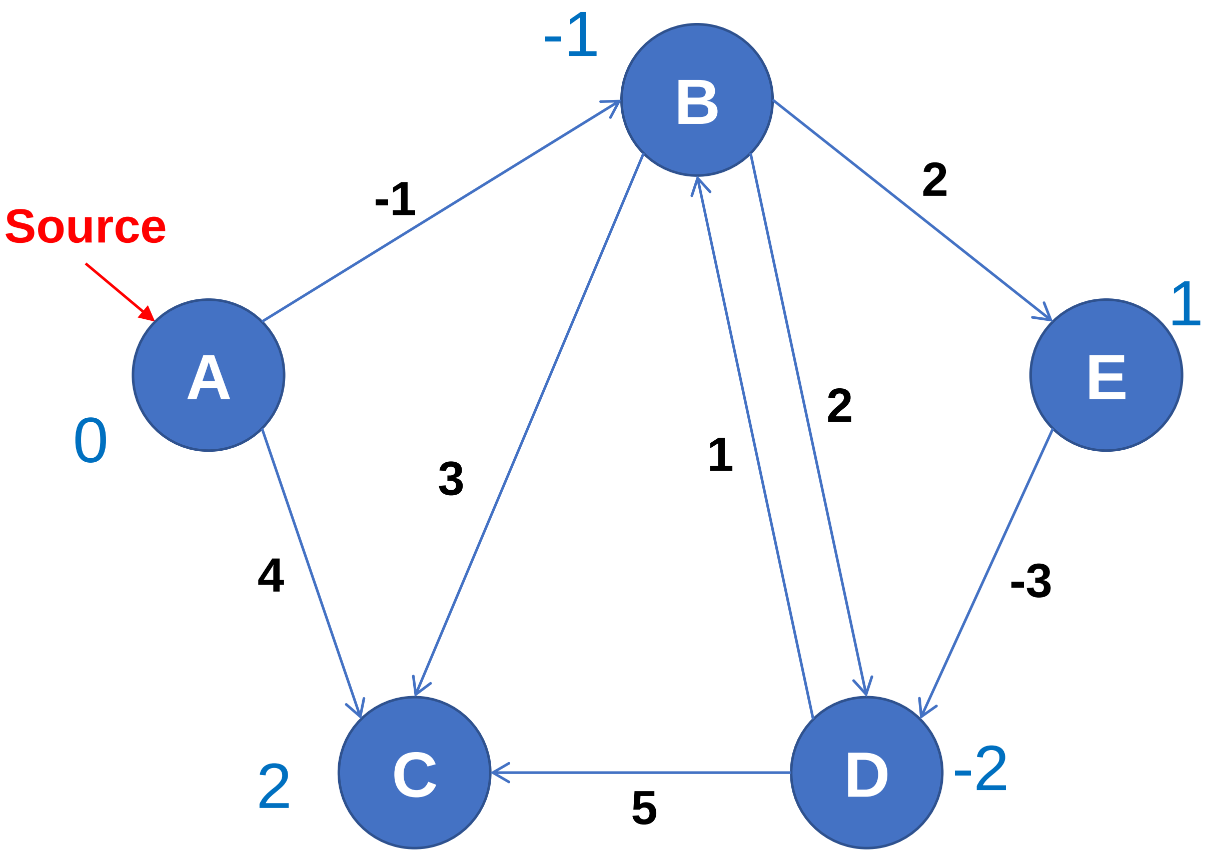


distance parent

Vertex	d	π
A	0	NIL
B	-1	A
C	2	B
D	-2	E
E	1	B

