Graphs Shortest Path and MST

Dijkstra, Bellman-Ford, Prim and Kruskal

SoftUni Team

Technical Trainers

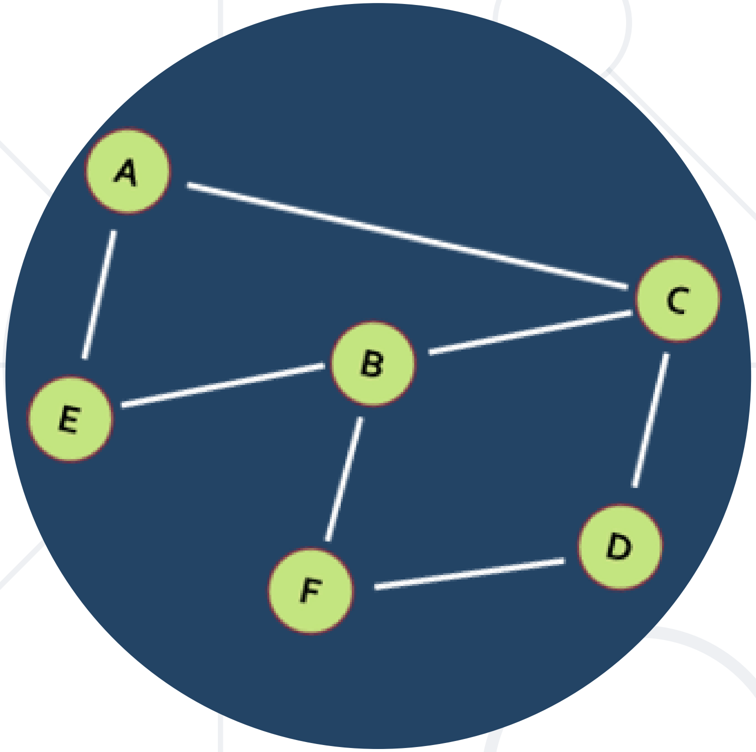
Software University

SoftUni

Software University

https://about.softuni.bg

Table of Contents

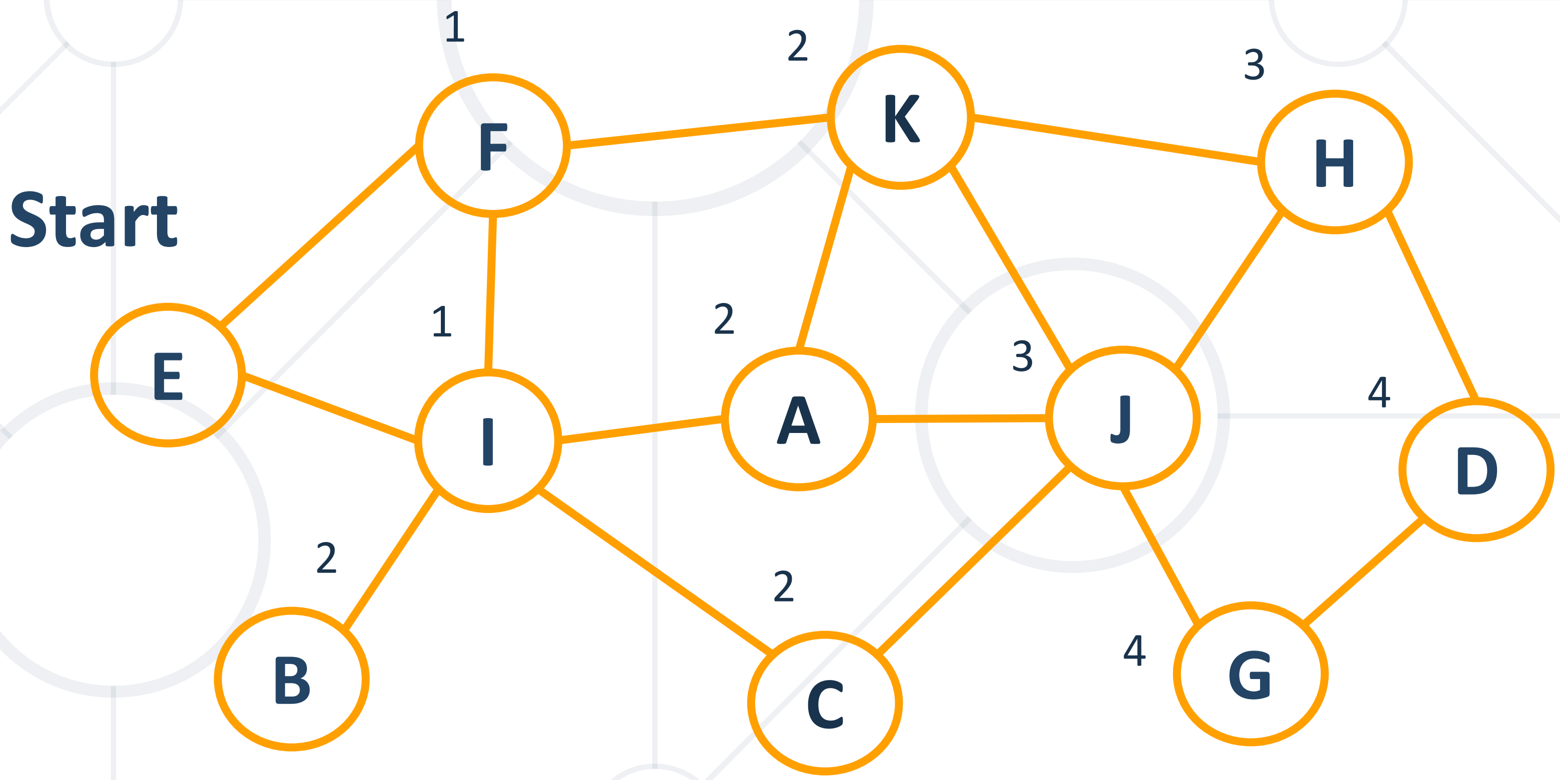


Shortest Path

Shortest Path in Unweighted Graph

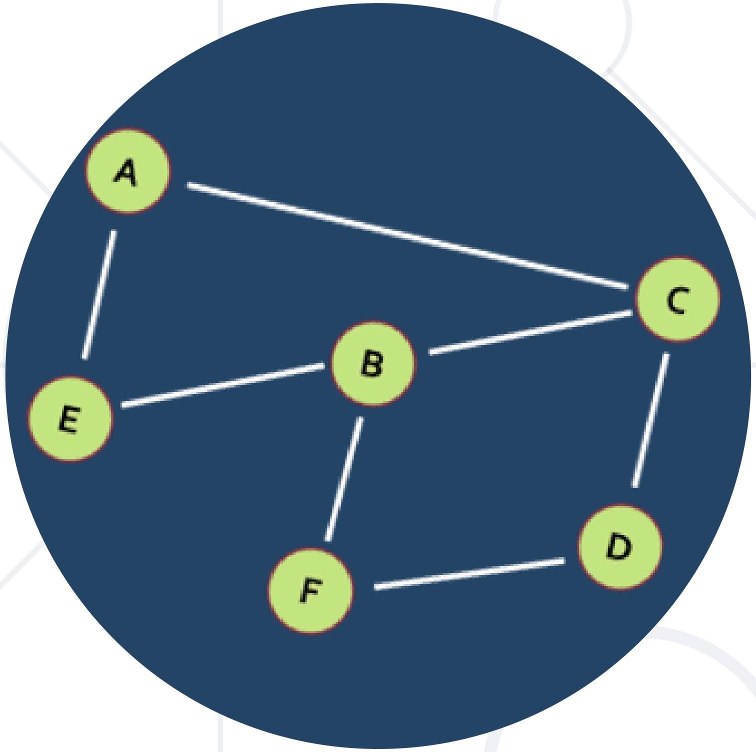
Shortest Path in Unweighted Graph

- In **unweighted** graphs finding the **shortest path** can be done with **BFS** (all edges have the same weight):



BFS Shortest Path

```
bfs(G, start, end)
    visited[start] = true
    queue.enqueue(start)
    while (!queue.isEmpty())
          v = queue.dequeue()
          if v is end
          return v
          for all edges from v to w in G.adjacentEdges(v) do
             if w is not labeled as discovered then
                 label w as discovered
                 w.parent = v
                 queue.enqueue(w)
```

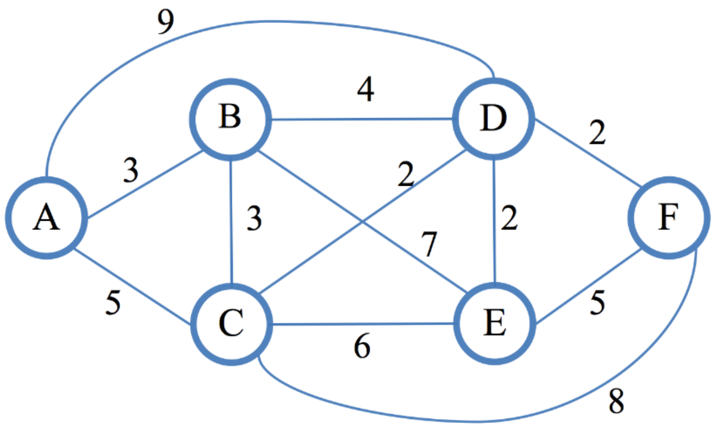


Dijkstra's Algorithm

Shortest Paths in Graph

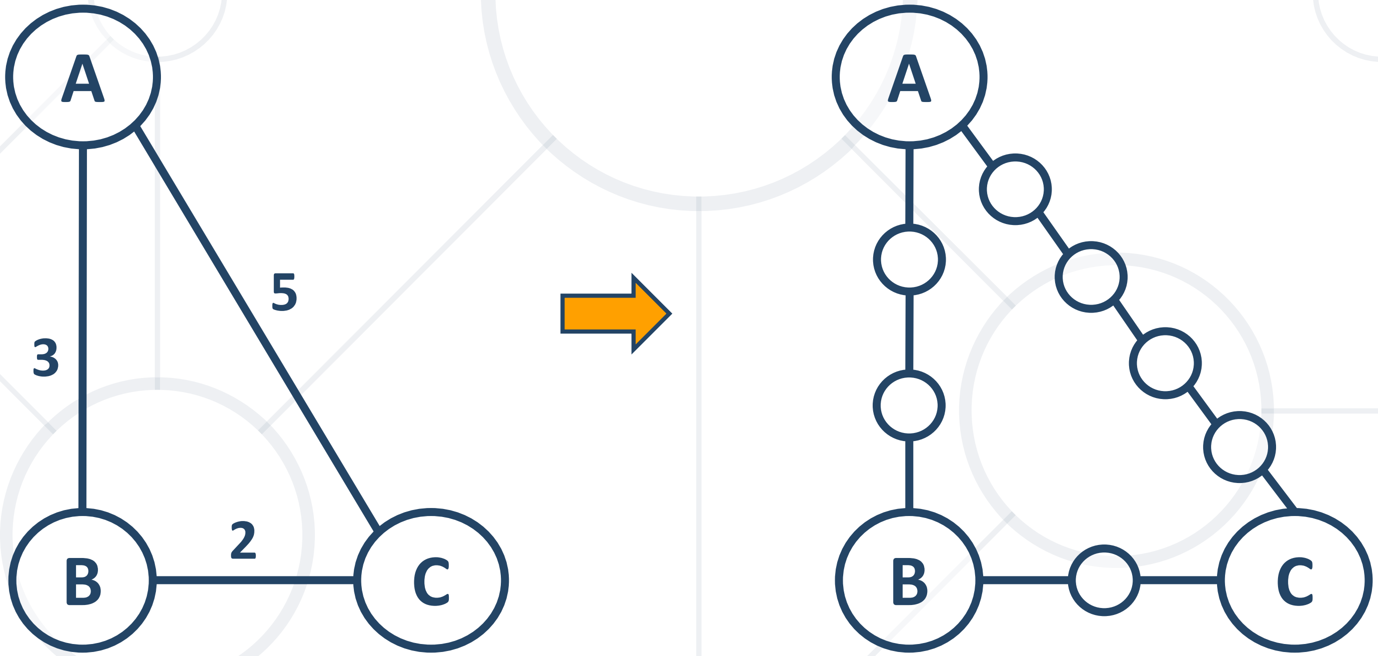
Dijkstra's Algorithm

- Dijkstra's algorithm finds the **shortest path** from given vertex to all other vertices in a directed / undirected **weighted graph**
  - First described by Edsger W. Dijkstra in 1956
- Assumptions
  - Weights on edges are non-negative
  - Edges can be directed or not
  - Weights do not have to be distances
  - Shortest path is not necessarily unique
  - Not all edges need to be reachable



Weighted Shortest Paths with BFS

- In weighted graphs
  - Break the edges into sub-vertices and use BFS



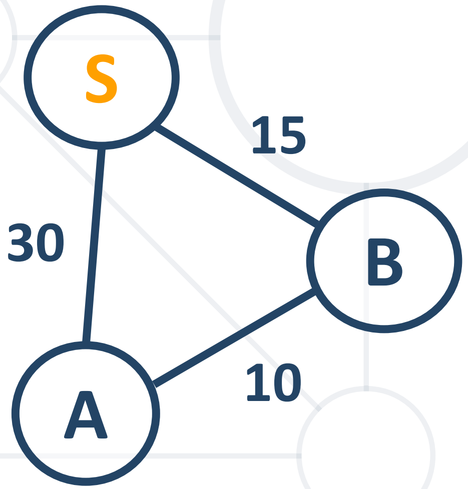
- * Too much memory usage even for smaller graphs!

Dijkstra's Algorithm

- **Dijskstra's algorithm** is similar to **BFS**
- Use a **priority queue** instead of **queue**
  - Keep the shortest distances so far
- Steps in Dijkstra's algorithm:

v	A	B
d[*v*]	30	15

v	A	B
d[*v*]	25	15

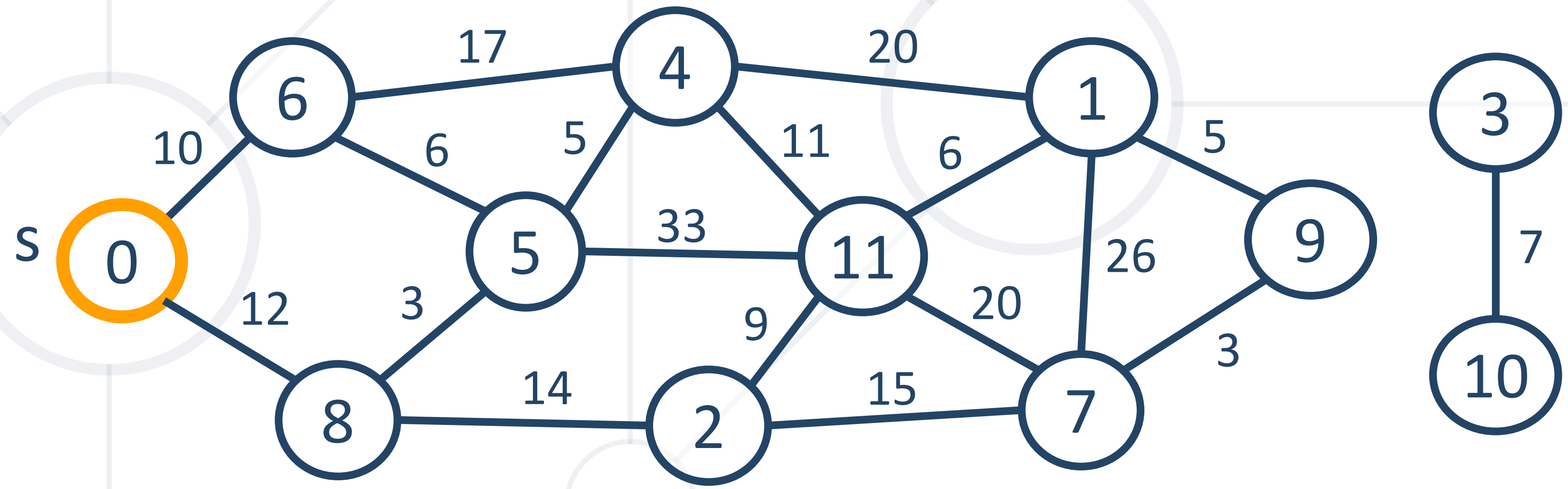



```
Initially calculate all direct distances d[] from S
Enqueue that start node S
While (queue not empty)
  Dequeue the nearest vertex B
  Enqueue all unvisited child nodes of B
  For each edge {B → A}, improve d[A] through B:
    d[S → A] = min(d[S → A], d[S → B] + weight[B → A])
```

Dijkstra's Algorithm: Step #1

- Initialize all distances d[] from s: d[0…n-1] = ∞; d[s] = 0
- Enqueue the start node (0)

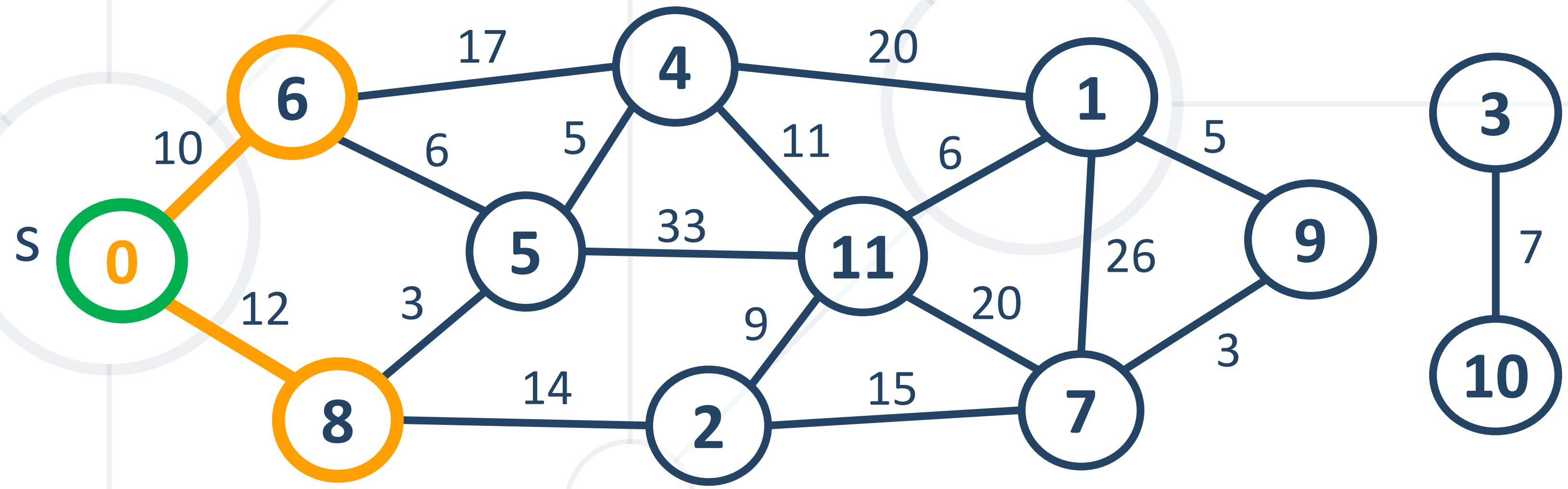
v	0	1	2	3	4	5	6	7	8	9	10	11
d[*v*]	0	-	-	-	-	-	-	-	-	-	-	-
prev[*v*]	-	-	-	-	-	-	-	-	-	-	-	-



Dijkstra's Algorithm: Step #2

- Dequeue the nearest vertex (0) and enqueue unvisited children: 6, 8
- Improve min distances through child edges of 0: {0 → 6}, {0 → 8}