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

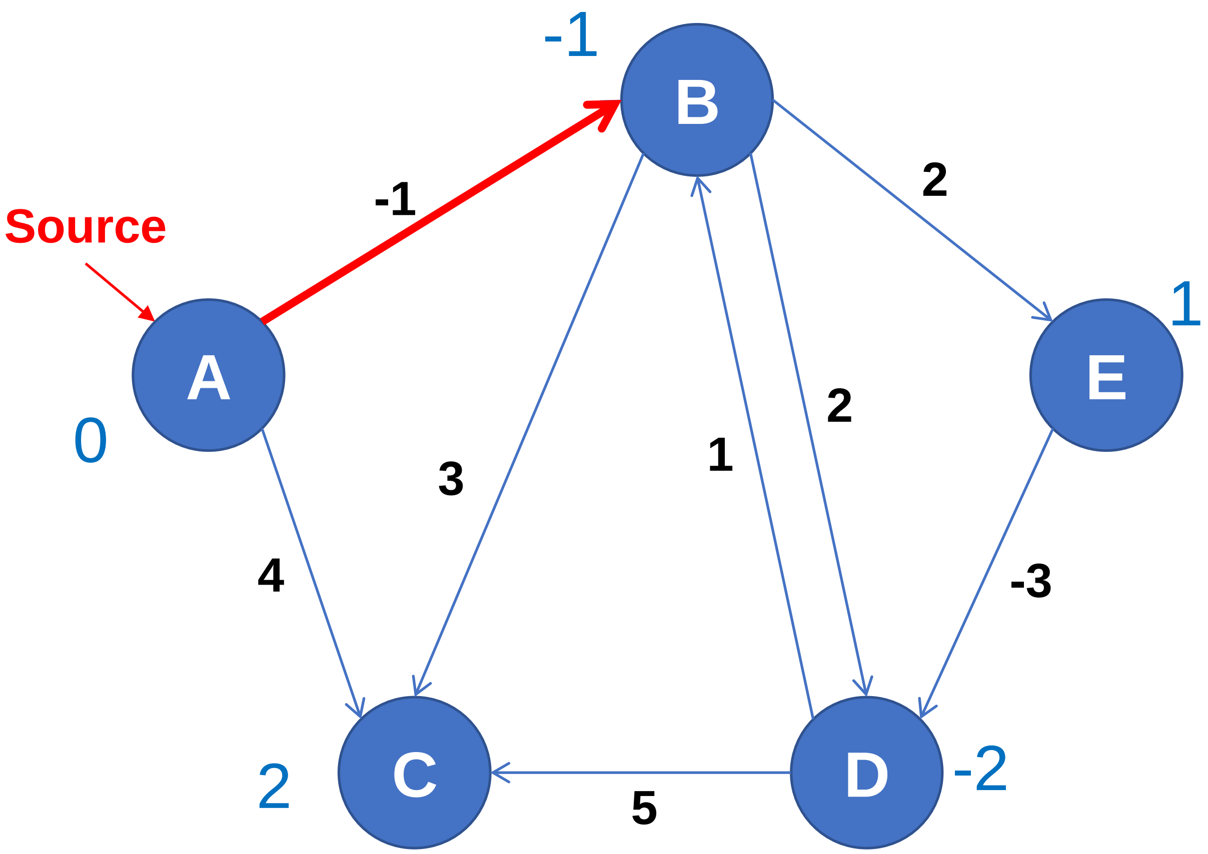
Bellman Ford's algorithm