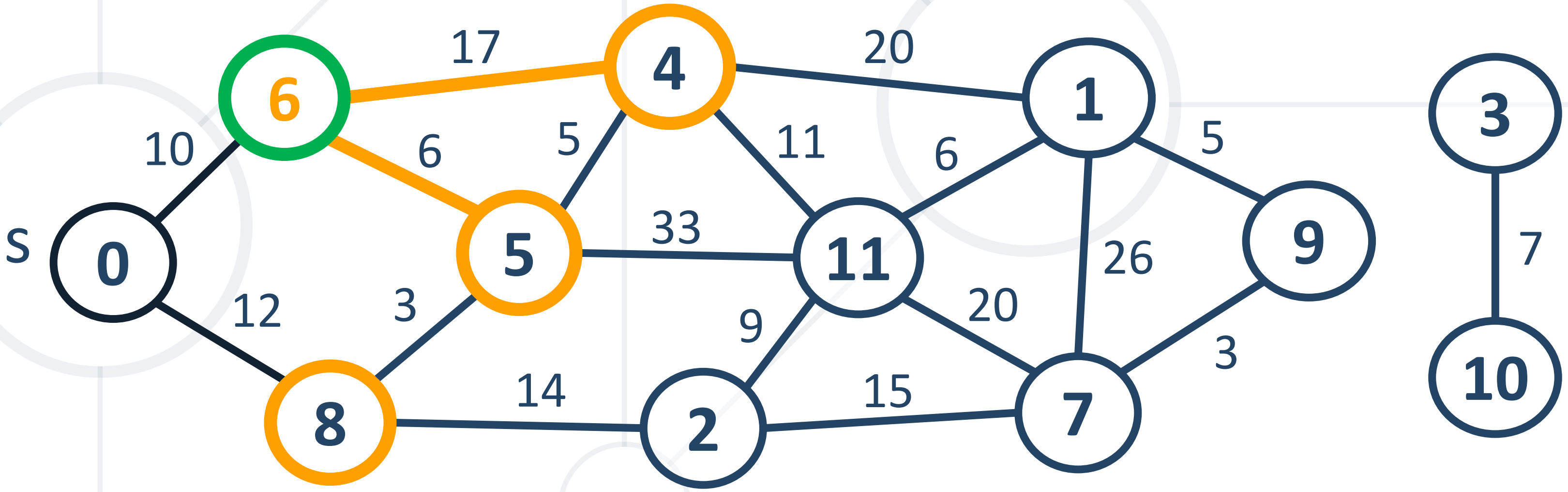
v	0	1	2	3	4	5	6	7	8	9	10	11
d[*v*]	0	-	-	-	-	-	10	-	12	-	-	-
prev[*v*]	-	-	-	-	-	-	0	-	0	-	-	-



Dijkstra's Algorithm: Step #3

- Dequeue the nearest vertex (6) and enqueue unvisited children: 4, 5
- Improve min distances through child edges of 6: {6 → 4}, {6 → 5}

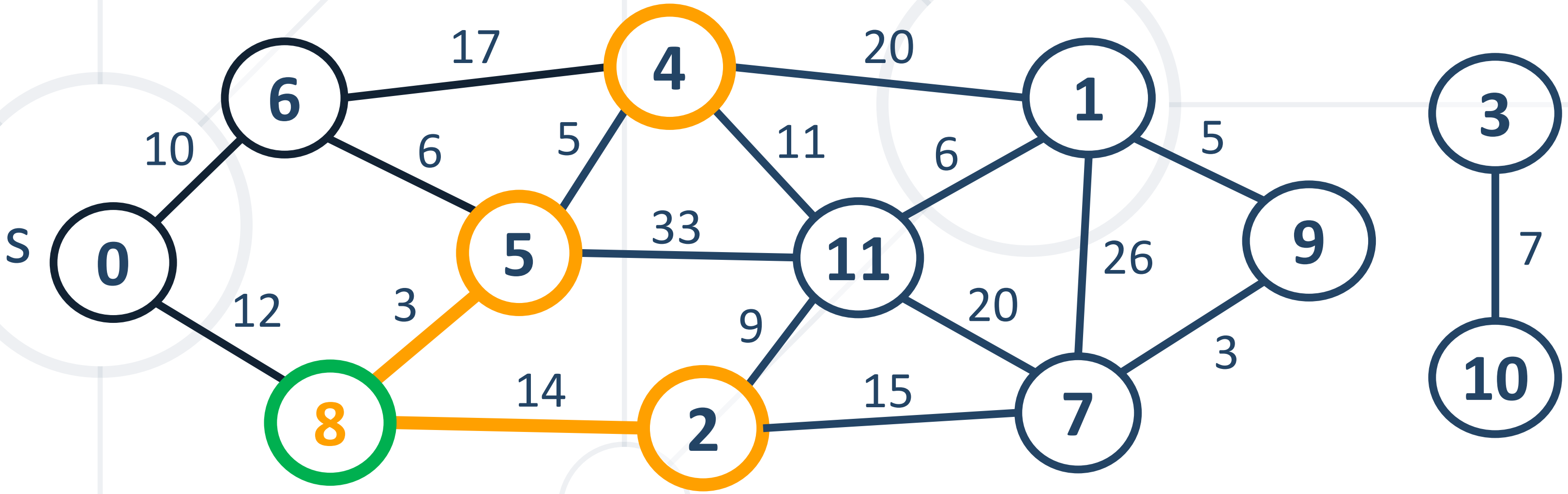
v	0	1	2	3	4	5	6	7	8	9	10	11
d[*v*]	0	-	-	-	27	16	10	-	12	-	-	-
prev[*v*]	-	-	-	-	6	6	0	-	0	-	-	-



Dijkstra's Algorithm: Step #4

- Dequeue the nearest vertex (8) and enqueue unvisited children: 2
- Improve min distances through child edges of 8: {8 → 2}, {8 → 5}

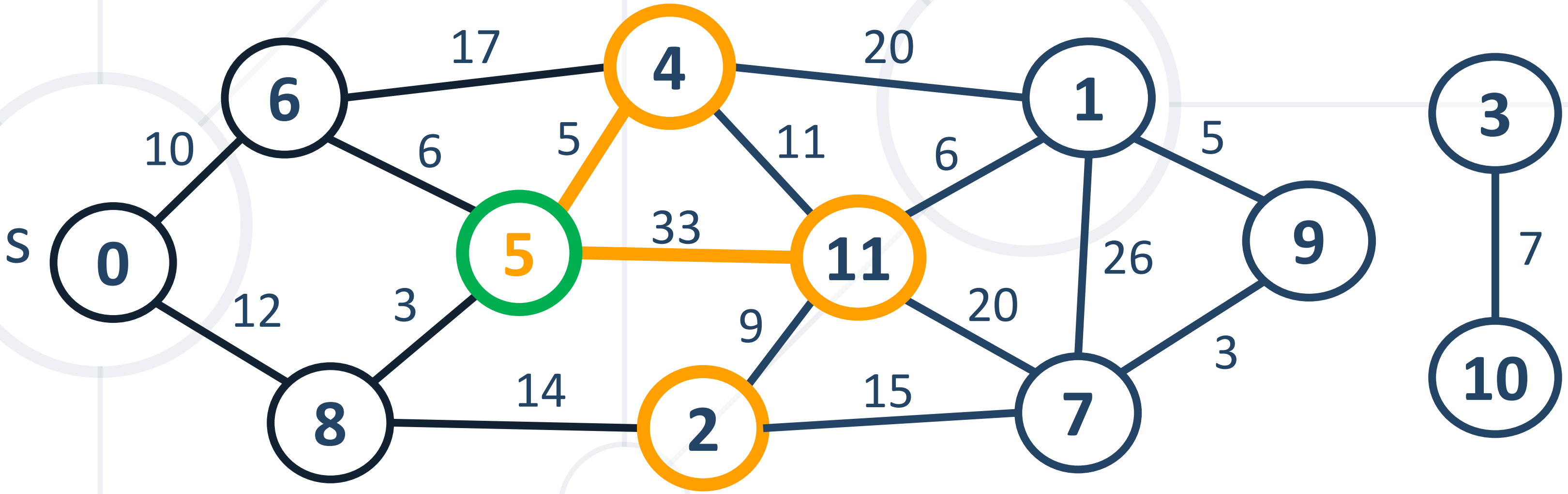
v	0	1	2	3	4	5	6	7	8	9	10	11
d[*v*]	0	-	26	-	27	15	10	-	12	-	-	-
prev[*v*]	-	-	8	-	6	8	0	-	0	-	-	-



Dijkstra's Algorithm: Step #5

- Dequeue the nearest vertex (**5**) and enqueue unvisited children: **11**
- Improve min distances through child edges of **5**: {5 → 4}, {5 → 11}

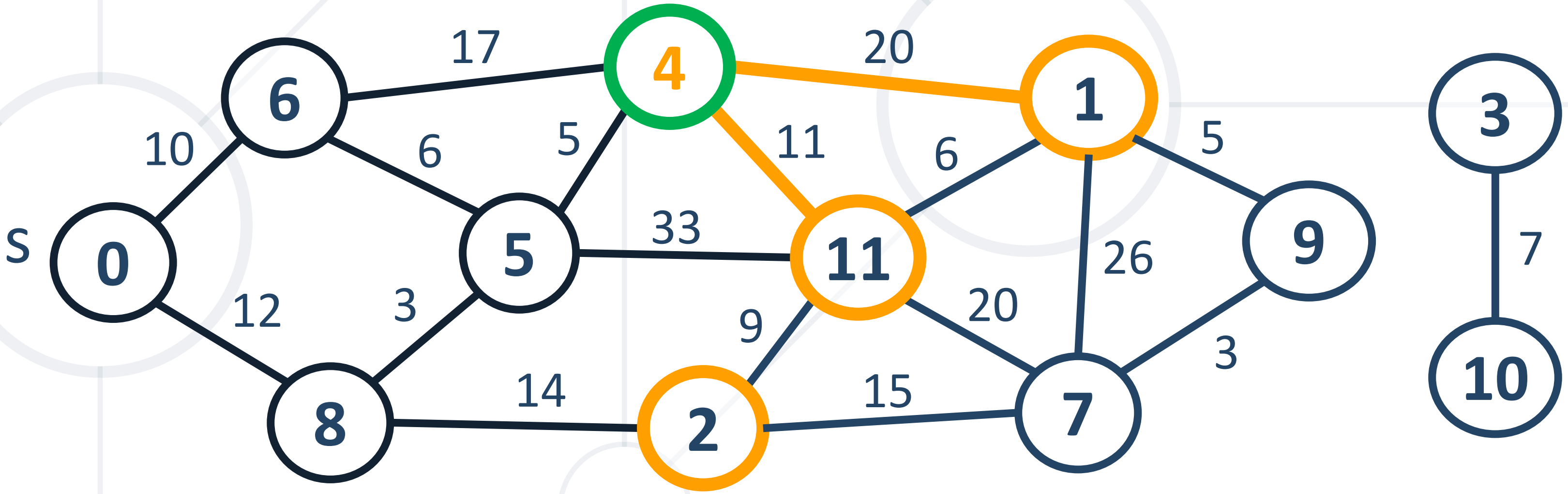
v	0	1	2	3	4	5	6	7	8	9	10	11
d[*v*]	0	-	26	-	20	15	10	-	12	-	-	48
prev[*v*]	-	-	8	-	5	8	0	-	0	-	-	5



Dijkstra's Algorithm: Step #6

- Dequeue the nearest vertex (4) and enqueue unvisited children: 1
- Improve min distances through child edges of 4: {4 → 1}, {4 → 11}

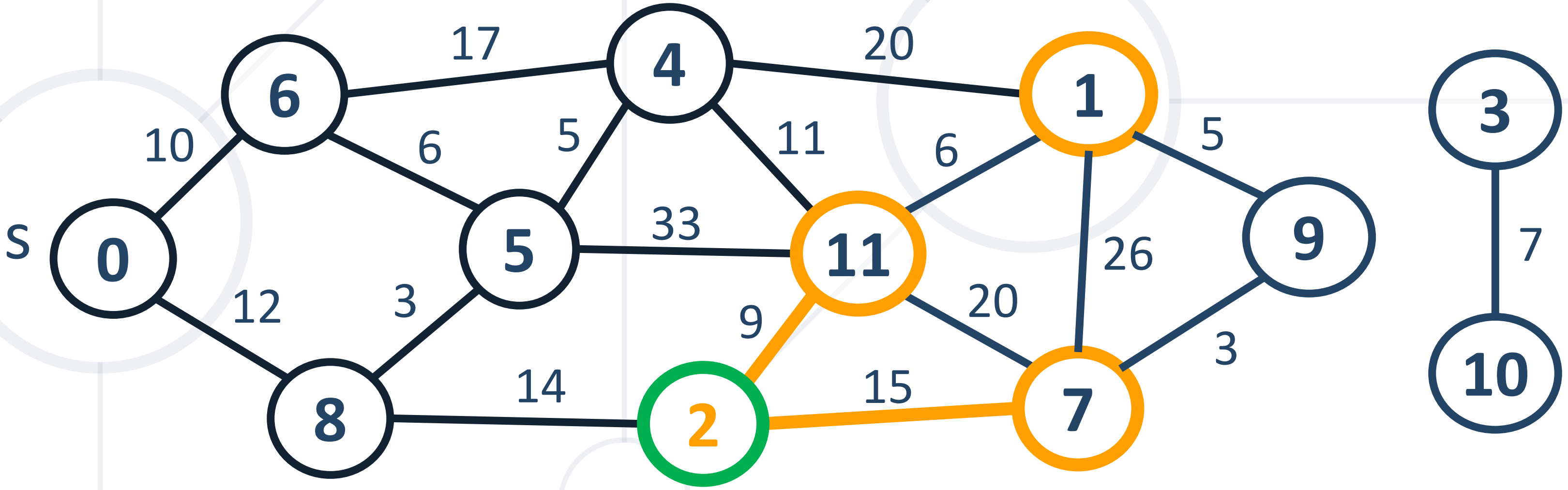
v	0	1	2	3	4	5	6	7	8	9	10	11
d[*v*]	0	40	26	-	20	15	10	-	12	-	-	31
prev[*v*]	-	4	8	-	5	8	0	-	0	-	-	4



Dijkstra's Algorithm: Step #7

- Dequeue the nearest vertex (2) and enqueue unvisited children: 7
- Improve min distances through child edges of 2: {2 → 7}, {2 → 11}

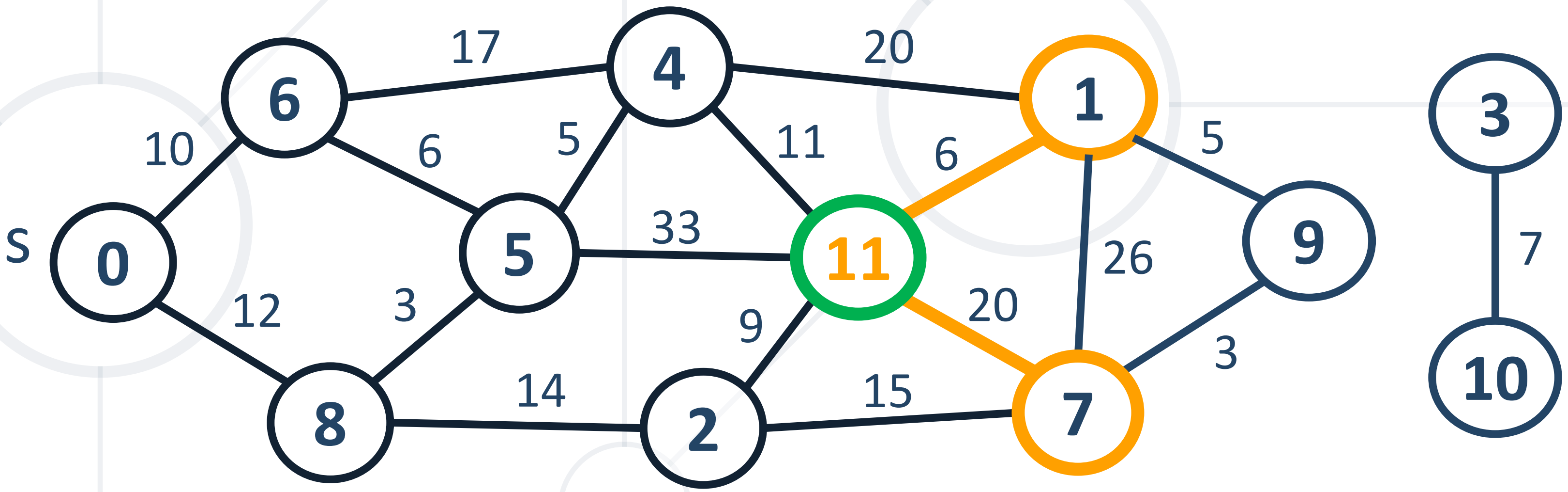
v	0	1	2	3	4	5	6	7	8	9	10	11
d[*v*]	0	40	26	-	20	15	10	41	12	-	-	31
prev[*v*]	-	4	8	-	5	8	0	2	0	-	-	4



Dijkstra's Algorithm: Step #8

- Dequeue the nearest vertex (**11**) and enqueue unvisited children: **none**
- Improve min distances through child edges of **11**: {11 → 1}, {11 → 7}

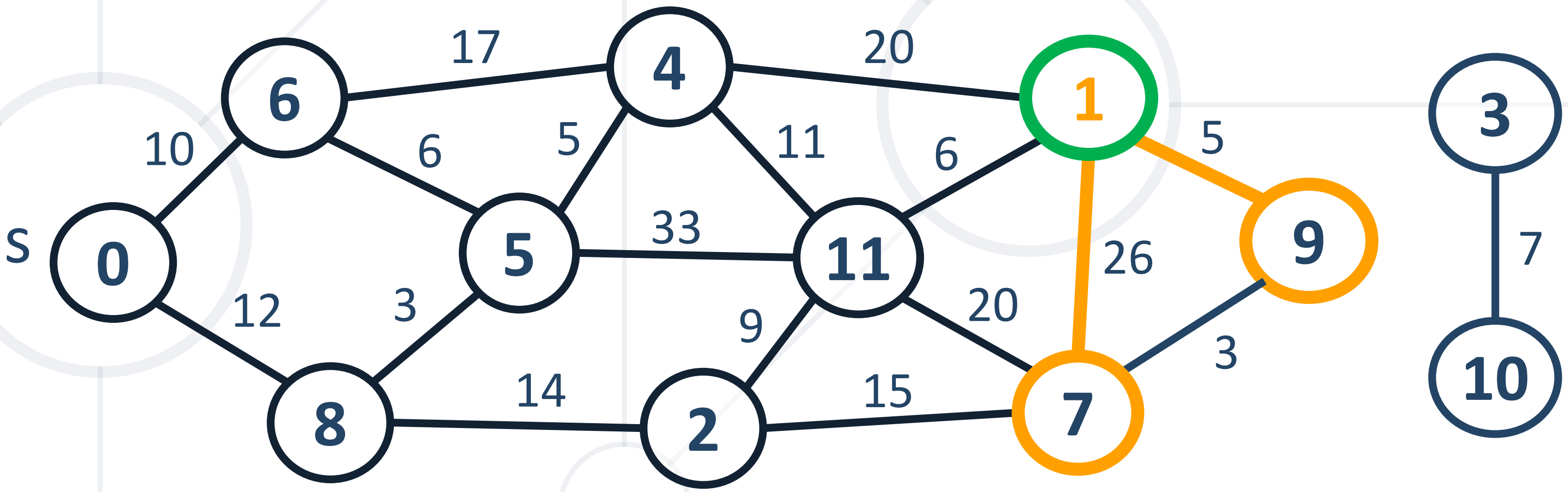
v	0	1	2	3	4	5	6	7	8	9	10	11
d[*v*]	0	**37**	26	-	20	15	10	**41**	12	-	-	31
prev[*v*]	-	**11**	8	-	5	8	0	**2**	0	-	-	4



Dijkstra's Algorithm: Step #9

- Dequeue the nearest vertex (1) and enqueue unvisited children: 9
- Improve min distances through child edges of 1: {1 → 7}, {1 → 9}

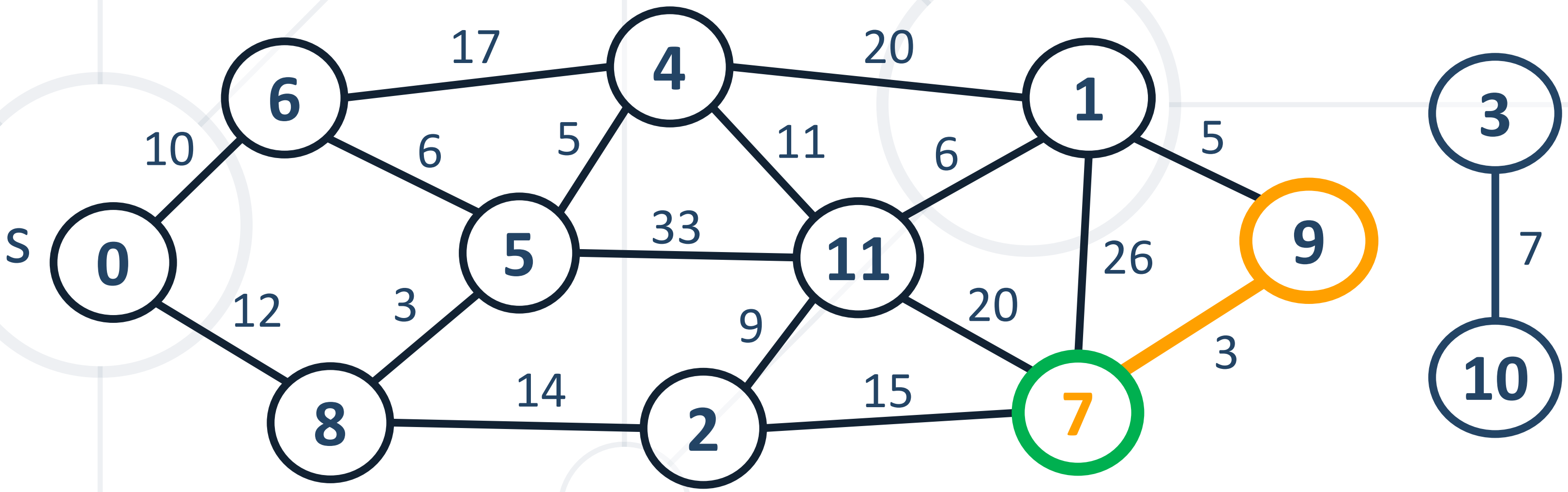
v	0	1	2	3	4	5	6	7	8	9	10	11
d[*v*]	0	37	26	-	20	15	10	41	12	42	-	31
prev[*v*]	-	11	8	-	5	8	0	2	0	1	-	4



Dijkstra's Algorithm: Step #10

- Dequeue the nearest vertex (**7**) and enqueue unvisited children: **none**
- Improve min distances through child edges of **7**: {7 → 9}

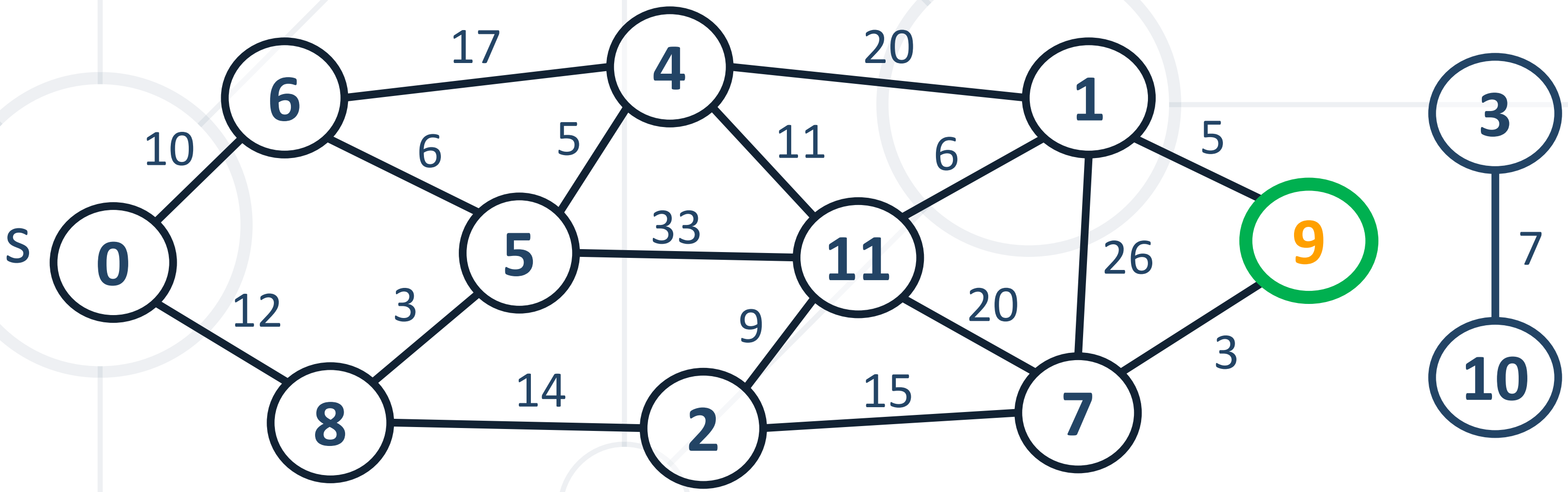
v	0	1	2	3	4	5	6	7	8	9	10	11
d[*v*]	0	37	26	-	20	15	10	41	12	**42**	-	31
prev[*v*]	-	11	8	-	5	8	0	2	0	**1**	-	4



Dijkstra's Algorithm: Step #11

- Dequeue the nearest vertex (**9**) and enqueue unvisited children: **none**
- Improve min distances through child edges of **9**: none

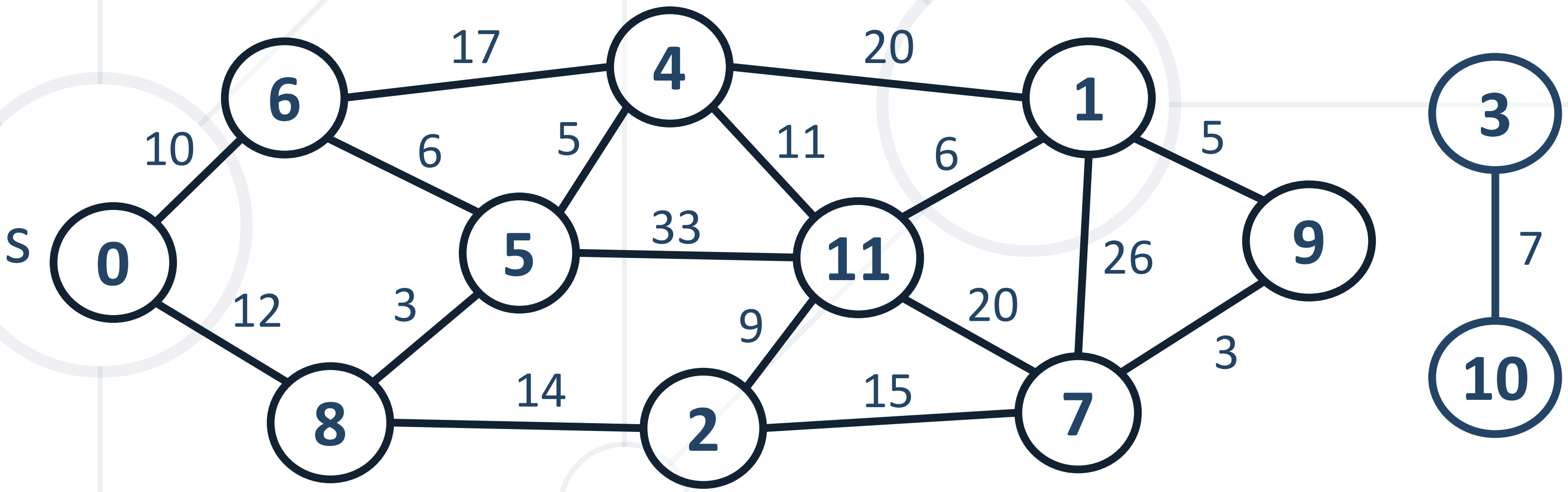
v	0	1	2	3	4	5	6	7	8	9	10	11
d[*v*]	0	37	26	-	20	15	10	41	12	**42**	-	31
prev[*v*]	-	11	8	-	5	8	0	2	0	**1**	-	4



Dijkstra's Algorithm: Step #12

- The queue is empty → Dijkstra's algorithm is completed
- **d[*v*]** hold shortest distances; **prev[*v*]** holds **shortest paths tree** edges

v	0	1	2	3	4	5	6	7	8	9	10	11
d[*v*]	0	37	26	-	20	15	10	41	12	**42**	-	31
prev[*v*]	-	11	8	-	5	8	0	2	0	**1**	-	4

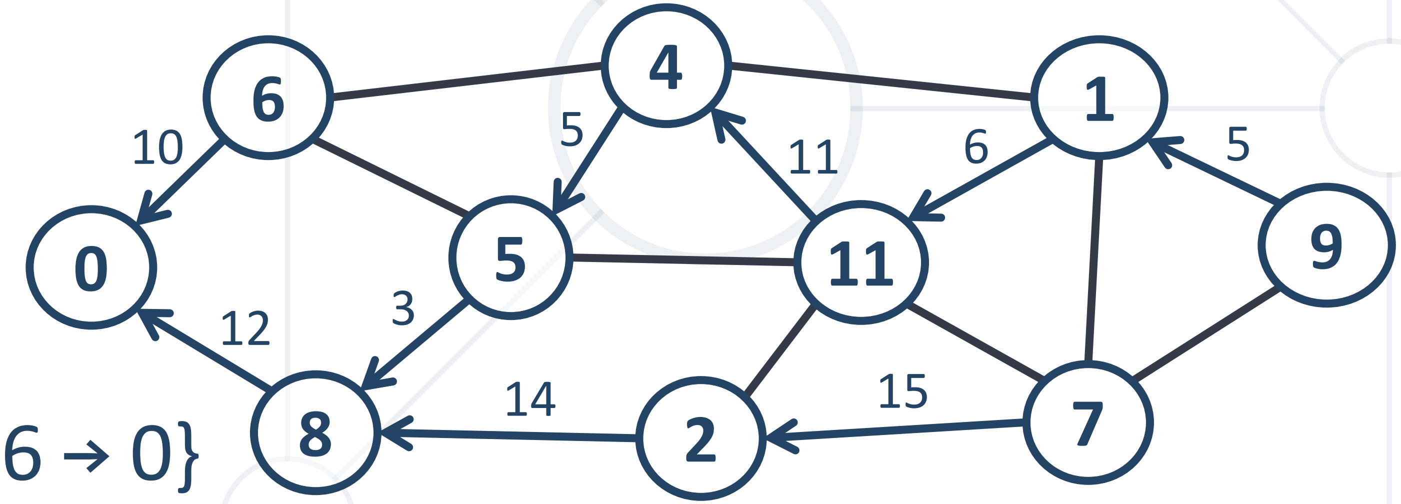


Dijkstra's Algorithm: Step #13

- The output is the **shortest paths tree** from the starting node to all others
- Reconstruct the path destination to source using `prev[v]`

v	0	1	2	3	4	5	6	7	8	9	10	11
d[v]	0	37	26	-	20	15	10	41	12	**42**	-	31
prev[v]	-	11	8	-	5	8	0	2	0	**1**	-	4

- `prev[v]` holds the
- shortest paths tree edges Path[9 → 0] = {9 → 1 → 11 → 4 → 5 → 6 → 0}

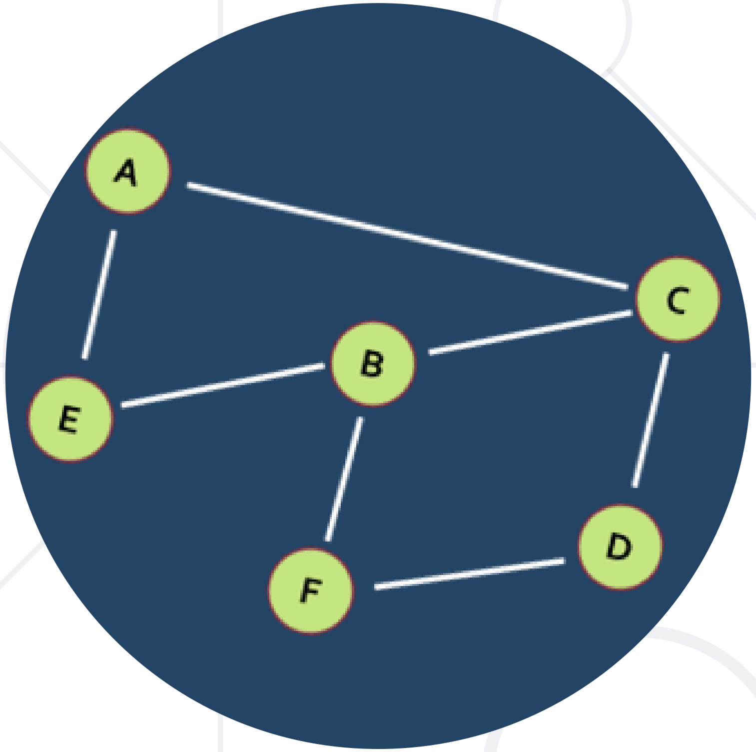


Dijkstra's Algorithm – Pseudo Code

```
d[0…n-1] = INFINITY; d[startNode] = 0
Q = priority queue holding nodes ordered by distance d[]
startNode add to Q
while (Q is not empty)
  minNode = dequeue the smallest node from Q
  if (d[minNode] == INFINITY) break;
  foreach (child c of minNode)
    if (c is unvisited) c add to Q
    newDistance = d[minNode] + distance {minNode → c}
    if (newDistance < d[c])
      d[c] = newDistance;
      reorder Q;
}
```

Dijkstra's Algorithm – More Details

- Modifications
  - Implementation with **array**, **priority queue**
  - Having a target node + stop when it is found
  - Saving the shortest paths tree (**`prev[v]`**)
- Complexity depends on the implementation
  - Typical implementation (with array): **$O(|V|^2)$**
  - With priority queue: **$O((|V|+|E|)*\log(|V|))$**
- Applications – maps, GPS, networks, air travel, etc.

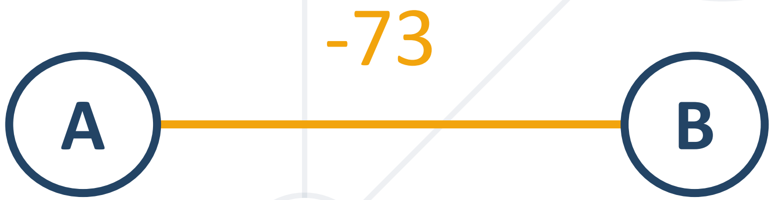


Negative Cycles and Edges

Introducing The Undefined Graph Path

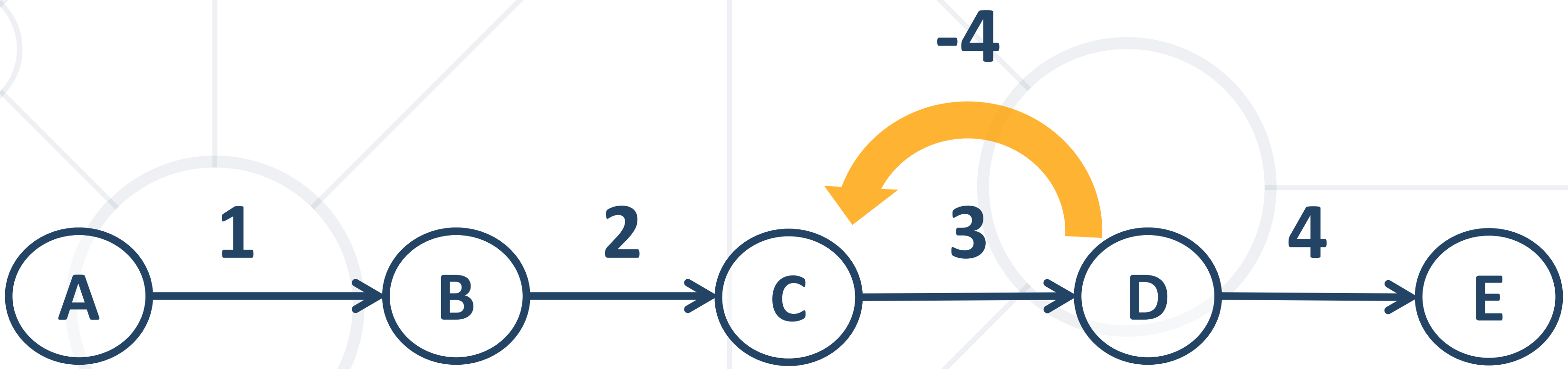
Negative Edge

- What is a negative edge:
  - Edge with **weight less** than **zero**
  - Can be presented in any **context** in the graph
  - Can be both **directed** or **undirected**
  - Can be a part of a **cycle**

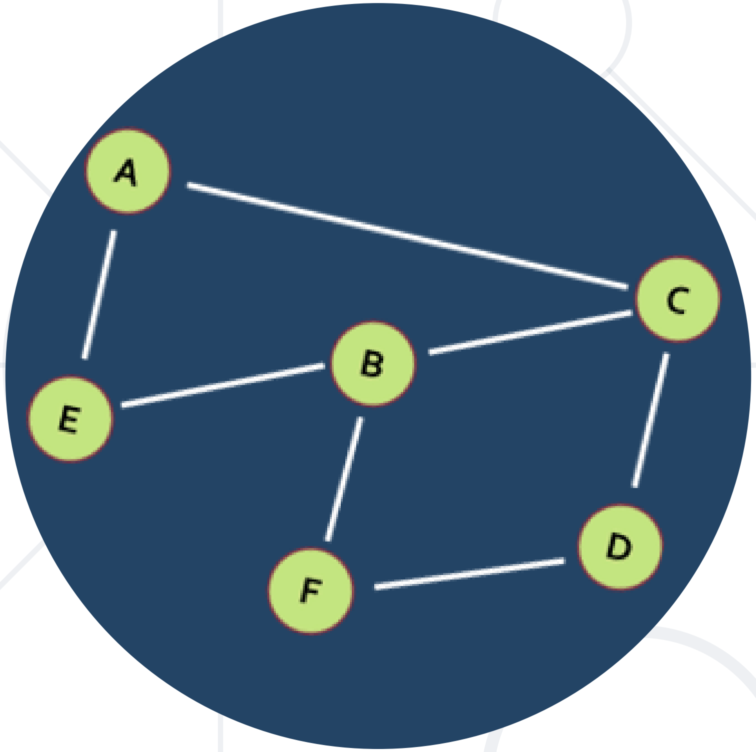


Negative Weight Cycles

- Negative **weight** cycle in graph
  - **Cycle** with **weights** that **sum** to a **negative** number
  - If there is **negative** cycle **reachable** from the **source** node, then the path is **undefined**



  - Path from **A** to **E** is **undefined**

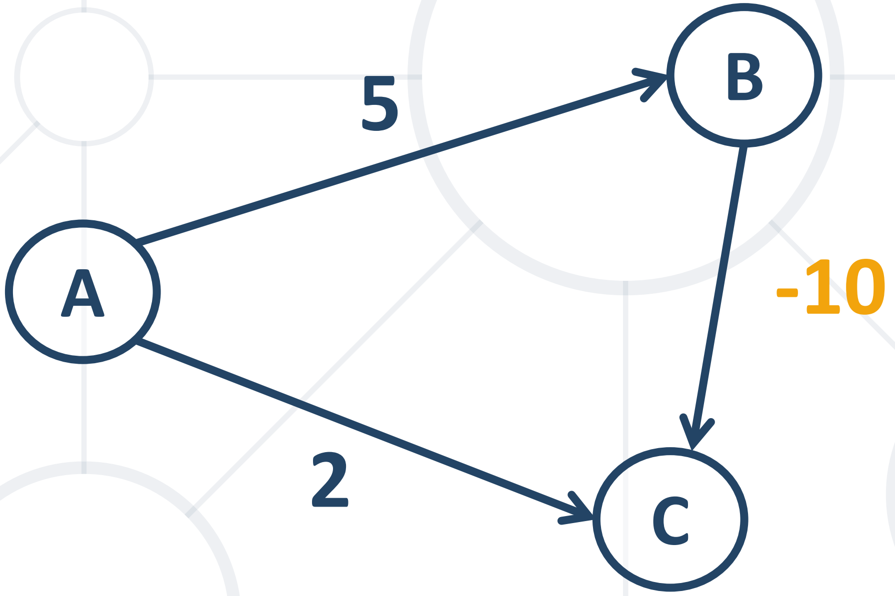


Negative Weights and Dijkstra

Dijkstra’s Killer

Negative Weights and Dijkstra

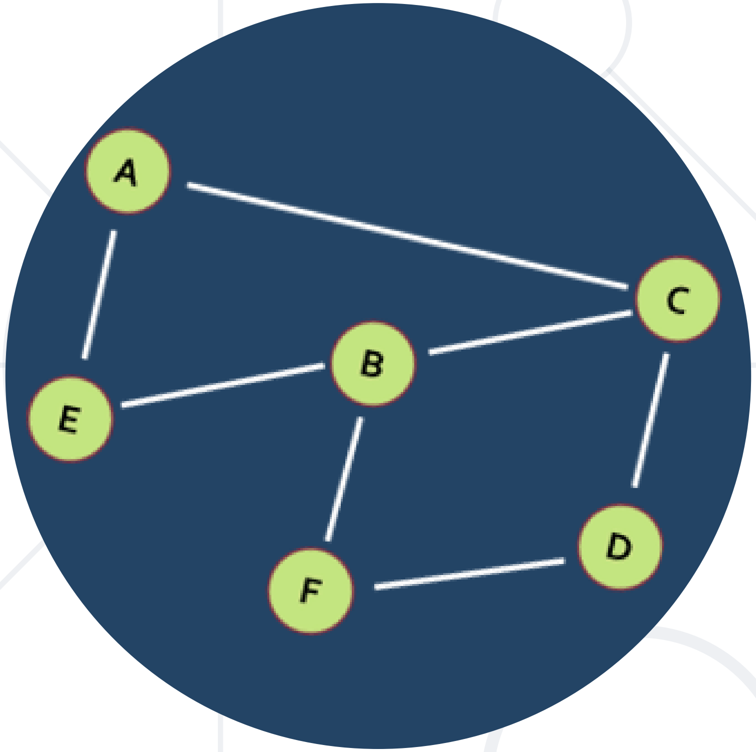
- Consider the following graph what is the shortest path **(A, C)**?