distance ↑ d, parent ↑ π

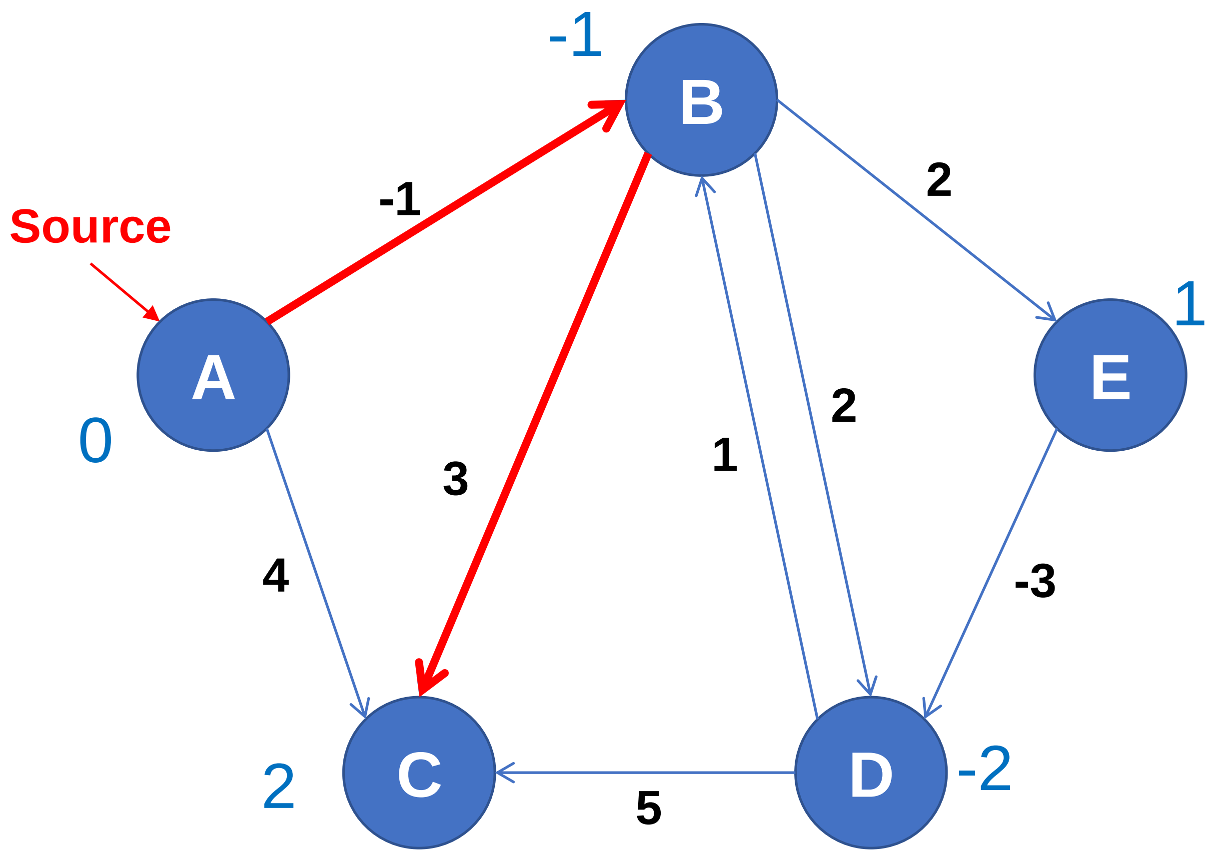
Vertex	d	π
A	0	NIL
B	-1	A
C	2	B
D	-2	E
E	1	B