- The output will be **2** for **A** to **C**
- We can see that the correct answer is **-5** for **A** to **B** to **C**

Negative Weights and Dijkstra

- Why does Dijkstra **fail** with negative edges?
  - Dijkstra assumes that once we mark the node as **visited** as a parent node the **shortest path** to it is **found**
  - The above assumption is **true** for **non-negative** weights
  - We **never** can change the **minimum** by adding any **positive** number, however we **can** by adding **negative** one



Bellman-Ford Algorithm

Shorts Path in Graph with Negative Edges

Bellman-Ford Algorithm

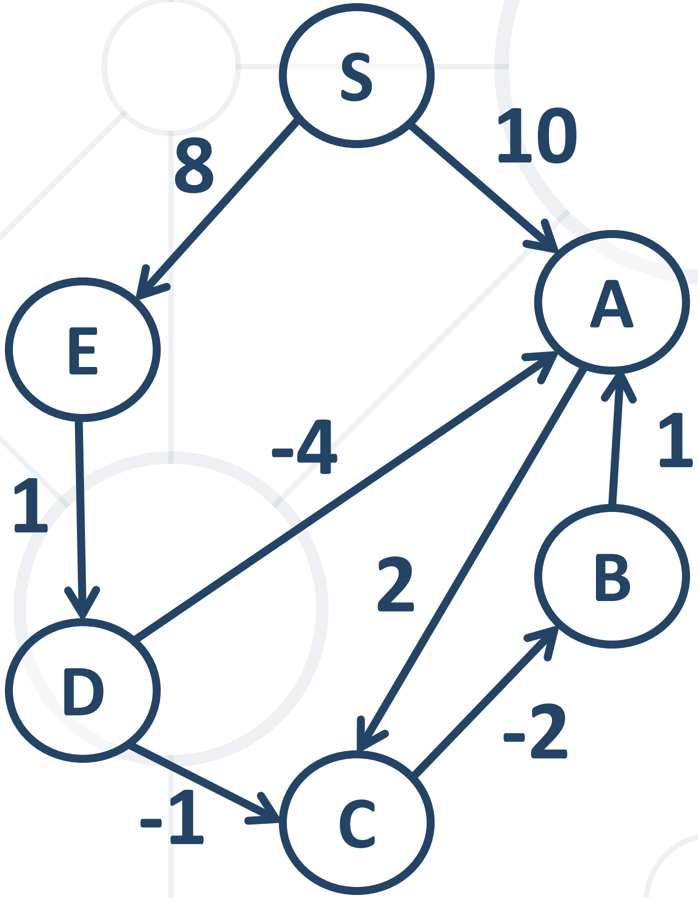
- Computes **shortest** paths from a **source** vertex to **all** of the **other** vertices in a **weighted** digraph
- Named after Richard **Bellman** and Lester **Ford** Jr., who published it in **1958** and **1956**, respectively
- Can **detect** and **report** negative cycles
- Time complexity: **O(VE)**

Bellman-Ford Algorithm

- The Bellman-Ford algorithm will do **V - 1** iterations where **V** is the **number** of vertices
  - For each iteration:
    - For each edge in the graph **(u, v, w)**
      - If **d[v] > d[u] + w(u, v)** and d[v] is visited before
      - Update **d[v]** with **d[u] + w(u, v)**
      - Update the **prev[v] = u**
- Run the algorithm **one** more time **for each edge**
  - If you can **update any d[v]** there is a **negative cycle**

Bellman-Ford in Action (step 1)

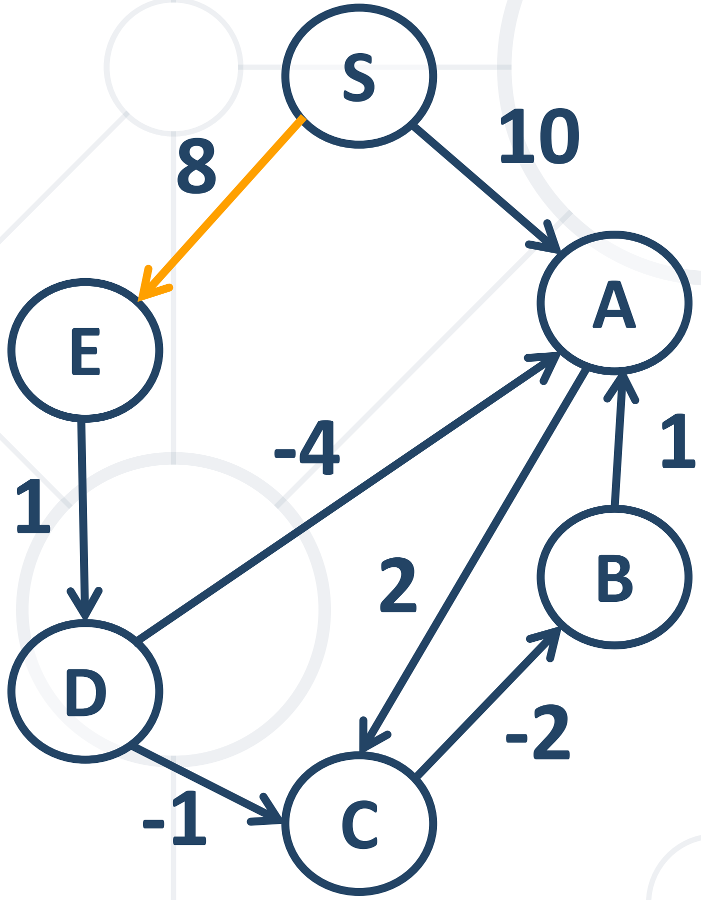
- We have **6** vertices so **5** iterations and **S** is the starting vertex



v	S	A	E	D	B	C
d[*v*]	0	-	-	-	-	-

Bellman-Ford in Action (step 2)

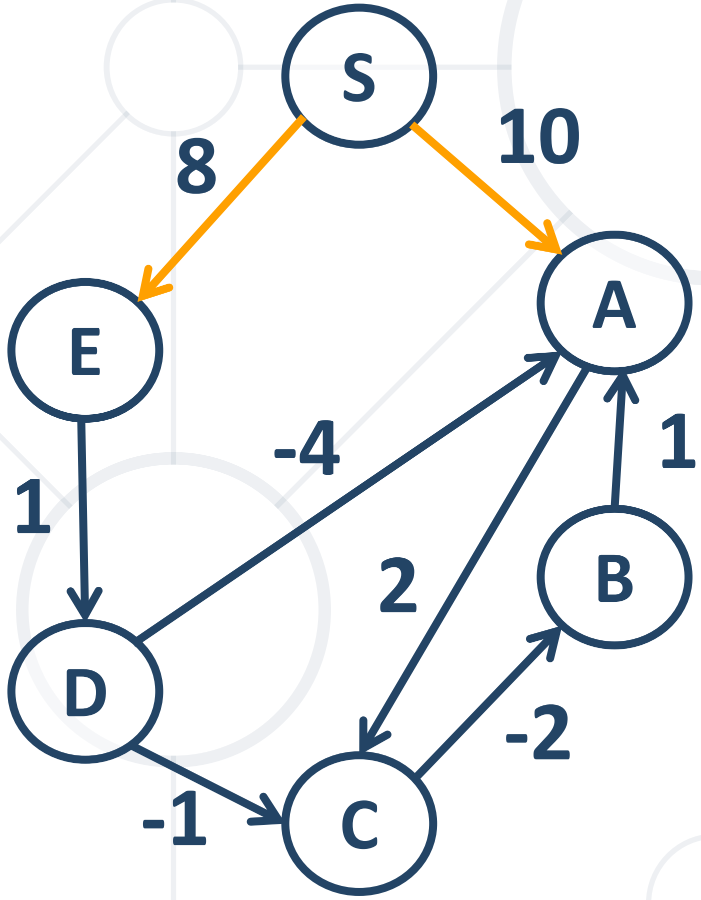
- Iteration #1:



v	S	A	E	D	B	C
d[*v*]	0	-	8	-	-	-

Bellman-Ford in Action (step 3)

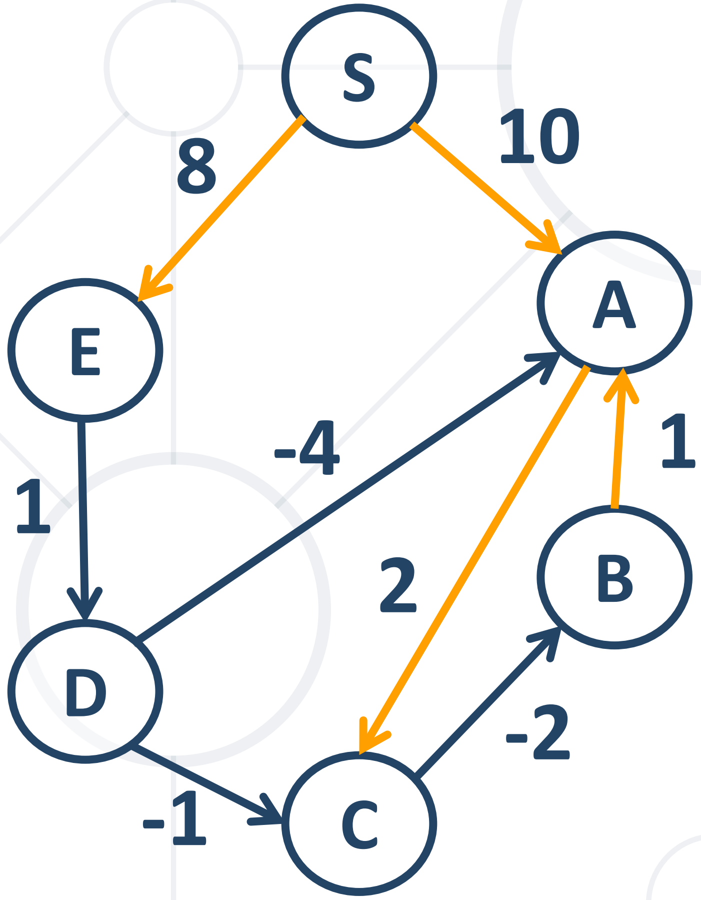
- Iteration #1:



v	S	A	E	D	B	C
d[*v*]	0	10	8	-	-	-

Bellman-Ford in Action (step 4)

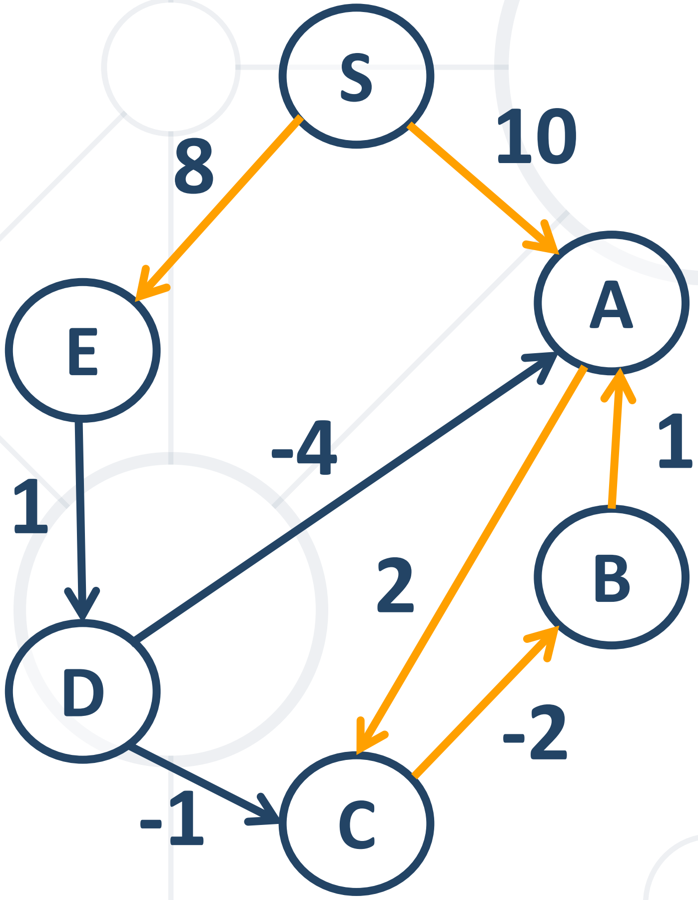
- Iteration #1:



v	S	A	E	D	B	C
d[*v*]	0	10	8	-	-	12

Bellman-Ford in Action (step 5)

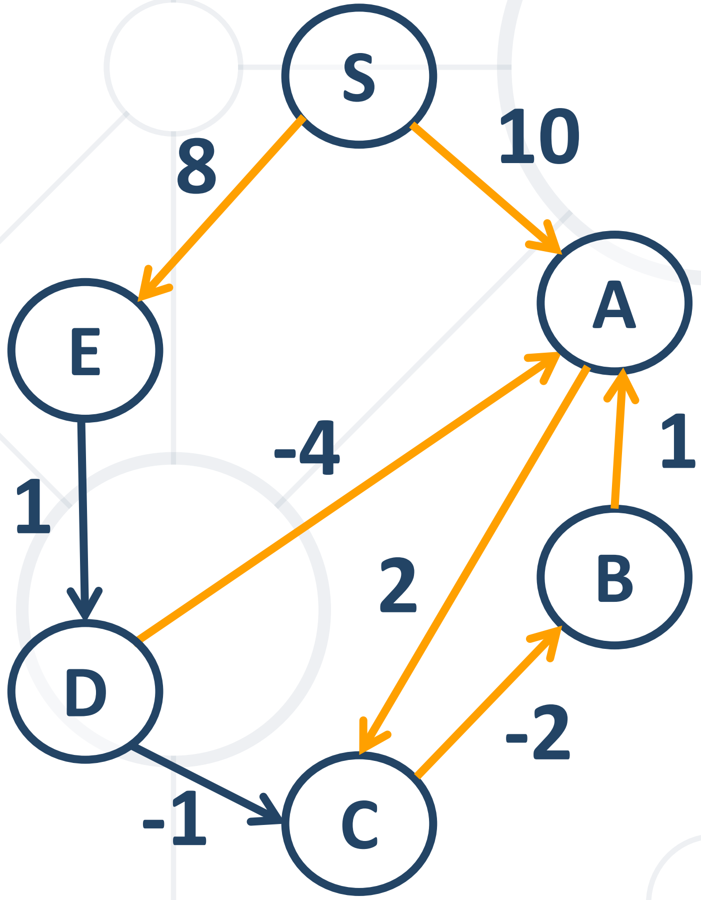
- Iteration #1:



v	S	A	E	D	B	C
d[*v*]	0	10	8	-	10	12

Bellman-Ford in Action (step 6)

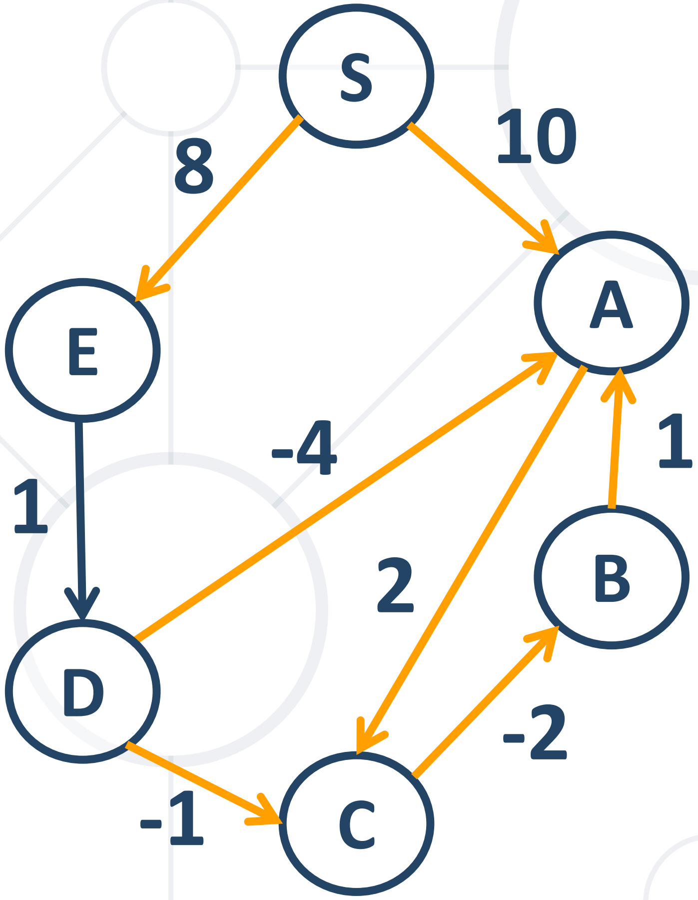
- Iteration #1:



v	S	A	E	D	B	C
d[*v*]	0	10	8	-	10	12

Bellman-Ford in Action (step 7)

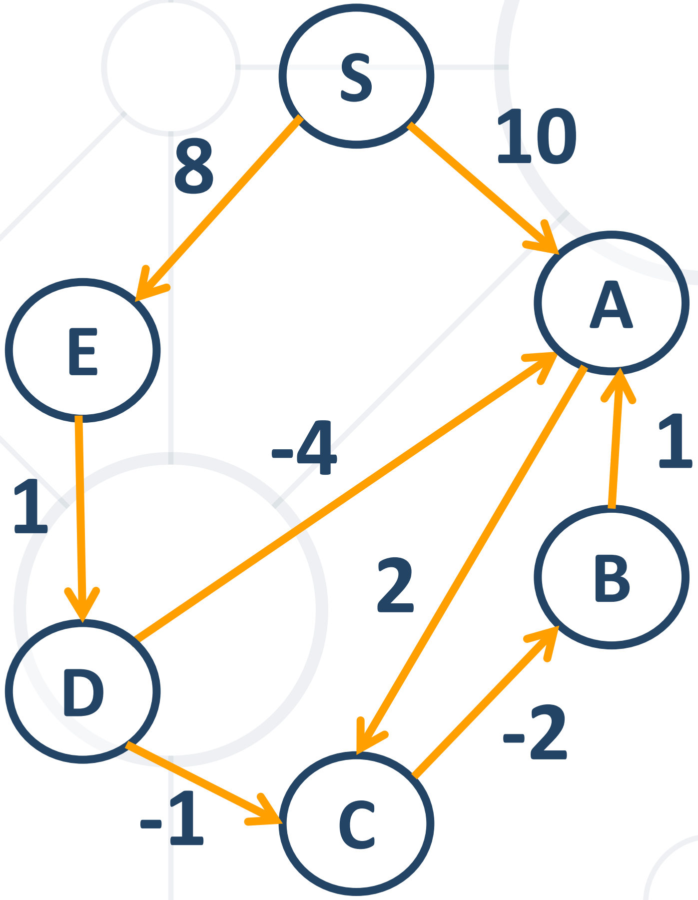
- Iteration #1:



v	S	A	E	D	B	C
d[*v*]	0	10	8	-	10	12

Bellman-Ford in Action (step 8)

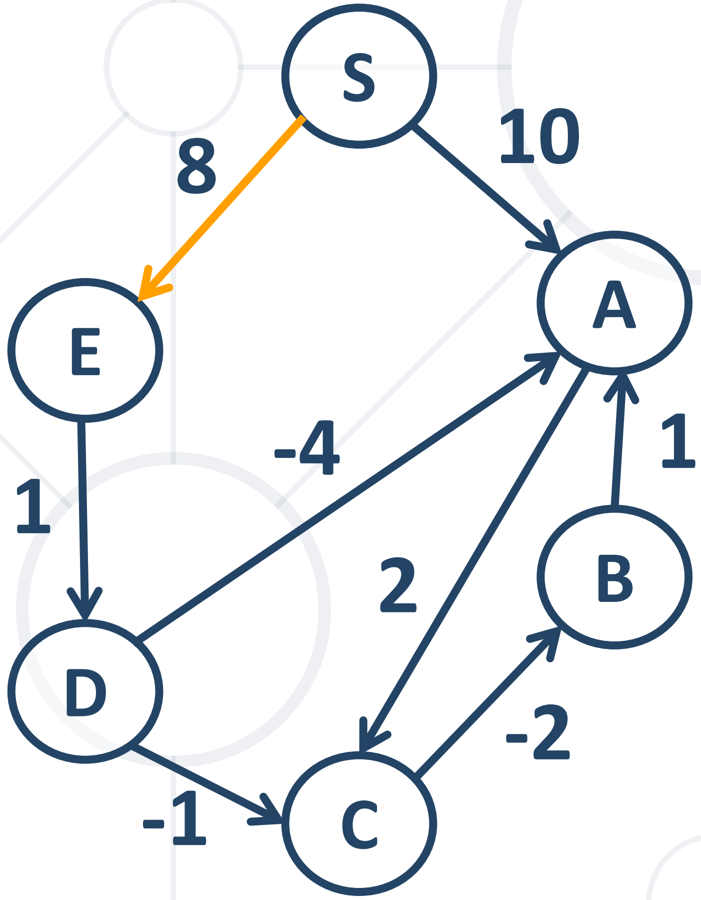
- Iteration #1:



v	S	A	E	D	B	C
d[*v*]	0	10	8	9	10	12

Bellman-Ford in Action (step 9)

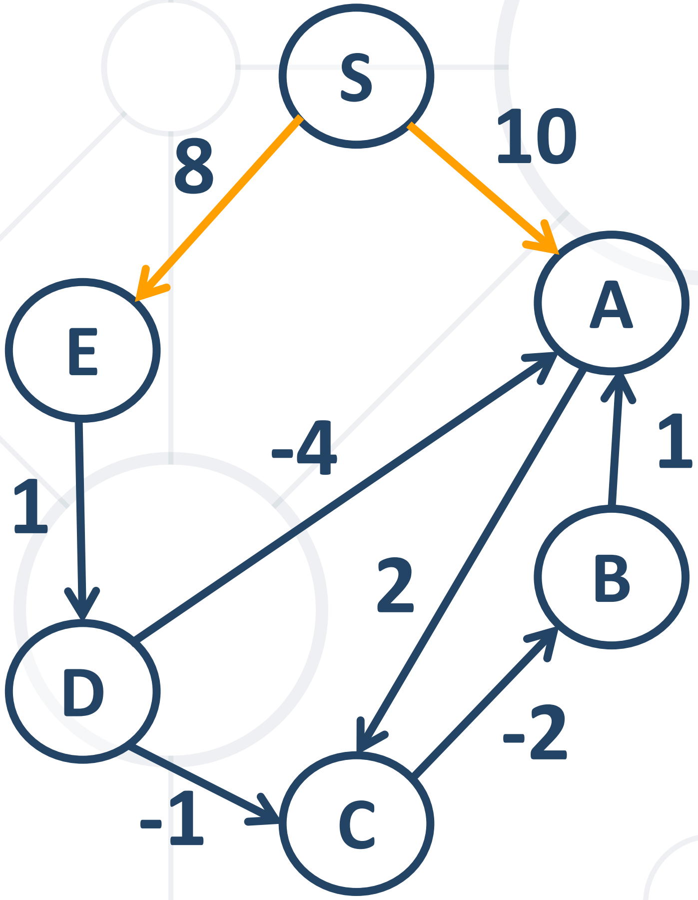
- Iteration #2:



v	S	A	E	D	B	C
d[*v*]	0	10	8	9	10	12

Bellman-Ford in Action (step 10)

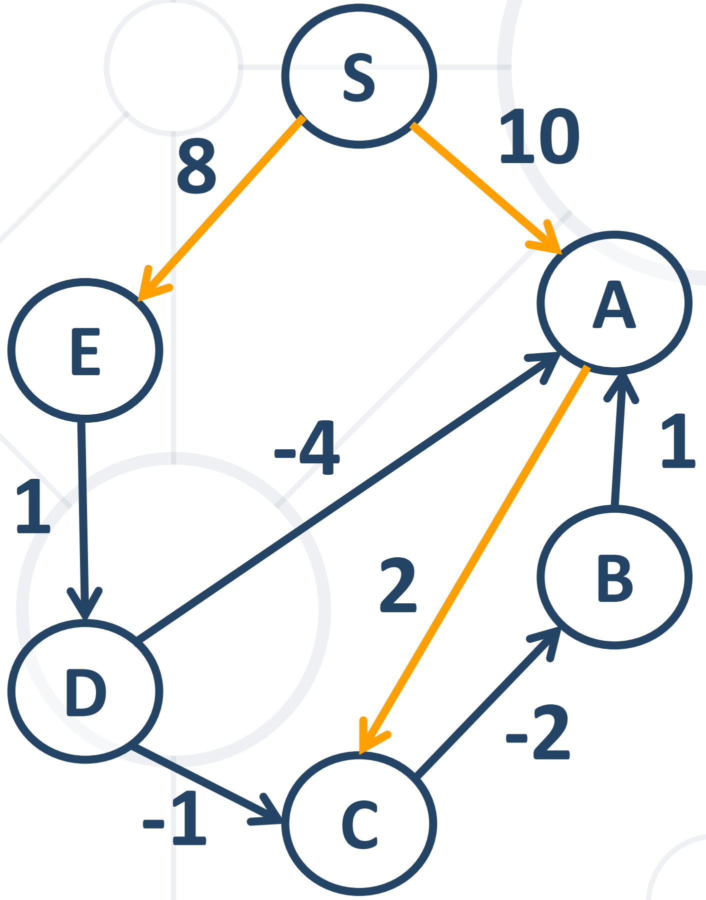
- Iteration #2:



v	S	A	E	D	B	C
d[*v*]	0	10	8	9	10	12

Bellman-Ford in Action (step 11)

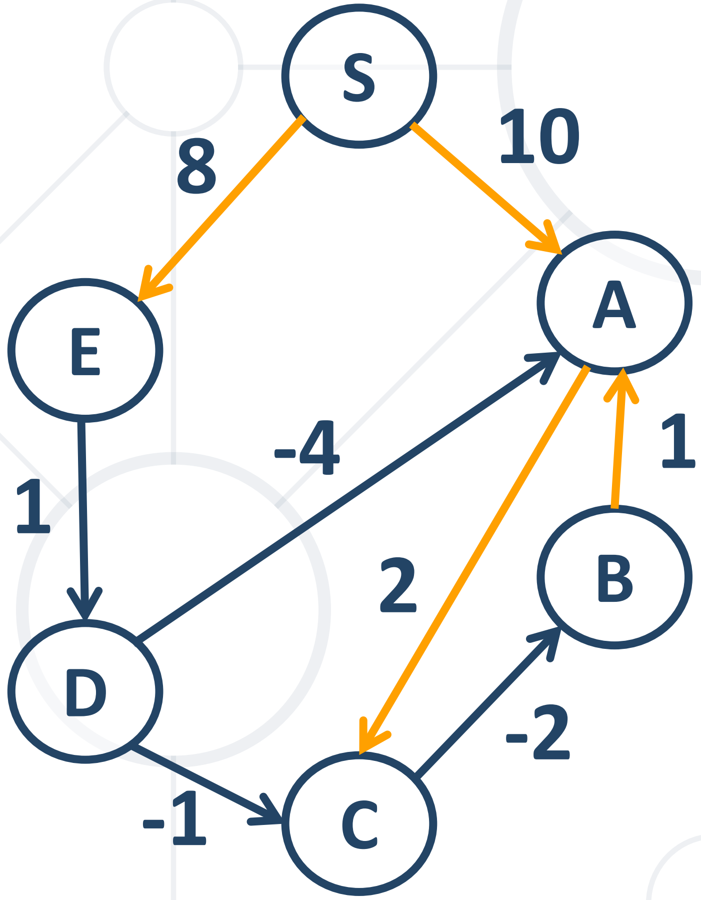
- Iteration #2:



v	S	A	E	D	B	C
d[*v*]	0	10	8	9	10	12

Bellman-Ford in Action (step 12)

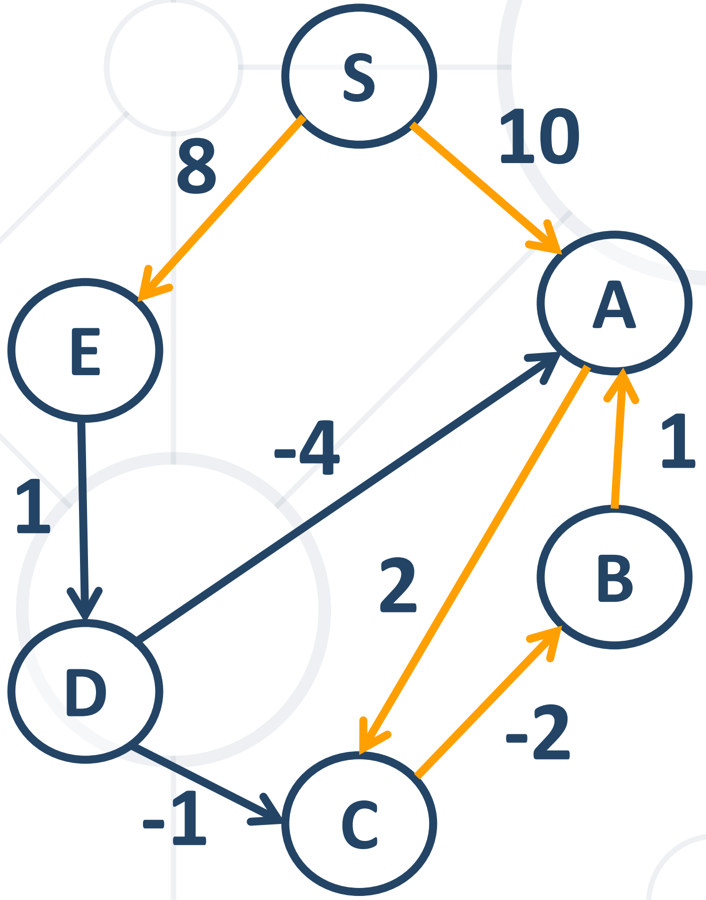
- Iteration #2:



v	S	A	E	D	B	C
d[*v*]	0	10	8	9	10	12

Bellman-Ford in Action (step 13)

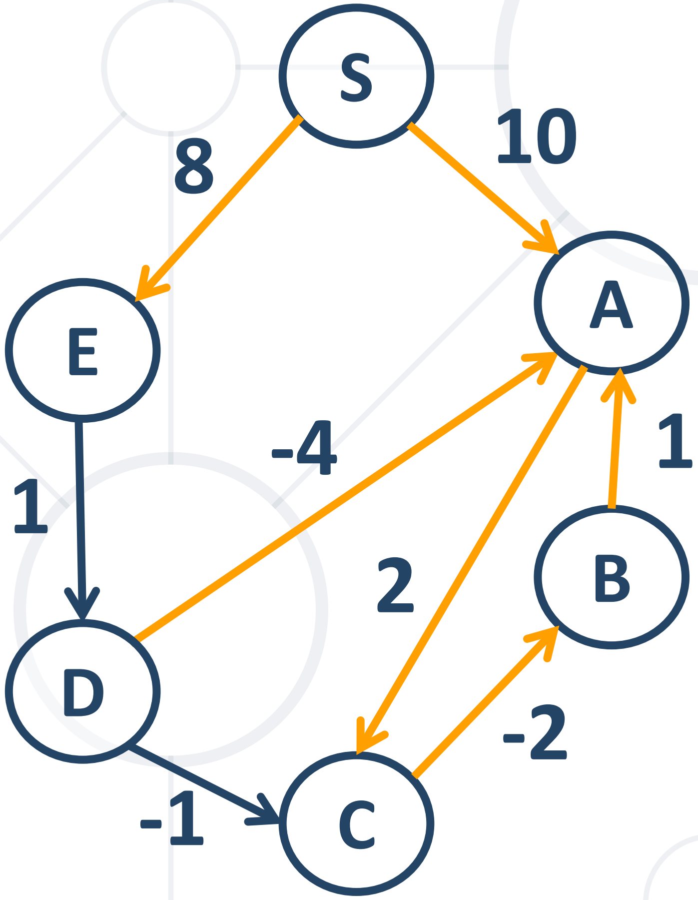
- Iteration #2:



v	S	A	E	D	B	C
d[*v*]	0	10	8	9	10	12

Bellman-Ford in Action (step 14)

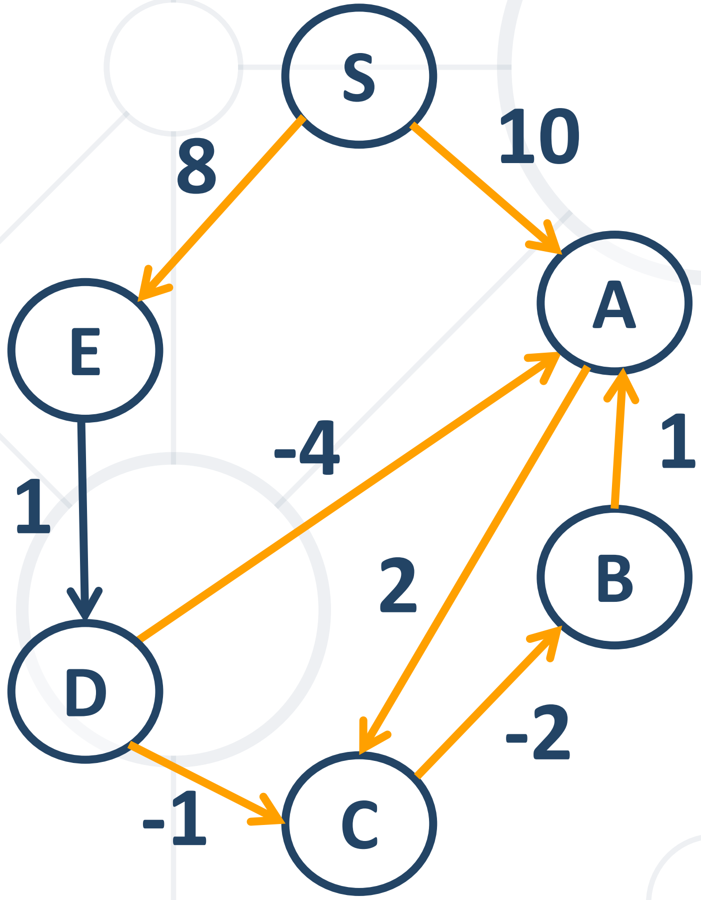
- Iteration #2:



v	S	A	E	D	B	C
d[*v*]	0	5	8	9	10	12

Bellman-Ford in Action (step 15)

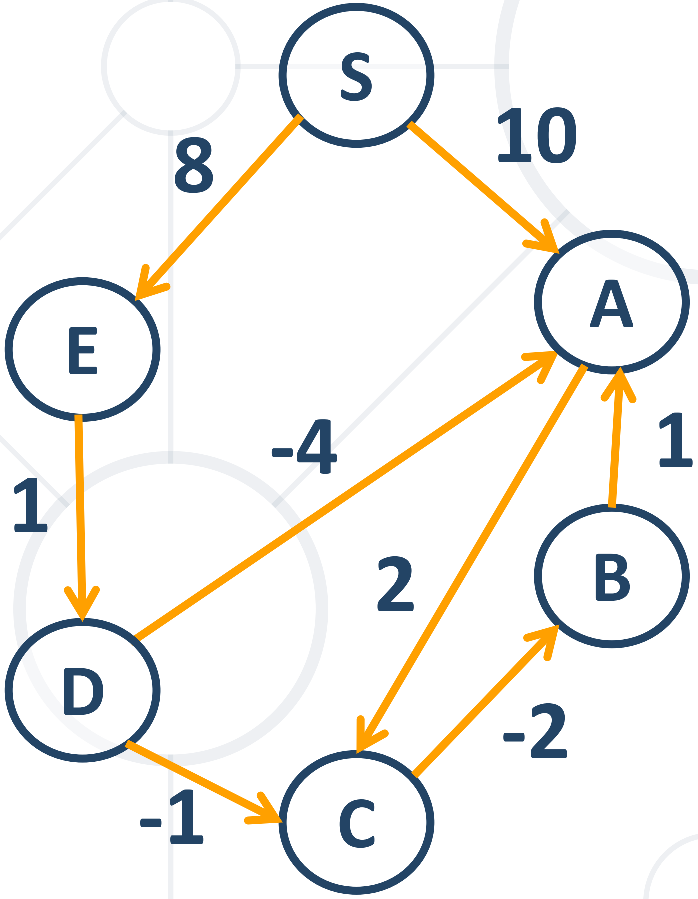
- Iteration #2:



v	S	A	E	D	B	C
d[*v*]	0	5	8	9	10	8

Bellman-Ford in Action (step 16)

- Iteration #2:

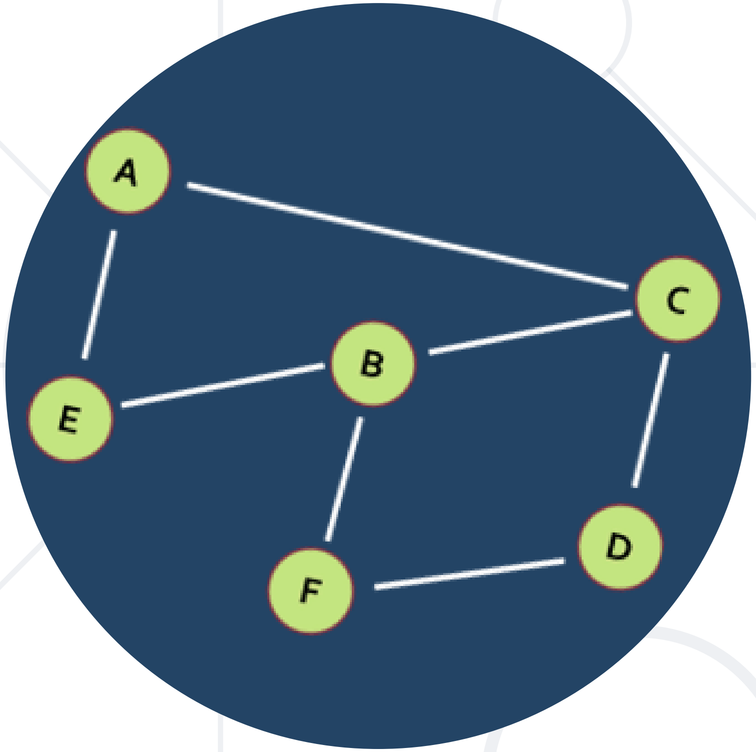


v	S	A	E	D	B	C
d[*v*]	0	5	8	9	10	8

Bellman-Ford Algorithm

- Algorithm steps pseudocode:

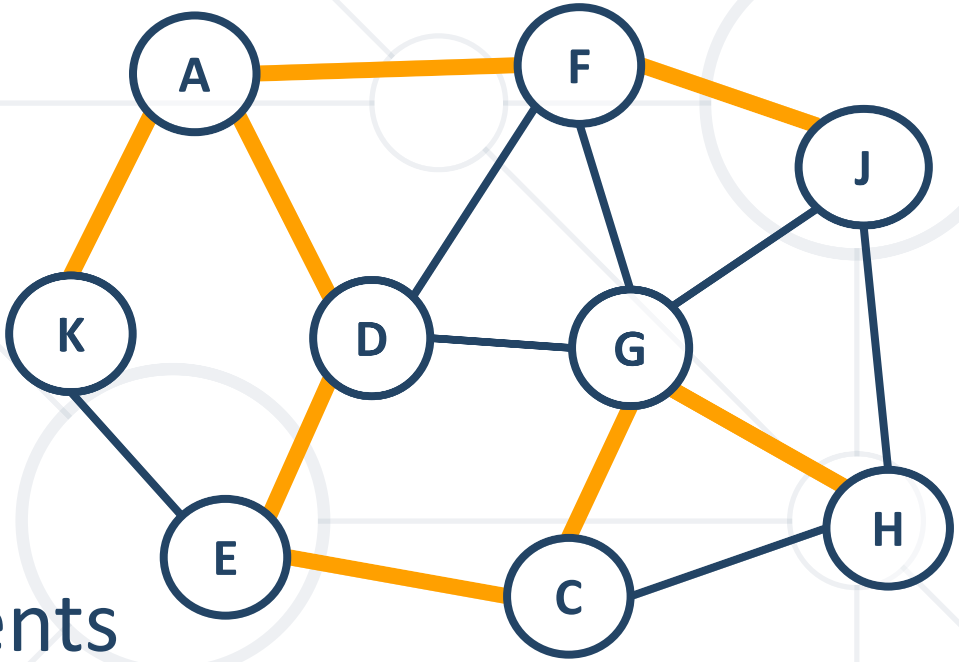
```
for v in G
   d[v] = infinity
   prev[v] = null
d[source] = 0
for vertex in G.vertices – 1
  for edge in edges
    if (d[edge.from] != infinity and
        d[edge.from] + edge.weight < d[edge.to])
      update d[edge.to]
// Run the algorithm second time if you can
// update any distance there is a negative cycle
```



Minimum Spanning Tree (MST)