Bellman Ford's algorithm



	distance	parent
Vertex	d	π
A	0	NIL
B	-1	A
C	2	B
D	-2	E
E	1	B

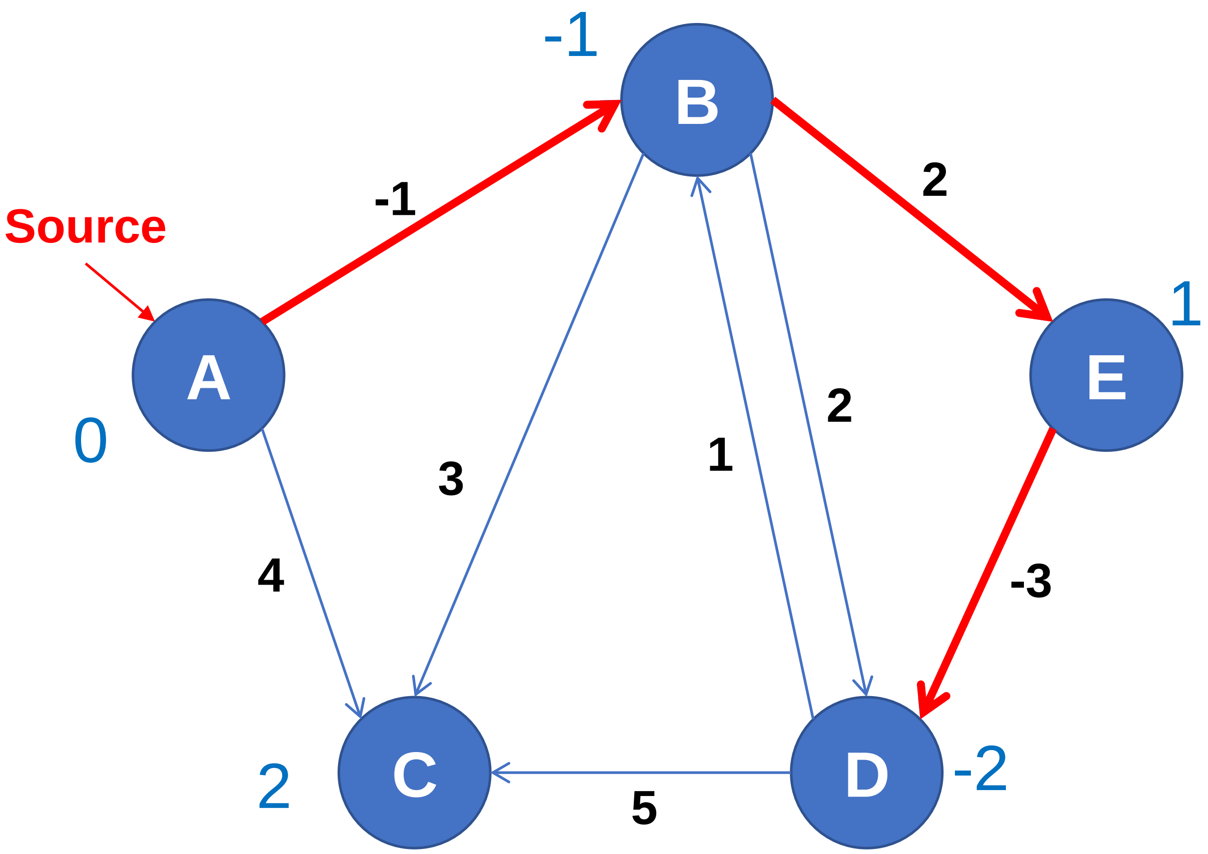
Bellman Ford's algorithm



distance parent

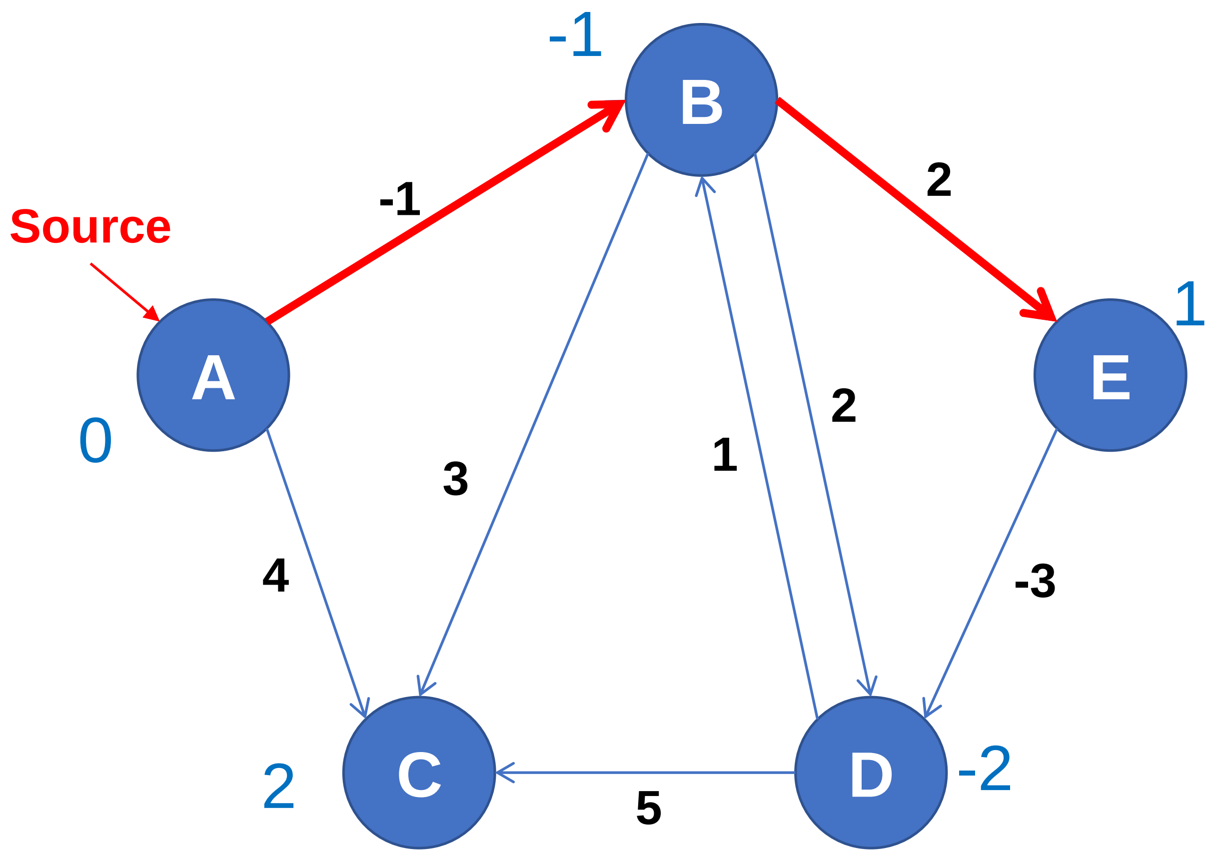
Vertex	d	π
A	0	NIL
B	-1	A
C	2	B
D	-2	E
E	1	B