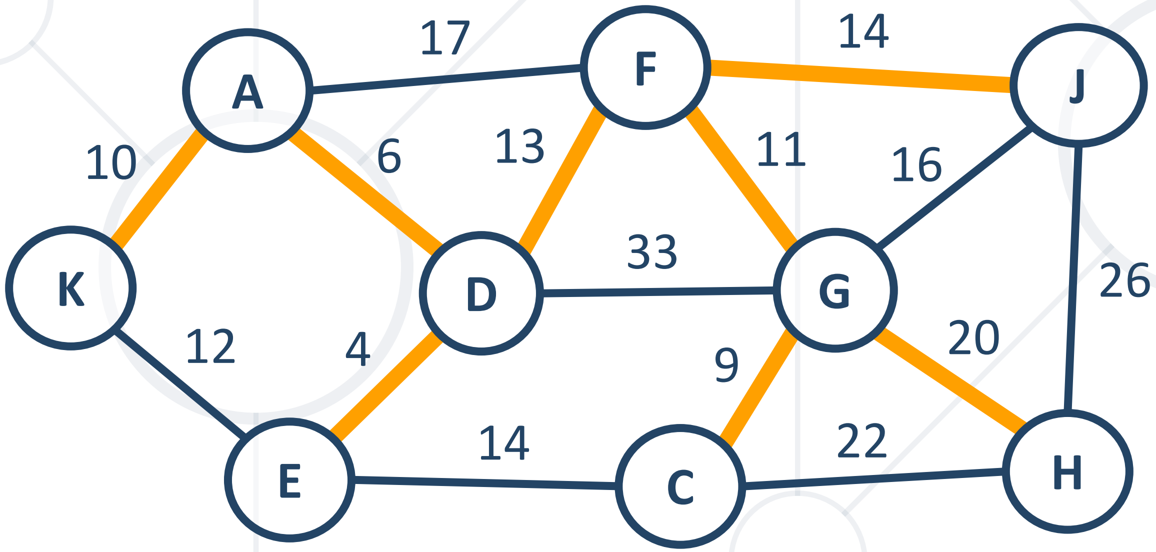
Spanning Tree

- **Spanning tree**
  - Subgraph without cycles (tree)
  - Connects all vertices together
- All connected graphs have a spanning tree
- All graphs with multiple components have **spanning forest**



Minimum Spanning Tree (MST)

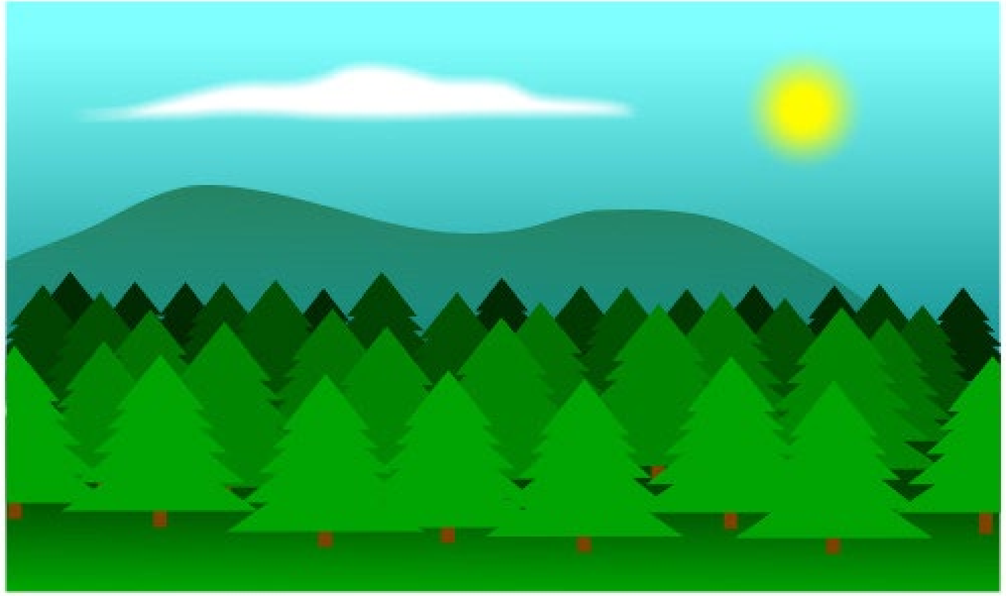
- **Minimum spanning tree** (MST)
  - Weight <= weight(all other spanning trees)
- First used in electrical networks
  - Minimal cost of wiring

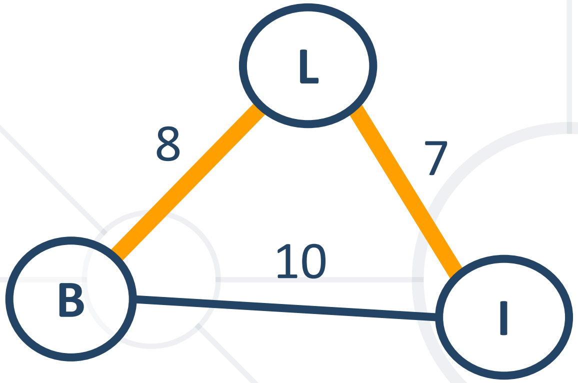


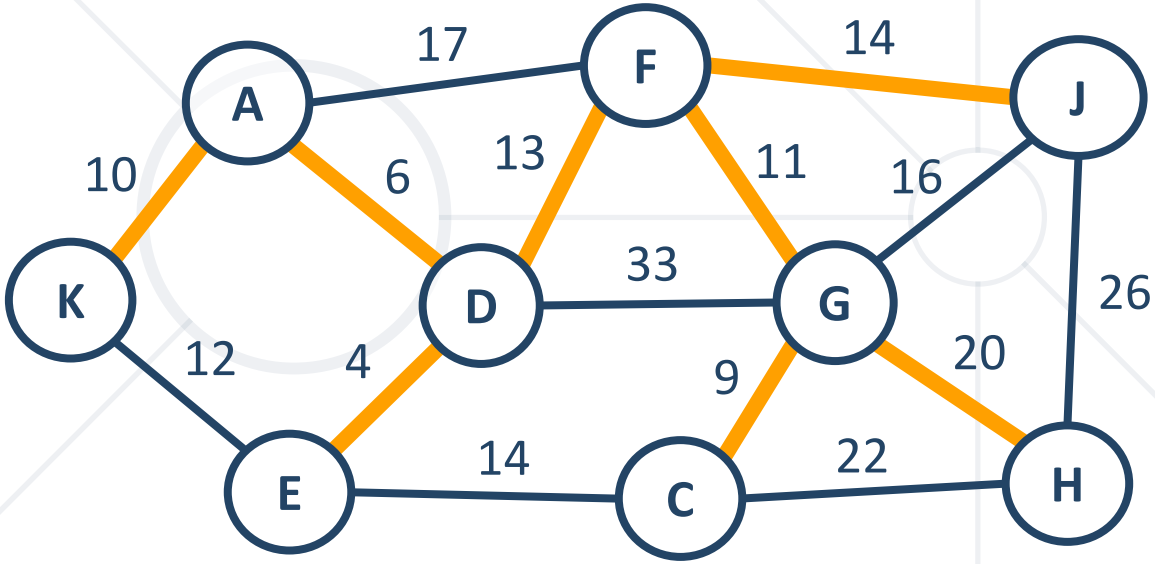
Minimum Spanning Forest (MSF)

Software University

- Minimum spanning forest
- Set of all minimum spanning trees (when the graph is not connected)



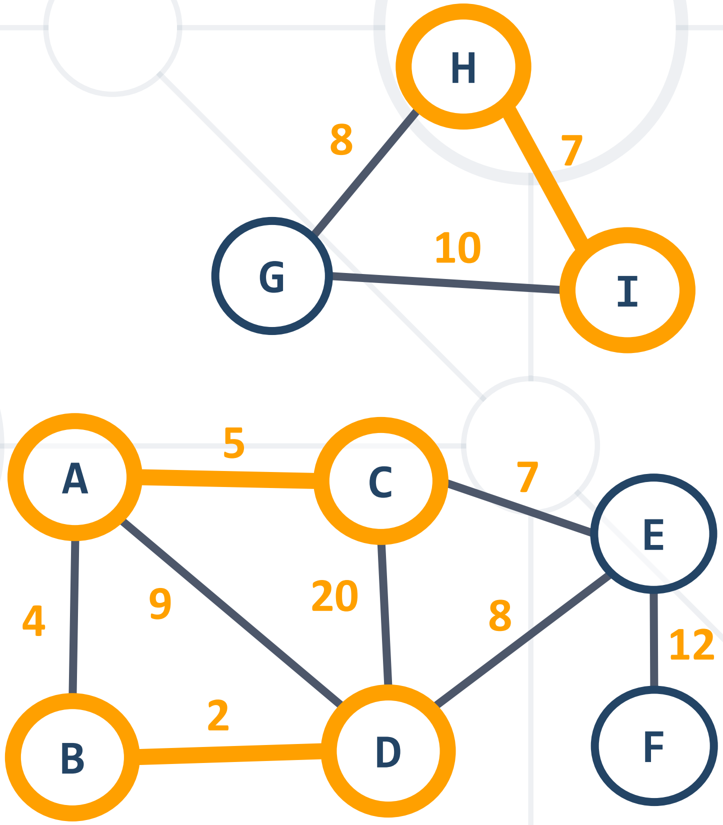




Kruskal's Algorithm

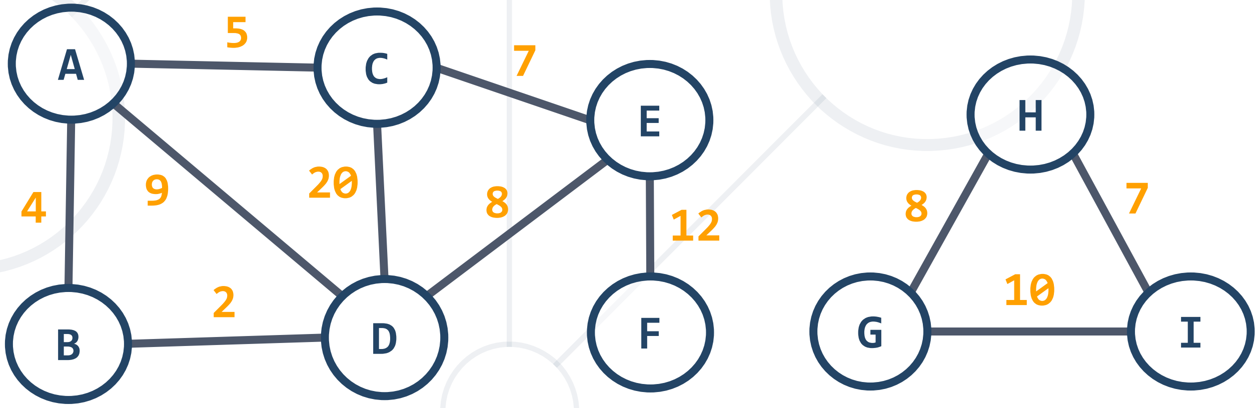
Kruskal's Algorithm

- Create a forest **F** holding all graph vertices and no edges
- Create a set **S** holding all edges in the graph
- While **S** is non-empty
  - Remove the edge ***e*** with min weight
  - If ***e*** connects two different trees
    - Add ***e*** to the forest
    - Join these two trees into a single tree
- The graph may not be connected



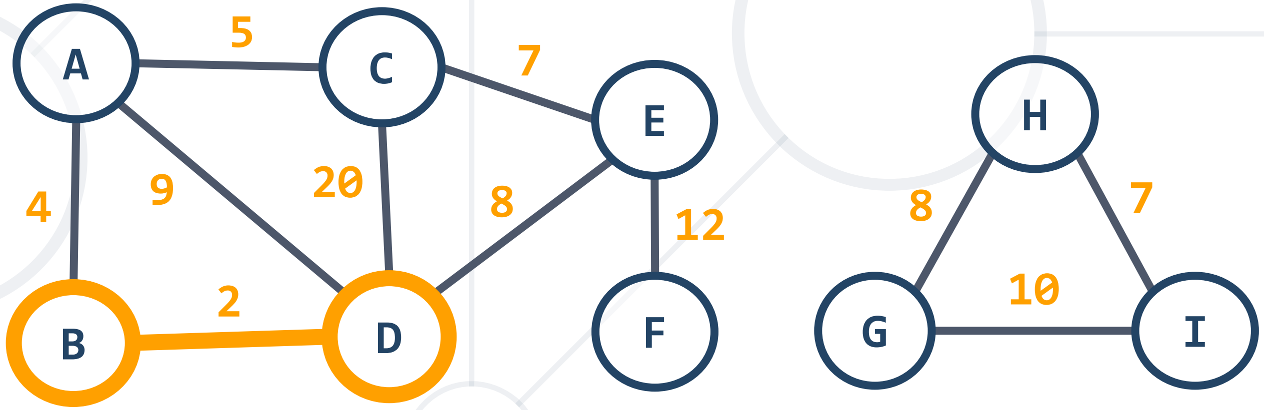
Kruskal's Algorithm – Step #1

- Start from forest holding all vertices and no edges
- **S** = all edges, ordered by weight
- **F** = { }
- **S** = {**BD**=2, **AB**=4, **AC**=5, **CE**=7, **HI**=7, **DE**=8, **GH**=8, **AD**=9, **GI**=10, **EF**=12, **CD**=20}



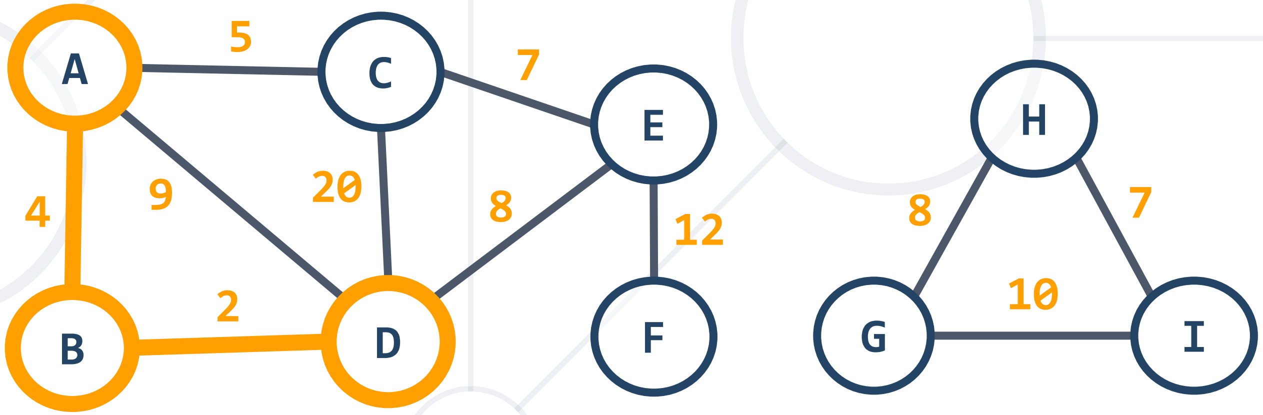
Kruskal's Algorithm – Step #2

- Take the smallest edge **BD** = 2
  - The edge **BD** connects different trees → add it to the forest
- **F** = {**BD**=2}
- **S** = {**AB**=4, **AC**=5, **CE**=7, **HI**=7, **DE**=8, **GH**=8, **AD**=9, **GI**=10, **EF**=12, **CD**=20}



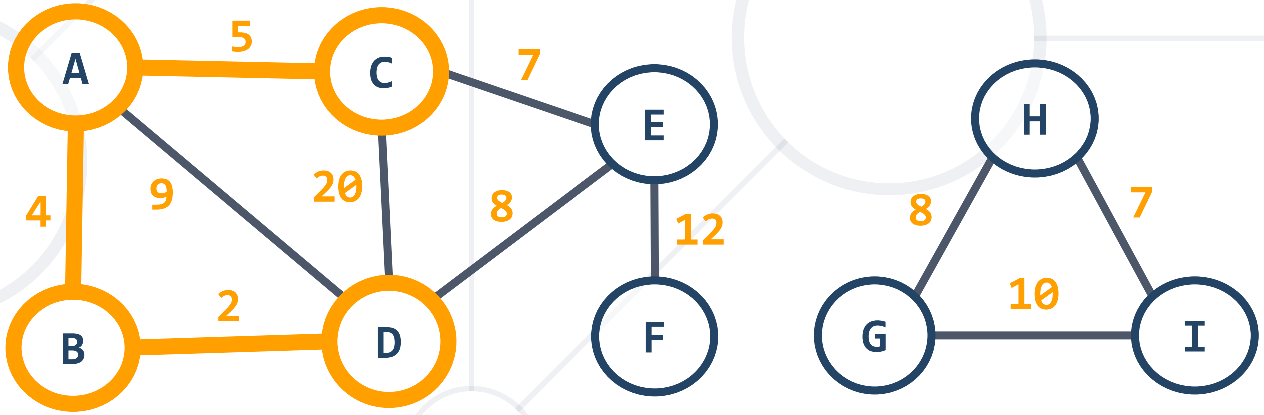
Kruskal's Algorithm – Step #3

- Take the smallest edge **AB** = 4
  - The edge **AB** connects different trees → add it to the forest
- **F** = {**BD**=2, **AB**=4}
- **S** = {**AC**=5, **CE**=7, **HI**=7, **DE**=8, **GH**=8, **AD**=9, **GI**=10, **EF**=12, **CD**=20}



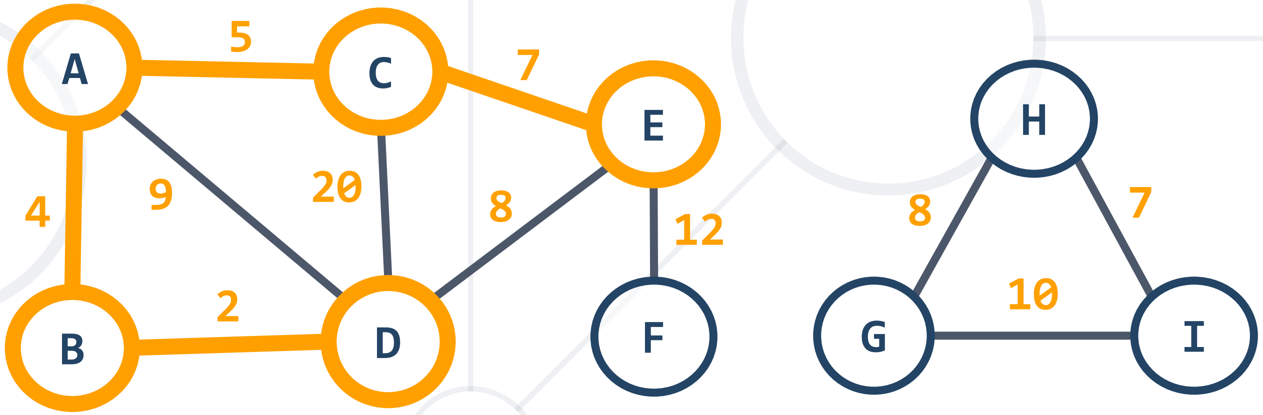
Kruskal's Algorithm – Step #4

- Take the smallest edge **AC** = 5
  - The edge **AC** connects different trees → add it to the forest
- **F** = {**BD**=2, **AB**=4, **AC**=5}
- **S** = {**CE**=7, **HI**=7, **DE**=8, **GH**=8, **AD**=9, **GI**=10, **EF**=12, **CD**=20}



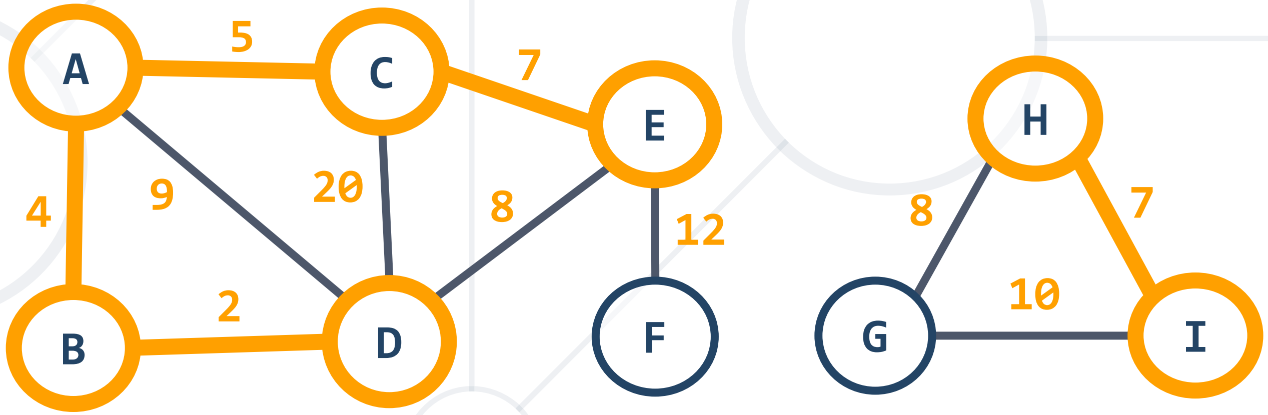
Kruskal's Algorithm – Step #5

- Take the smallest edge **CE** = 7
  - The edge **CE** connects different trees → add it to the forest
- **F** = {**BD**=2, **AB**=4, **AC**=5, **CE**=7}
- **S** = {**HI**=7, **DE**=8, **GH**=8, **AD**=9, **GI**=10, **EF**=12, **CD**=20}