Bellman Ford's algorithm



		distance	parent
	Vertex	d	π
	A	0	NIL
	B	-1	A
	C	2	B
	D	-2	E
	E	1	B

Bellman Ford's algorithm



Vertex	d	π
A	0	NIL
B	-1	A
C	2	B
D	-2	E
E	1	B

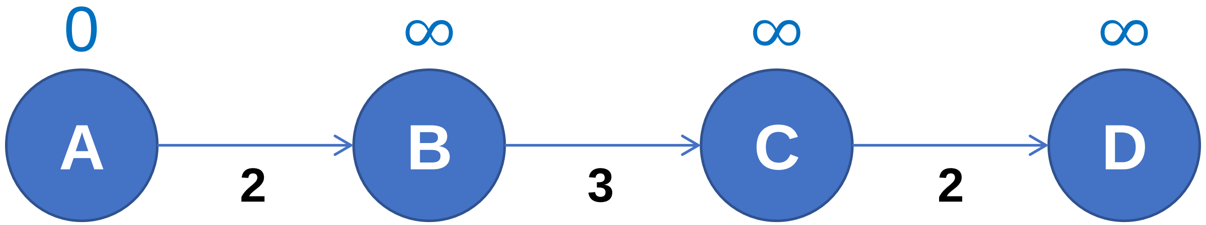
distance (d), parent (π)

Complexity

Time Complexity: O(VE)

Why do we need |V|-1 times

1st Iteration



C → D

B → C

$\boxed{\text{A → B}}$

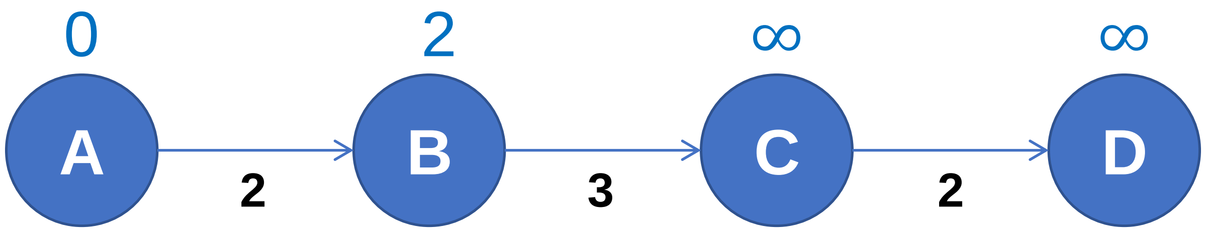
A → B

B → C

C → D

Why do we need |V|-1 times

1st Iteration



C → D

B → C

$\boxed{\text{A → B}}$

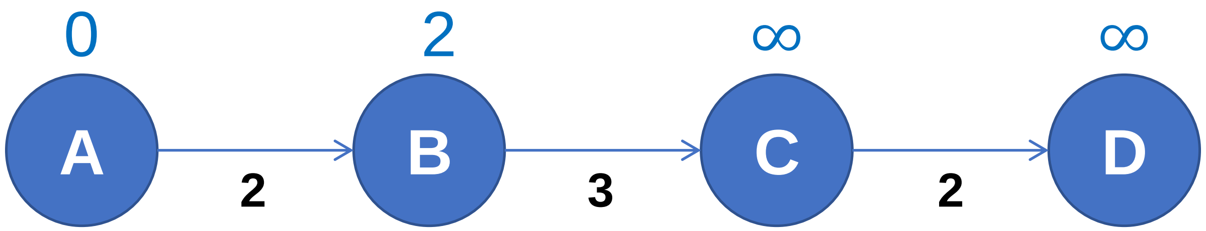
A → B

B → C

C → D

Why do we need |V|-1 times

2nd Iteration



C → D

$\boxed{\textbf{B → C}}$

A → B

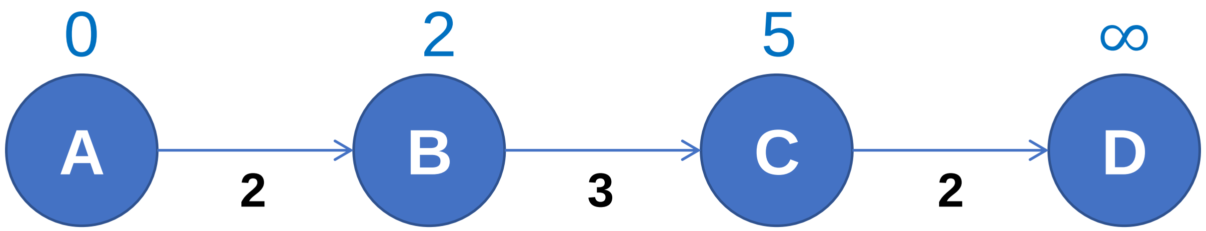
A → B

B → C

C → D

Why do we need |V|-1 times

2nd Iteration



C → D

$\boxed{\text{B → C}}$

A → B

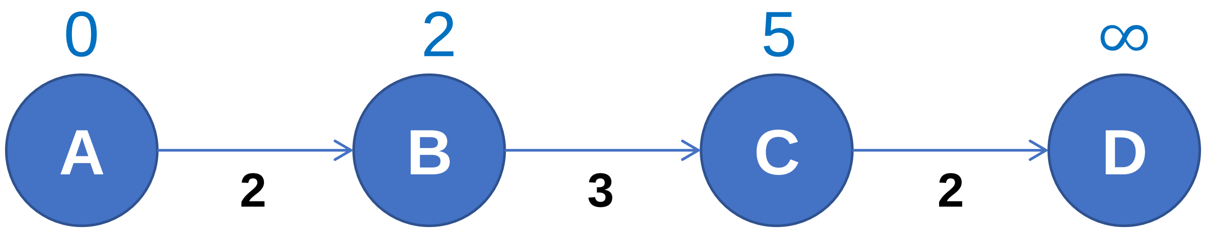
A → B

B → C

C → D

Why do we need |V|-1 times

3rd Iteration



C → D

B → C

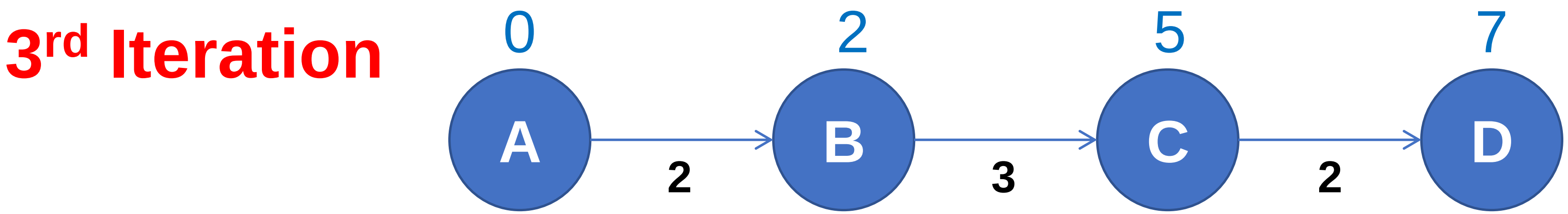
A → B

A → B

B → C

C → D

Why do we need |V|-1 times



$\boxed{C \rightarrow D}$

$B \rightarrow C$

$A \rightarrow B$

$A \rightarrow B$

$B \rightarrow C$

$C \rightarrow D$

Bellman-Ford in practice

Distance-vector routing protocol

– Repeatedly relax edges until convergence

– Relaxation is local!

On the Internet

– Routing Information Protocol (RIP)

– Interior Gateway Routing Protocol (IGRP)

Complexity

Time Complexity: $O(VE)$

Pseudo-Code

for v in V:
    $v.d = \infty$
    $v.\pi = \text{None}$
$s.d = 0$
for i from 1 to $|V| - 1$:
    for (u, v) in E:
        relax(u, v):
            if $v.d > u.d + w(u, v)$:
                $v.d = u.d + w(u, v)$
                $v.\pi = u$

Implementation

```
// a structure to represent a weighted edge in graph
struct Edge {
    int src, dest, weight;
};

// a structure to represent a connected, directed and
// weighted graph
struct Graph {
    // V-> Number of vertices, E-> Number of edges
    int V, E;

    // graph is represented as an array of edges.
    struct Edge* edge;
};