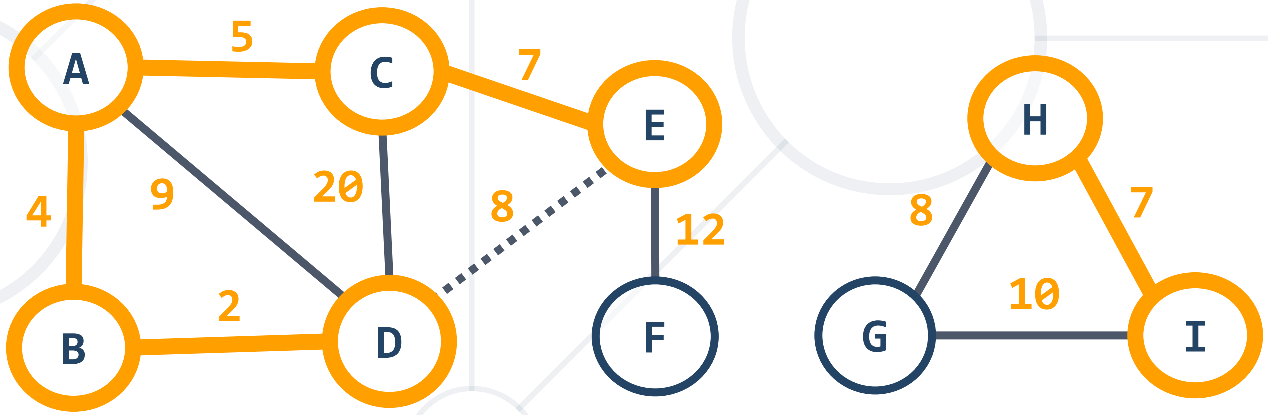
Kruskal's Algorithm – Step #6

- Take the smallest edge **HI** = 7
  - The edge **CE** connects different trees → add it to the forest
- **F** = {**BD**=2, **AB**=4, **AC**=5, **CE**=7, **HI**=7}
- **S** = {**DE**=8, **GH**=8, **AD**=9, **GI**=10, **EF**=12, **CD**=20}



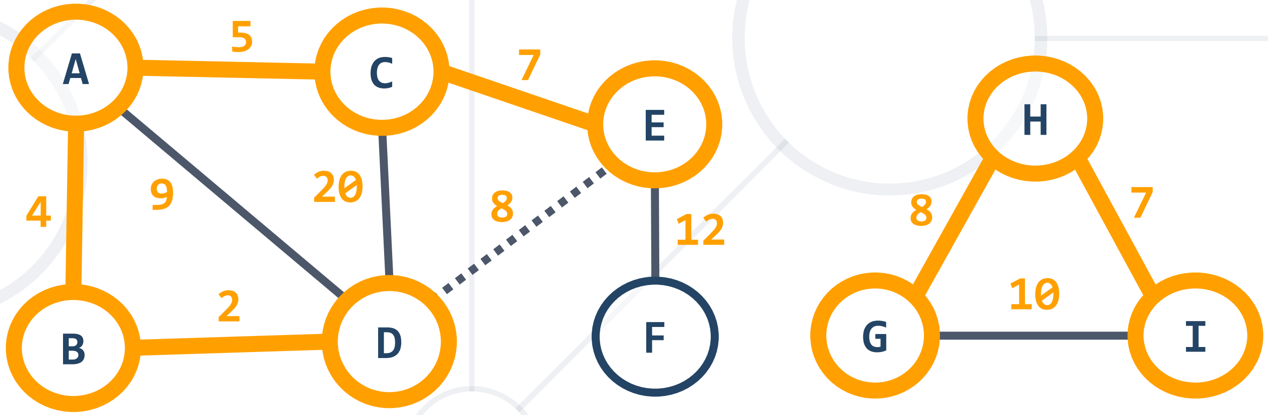
Kruskal's Algorithm – Step #7

- Take the smallest edge **DE** = 8
  - The edge **DE** causes a cycle (connects the same tree) → skip it
- **F** = {**BD**=2, **AB**=4, **AC**=5, **CE**=7, **HI**=7}
- **S** = {**GH**=8, **AD**=9, **GI**=10, **EF**=12, **CD**=20}



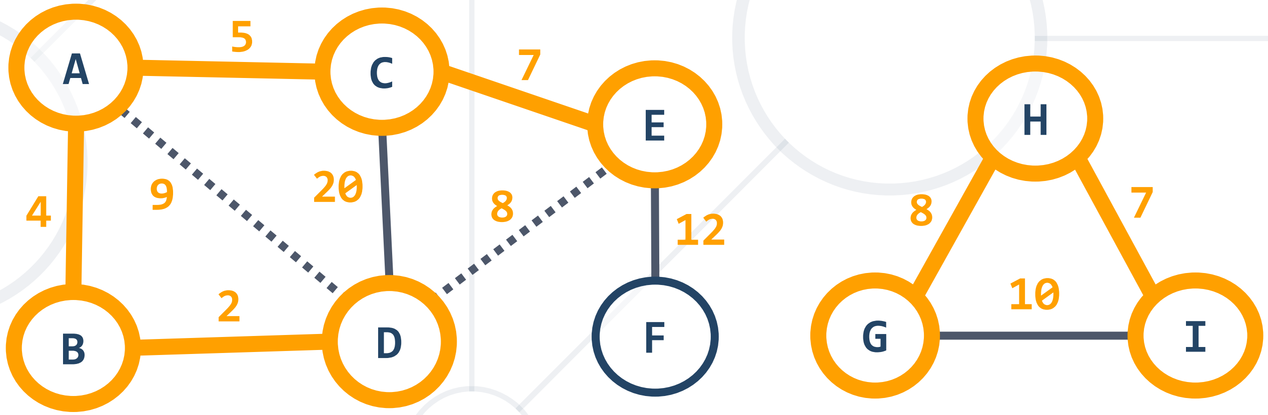
Kruskal's Algorithm – Step #8

- Take the smallest edge **GH** = 8
  - The edge **GH** connects different trees → add it to the forest
- **F** = {**BD**=2, **AB**=4, **AC**=5, **CE**=7, **HI**=7, **GH**=7}
- **S** = {**AD**=9, **GI**=10, **EF**=12, **CD**=20}



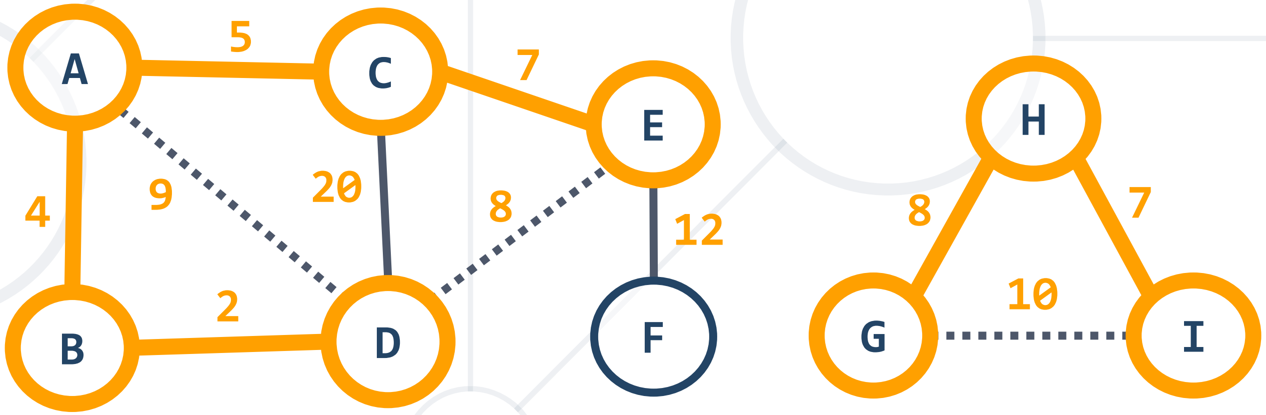
Kruskal's Algorithm – Step #9

- Take the smallest edge **AD** = 9
  - The edge **AD** causes a cycle (connects the same tree) → skip it
- **F** = {**BD**=2, **AB**=4, **AC**=5, **CE**=7, **HI**=7, **GH**=7}
- **S** = {**GI**=10, **EF**=12, **CD**=20}



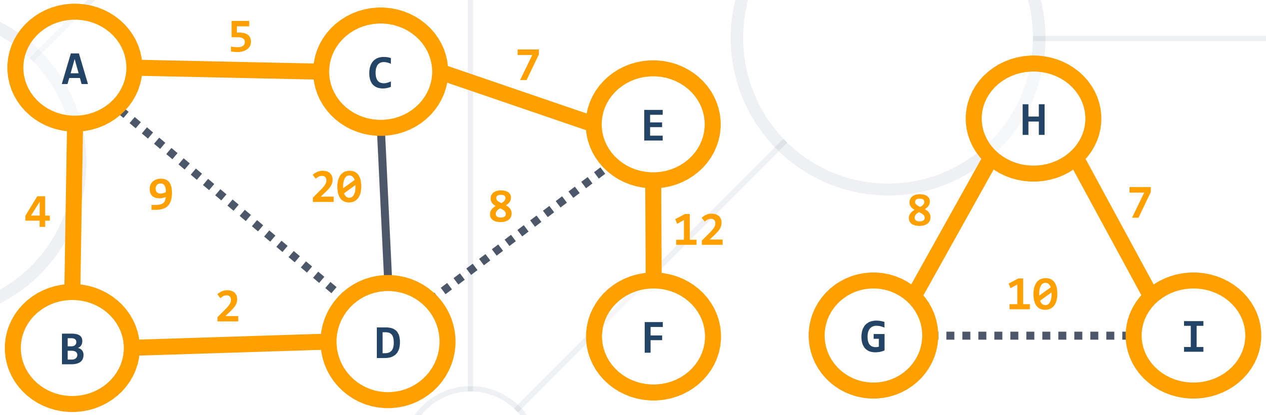
Kruskal's Algorithm – Step #10

- Take the smallest edge **GI** = 10
  - The edge **GI** causes a cycle (connects the same tree) → skip it
- **F** = {**BD**=2, **AB**=4, **AC**=5, **CE**=7, **HI**=7, **GH**=7}
- **S** = {**EF**=12, **CD**=20}



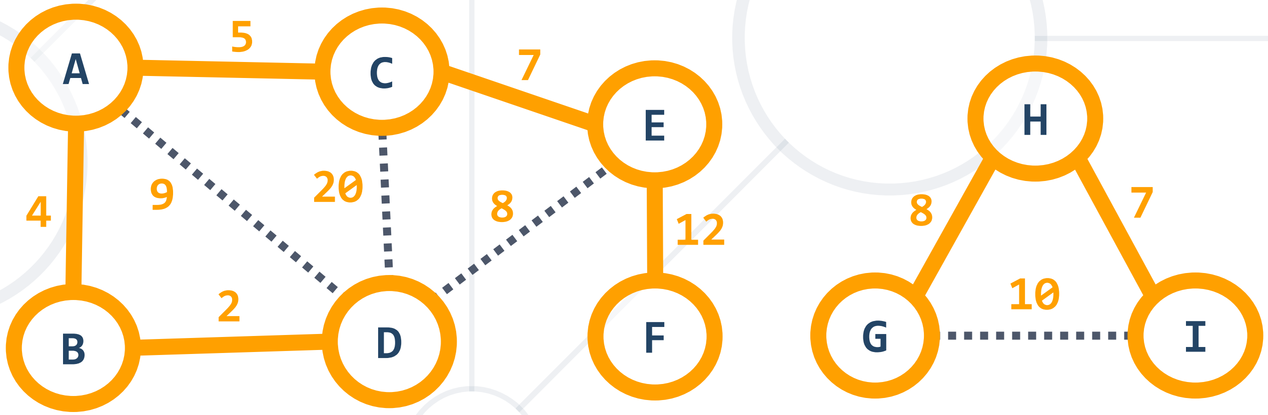
Kruskal's Algorithm – Step #11

- Take the smallest edge **EF** = 12
  - The edge **EF** connects different trees → add it to the forest
- **F** = {**BD**=2, **AB**=4, **AC**=5, **CE**=7, **HI**=7, **GH**=7, **EF**=12}
- **S** = {**CD**=20}



Kruskal's Algorithm – Step #12

- Take the smallest edge **CD** = 20
  - The edge **CD** causes a cycle (connects the same tree) → skip it
- **F** = {**BD**=2, **AB**=4, **AC**=5, **CE**=7, **HI**=7, **GH**=7, **EF**=12}
- **S** = { } → stop the algorithm



Kruskal's Algorithm – Pseudo Code

- Time complexity: **$O(|E| * \log^* |E|)$**

```
foreach v ∈ graph edges
  parent[v] = v
foreach edge {u, v} ordered by weight(u, v)
  rootU = findRoot(u)
  rootV = findRoot(v)
  if rootU ≠ rootV
    print edge {u, v}
    parent[rootU] = rootV
findRoot(node)
  while (parent[node] ≠ node)
    node = parent[node]
  return node
```

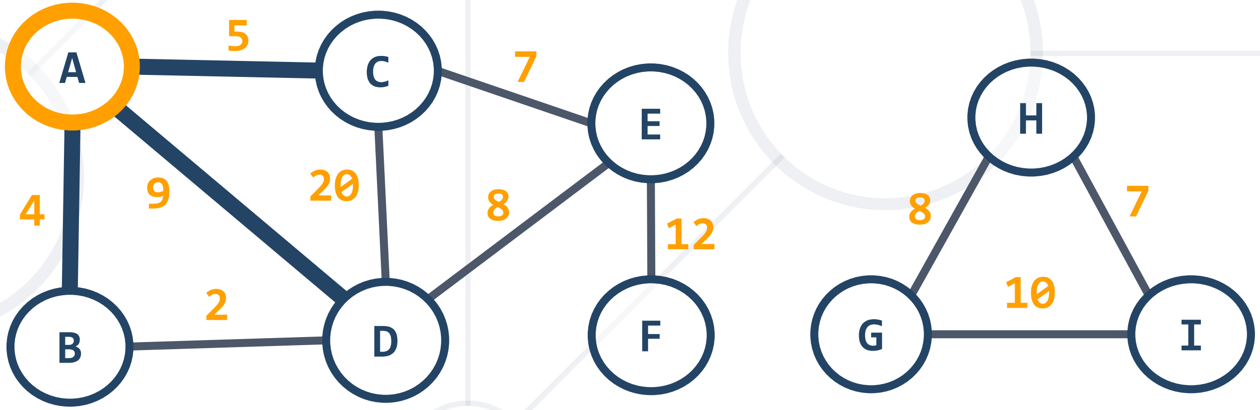
Prim's Algorithm

Prim's Algorithm

- Given a graph **G(V, E)** find the minimum spanning forest **T(V', E')**
- Attach to the tree **T** the starting node
- While smallest edge exists
  - Attach to **T** the smallest possible edge from **G** without creating a cycle in **T**
    - Use the smallest edge **(u, v)**, such that **u** ∈ **T** and **v** ∉ **T**
- Start the Prim's algorithm for each node from **G**

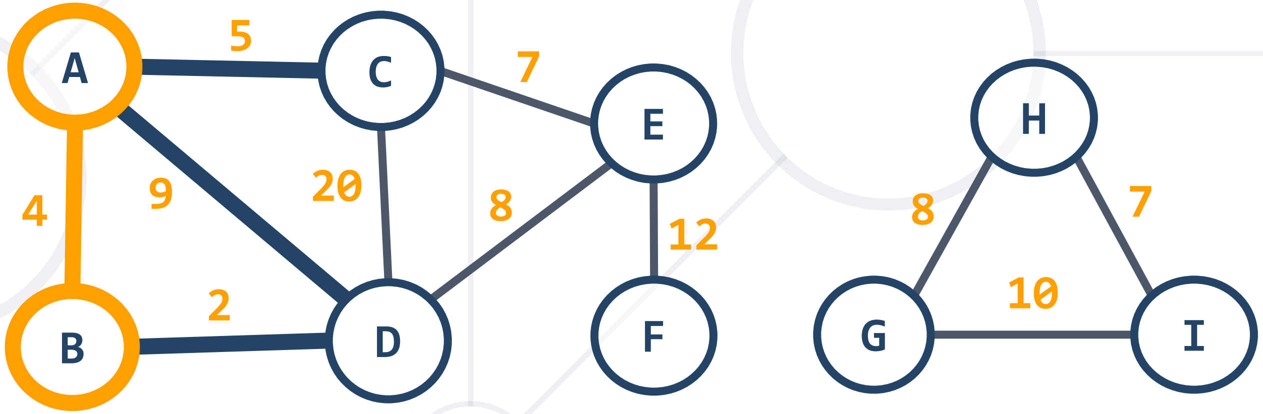
Prim's Algorithm – Step #1

- **Start** from the initial node **A**
- **Enqueue** all edges from **A** to other graph nodes: **AB**, **AC**, **AD**
- **Spanning tree** = {A}
- **Priority queue** = {**AB** = 4}, {**AC** = 5}, {**AD** = 9}



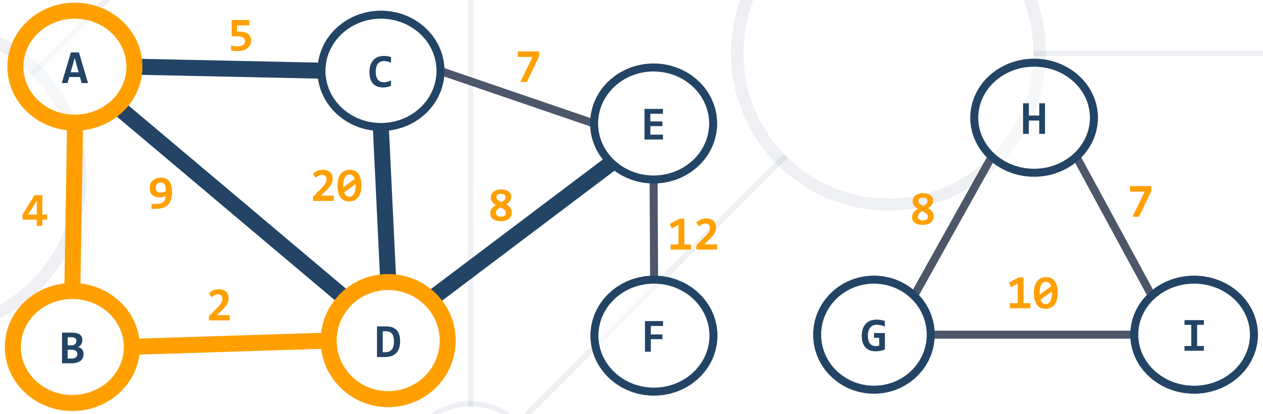
Prim's Algorithm – Step #2

- **Dequeue the shortest edge** {**AB** = 4} and add it to the tree
- **Enqueue** all edges from **B** to other graph nodes: **BD**
- **Spanning tree** = {**AB** = 4}
- **Priority queue** = {**BD** = 2}, {**AC** = 5}, {**AD** = 9}



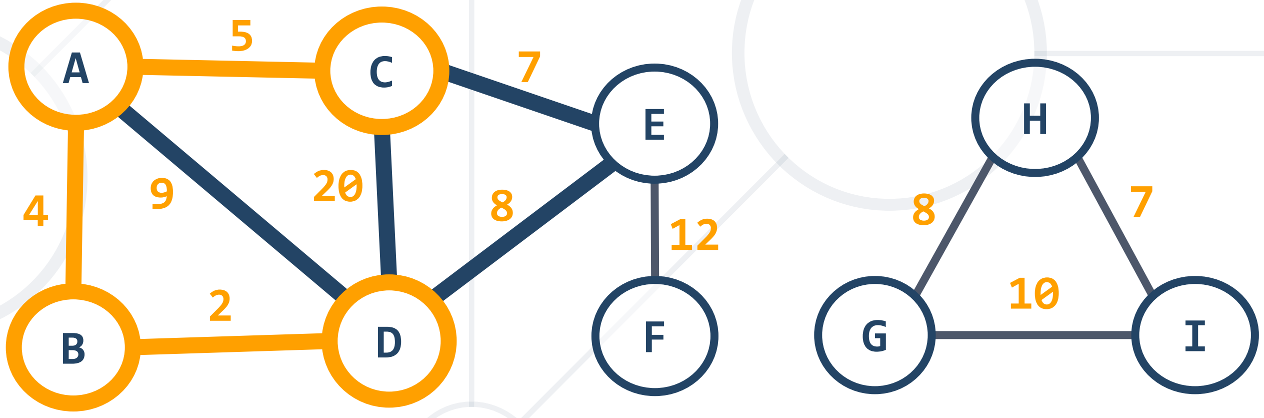
Prim's Algorithm – Step #3

- **Dequeue the shortest edge** {**BD** = 2} and add it to the tree
- **Enqueue** all edges from **D** to other graph nodes: **DC**, **DE**
- **Spanning tree** = {**AB** = 4}, {**BD** = 2}
- **Priority queue** = {**AC** = 5}, {**DE** = 8}, {**AD** = 9}, {**CD** = 20}



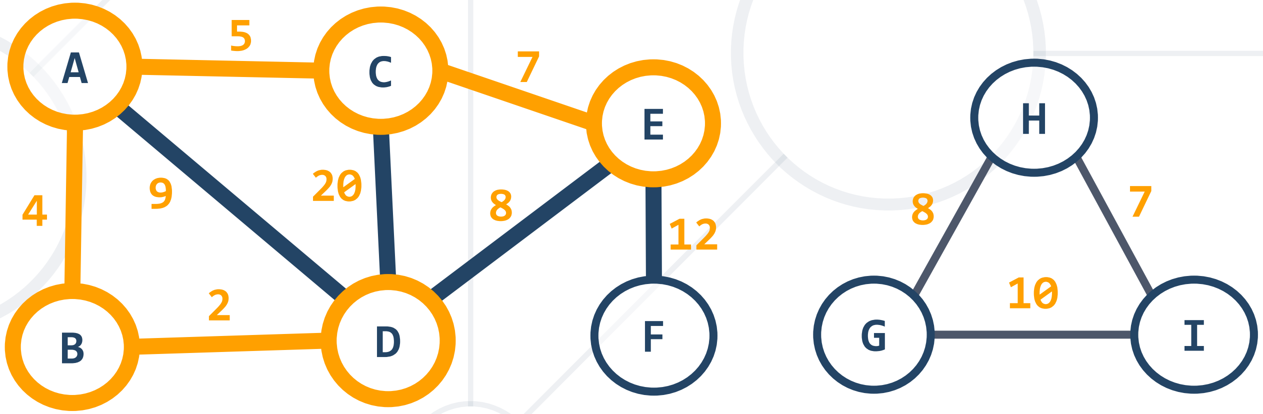
Prim's Algorithm – Step #4

- **Dequeue the shortest edge** {**AC** = 5} and add it to the tree
- **Enqueue** all edges from **C** to other graph nodes: **CE**
- **Spanning tree** = {**AB** = 4}, {**BD** = 2}, {**AC** = 5}
- **Priority queue** = {**CE** = 7}, {**DE** = 8}, {**AD** = 9}, {**CD** = 20}



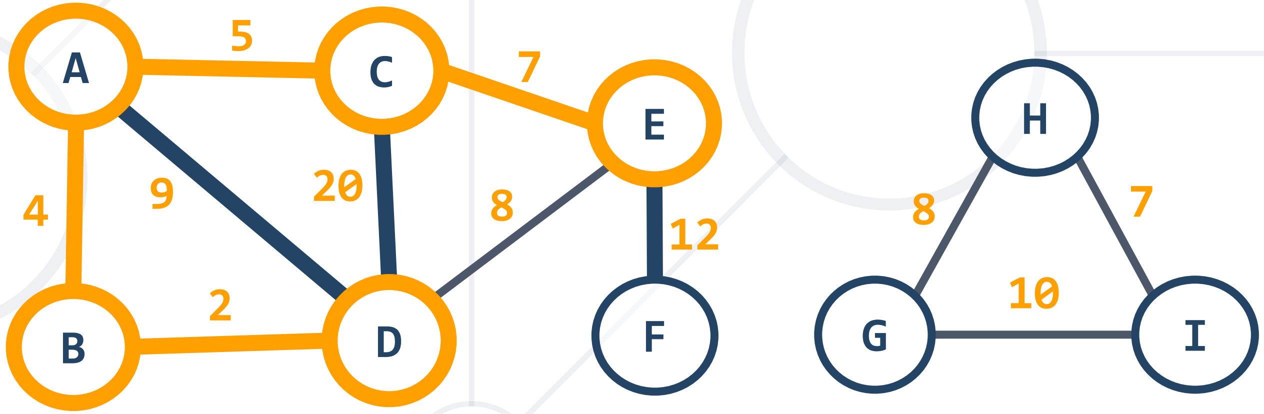
Prim's Algorithm – Step #5

- **Dequeue the shortest edge** {**CE** = 7} and add it to the tree
- **Enqueue** all edges from **E** to other graph nodes: **EF**
- **Spanning tree** = {**AB** = 4}, {**BD** = 2}, {**AC** = 5}, {**CE** = 7}
- **Priority queue** = {**DE** = 8}, {**AD** = 9}, {**EF** = 12}, {**CD** = 20}



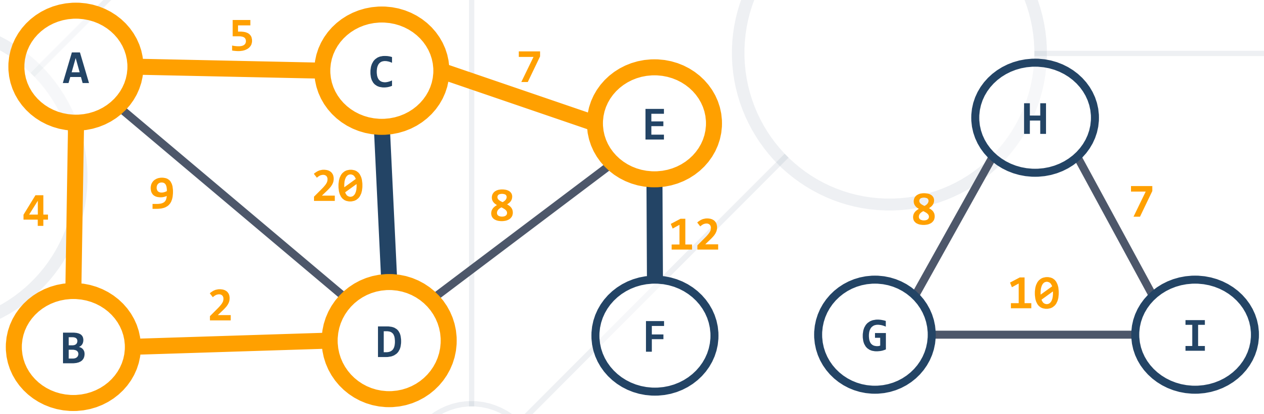
Prim's Algorithm – Step #6

- **Dequeue the shortest edge** {**DE** = 8}
  - It would create a loop in the spanning tree → skip it
- **Spanning tree** = {**AB** = 4}, {**BD** = 2}, {**AC** = 5}, {**CE** = 7}
- **Priority queue** = {**AD** = 9}, {**EF** = 12}, {**CD** = 20}



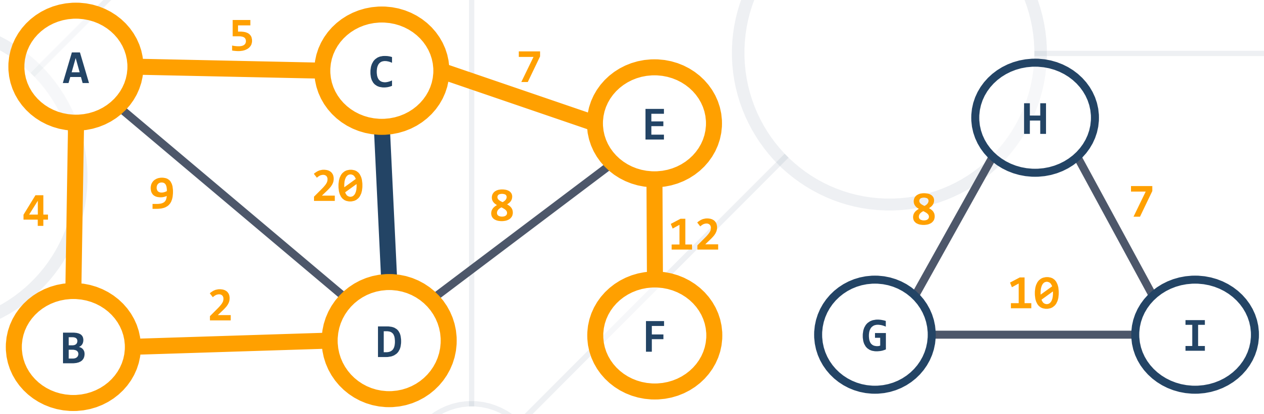
Prim's Algorithm – Step #7

- **Dequeue the shortest edge** {**AD** = 9}
  - It would create a loop in the spanning tree → skip it
- **Spanning tree** = {**AB** = 4}, {**BD** = 2}, {**AC** = 5}, {**CE** = 7}
- **Priority queue** = {**EF** = 12}, {**CD** = 20}



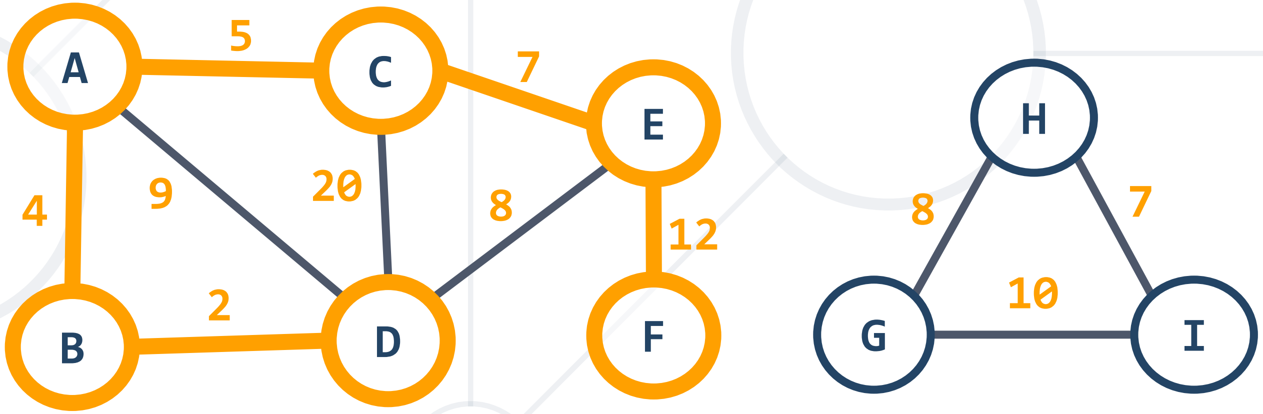
Prim's Algorithm – Step #8

- **Dequeue the shortest edge** {**EF** = 12} and add it to the tree
- **Enqueue** all edges from **F** to other graph nodes: no such edges
- **Spanning tree** = {**AB** = 4}, {**BD** = 2}, {**AC** = 5}, {**CE** = 7}, {**EF** = 12}
- **Priority queue** = {**CD** = 20}



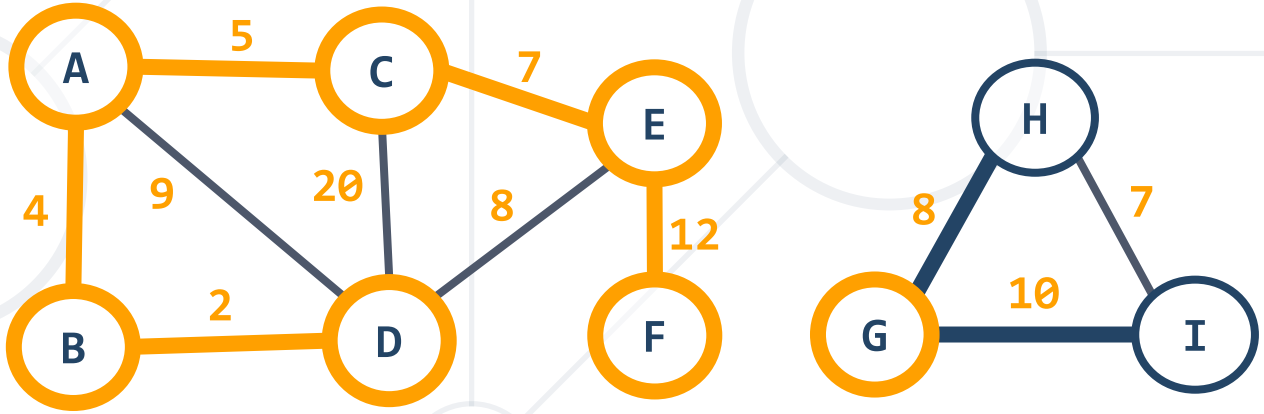
Prim's Algorithm – Step #9

- **Dequeue the shortest edge** {**CD** = 20}
  - It would create a loop in the spanning tree → skip it
- **Spanning tree** = {**AB** = 4}, {**BD** = 2}, {**AC** = 5}, {**CE** = 7}, {**EF** = 12}
- **Priority queue** = { } → stop the algorithm



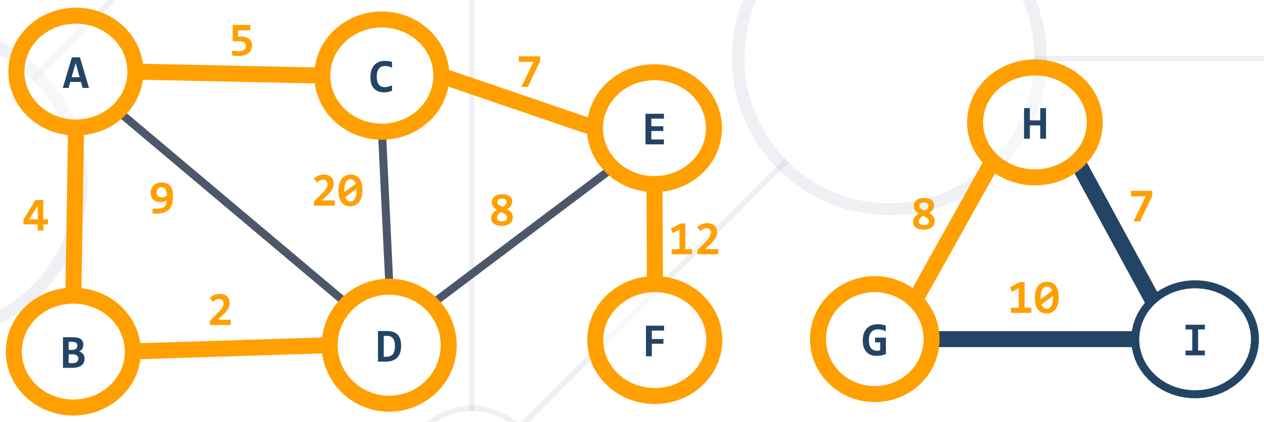
Prim's Algorithm – Step #10

- **Start** from the initial node **G**
- **Enqueue** all edges from **G** to other graph nodes: **GH**, **GI**
- **Spanning tree** = {**AB**, **BD**, **AC**, **CE**, **EF**}, {**G**}
- **Priority queue** = {**GH** = 8}, {**GI** = 10}



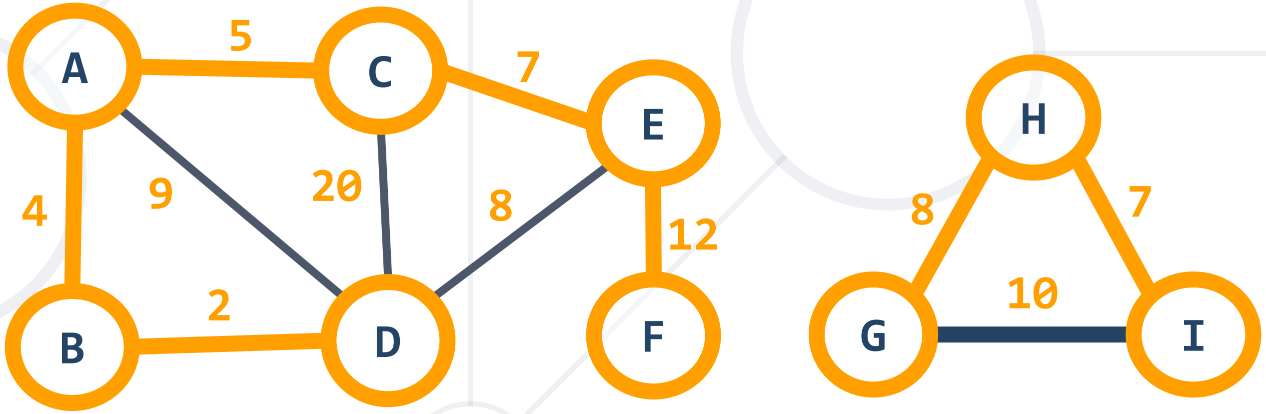
Prim's Algorithm – Step #11

- **Dequeue the shortest edge** {**GH** = 8} and add it to the tree
- **Enqueue** all edges from **H** to other graph nodes: **HI**
- **Spanning tree** = {**AB**, **BD**, **AC**, **CE**, **EF**}, {**GH** = 8}
- **Priority queue** = {**HI** = 7}, {**GI** = 10}



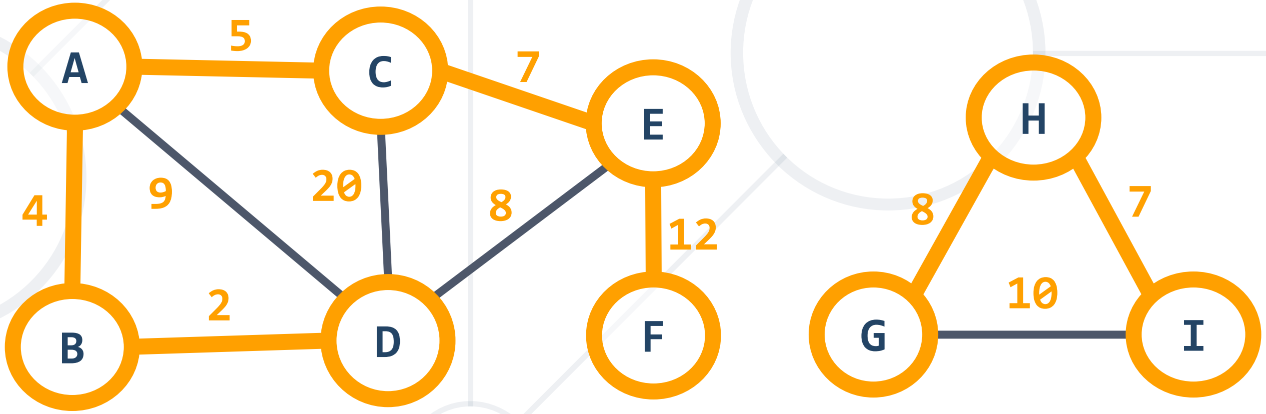
Prim's Algorithm – Step #12

- **Dequeue the shortest edge** {**HI** = 7} and add it to the tree
- **Enqueue** all edges from **I** to other graph nodes: no such edges
- **Spanning tree** = {**AB**, **BD**, **AC**, **CE**, **EF**}, {**GH** = 8}, {**HI** = 7}
- **Priority queue** = {**GI** = 10}



Prim's Algorithm – Step #13

- **Dequeue the shortest edge** {GI = 7} and add it to the tree
  - It would create a loop in the spanning tree → skip it
- **Spanning tree** = {AB, BD, AC, CE, EF}, {GH = 8}, {HI = 7}
- **Priority queue** = { } → stop the algorithm



Prim's Algorithm (with Priority Queue)

Time complexity: O(|*E*| * log |*E*|)

```
spanningTreeNodes = ∅
foreach (v ϵ graphVertices)
  if (v ∉ spanningTreeNodes)
    prim(v)
prim(startNode)
  spanningTreeNodes -> startNode
  var priorityQueue = ∅
  priorityQueue -> childEdges(startNode)
  while (priorityQueue is not empty)
    smallestEdge = priorityQueue.ExtractMin()
    if (smallestEdge connects tree node with non-tree node)
      print smallestEdge
      spanningTreeNodes -> smallestEdge.nonTreeNode
      priorityQueue -> childEdges(smallestEdge.nonTreeNode
```

Summary

- Shortest paths in a graph:
  - BFS in Unweighted Graph
  - Dijkstra's algorithm – finds the shortest path from a single source
  - Bellman ford's algorithm – finds the shortest path in graph with negative weights
- Minimum spanning tree (MST)
  - Solved by Prim's and Kruskal's algorithms

Software University

Questions?

SoftUni

Software University | SoftUni Creative | SoftUni Digital | SoftUni Foundation | SoftUni Kids | Finance Academy

© SoftUni – https://about.softuni.bg. Copyrighted document. Unauthorized copy, reproduction or use is not permitted.

SoftUni Diamond Partners

Software University

SUPER HOSTING .BG

Coca-Cola HBC Bulgaria

POKERSTARS
POKER | CASINO | SPORTS
a Flutter International brand

INDEAVR
Serving the high achievers

AMBITIONED

DRAFT KINGS

SOFTWARE GROUP

createX

Postbank
Решения за твоето утре

BOSCH

DXC TECHNOLOGY

SmartIT

Educational Partners

CODE IT UP

License

- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – https://about.softuni.bg/
- © Software University – https://softuni.bg

Trainings @ Software University (SoftUni)

Software University

- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg, about.softuni.bg
- Software University Foundation
  - softuni.foundation
- Software University @ Facebook
  - facebook.com/SoftwareUniversity
- Software University Forums
  - forum.softuni.bg

Software University

FORUM