// Creates a graph with V vertices and E edges
struct Graph* createGraph(int V, int E)
{
    struct Graph* graph = new Graph;
    graph->V = V;
    graph->E = E;
    graph->edge = new Edge[E];
    return graph;
}
```

```
void BellmanFord(struct Graph* graph, int src)
{
    int V = graph->V;
    int E = graph->E;
    int dist[V];

    // Step 1: Initialize distances from src to all other vertices
    // as INFINITE
    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX;
    dist[src] = 0;

    // Step 2: Relax all edges |V| - 1 times. A simple shortest
    // path from src to any other vertex can have at-most |V| - 1
    // edges
    for (int i = 1; i <= V - 1; i++) {
        for (int j = 0; j < E; j++) {
            int u = graph->edge[j].src;
            int v = graph->edge[j].dest;
            int weight = graph->edge[j].weight;
            if (dist[u] != INT_MAX && dist[u] + weight < dist[v])
                dist[v] = dist[u] + weight;
        }
    }

    // Step 3: check for negative-weight cycles.  The above step
    // guarantees shortest distances if graph doesn't contain
    // negative weight cycle.  If we get a shorter path, then there
    // is a cycle.
    for (int i = 0; i < E; i++) {
        int u = graph->edge[i].src;
        int v = graph->edge[i].dest;
        int weight = graph->edge[i].weight;
        if (dist[u] != INT_MAX && dist[u] + weight < dist[v]) {
            printf("Graph contains negative weight cycle");
            return; // If negative cycle is detected, simply return
        }
    }

    printArr(dist, V);

    return;
}
```

Reference

- Charles Leiserson and Piotr Indyk, “*Introduction to Algorithms*”, September 29, 2004

- https://www.geeksforgeeks.org

- https://en.wikipedia.org/wiki