# CS2040C: Data Structures and Algorithms

## Single Source Shortest Paths (more special cases)

# Outline

SSSP for special cases and the algorithms that are applicable and can run faster for these cases

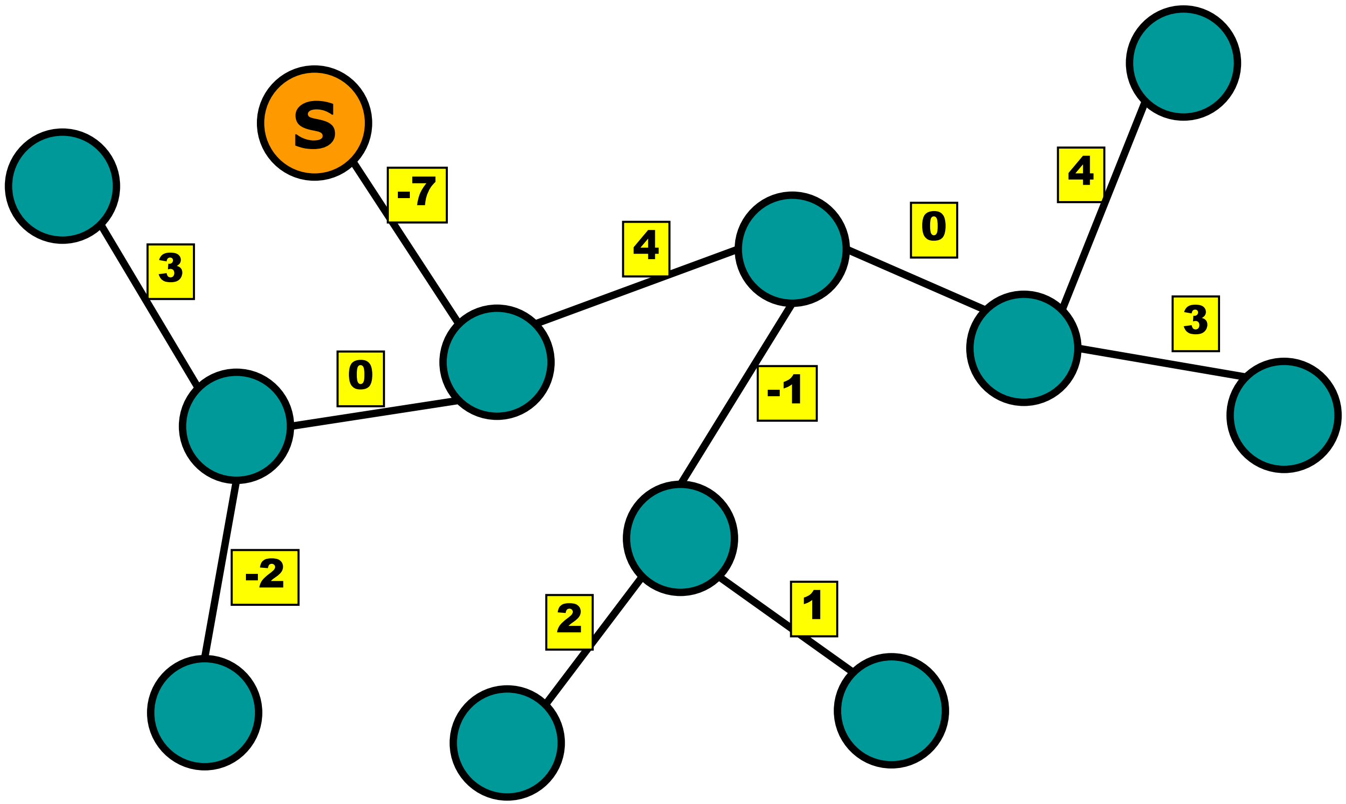
- Using BFS/DFS on Trees
- Dijkstra's algorithm for graphs with no negative weights
- Modified Dijkstra's algorithm for graphs with negative weights
- Dynamic programming for DAGs

# Special Cases

We have already covered the first two cases in the previous lecture

| Condition | Algorithm | Time Complexity |
|---|---|---|
| No Negative Weight Cycles | Bellman-Ford Algorithm | $O(VE)$ |
| On Unweighted Graph (or equal weights) | BFS | $O(V+E)$ |
| No Negative Weights | Dijkstra's Algorithm | |
| Negative weights | Modified Dijkstra's Algorithm | |
| On Tree | BFS / DFS | |
| On DAG | Dynamic Programming (one-pass Bellman-Ford) | |

# **Special Case:** Undirected, Weighted Tree



# Trees (redefined)

What is an (undirected) tree?

- A graph with no cycles is an (undirected) tree

What is a *rooted* tree?

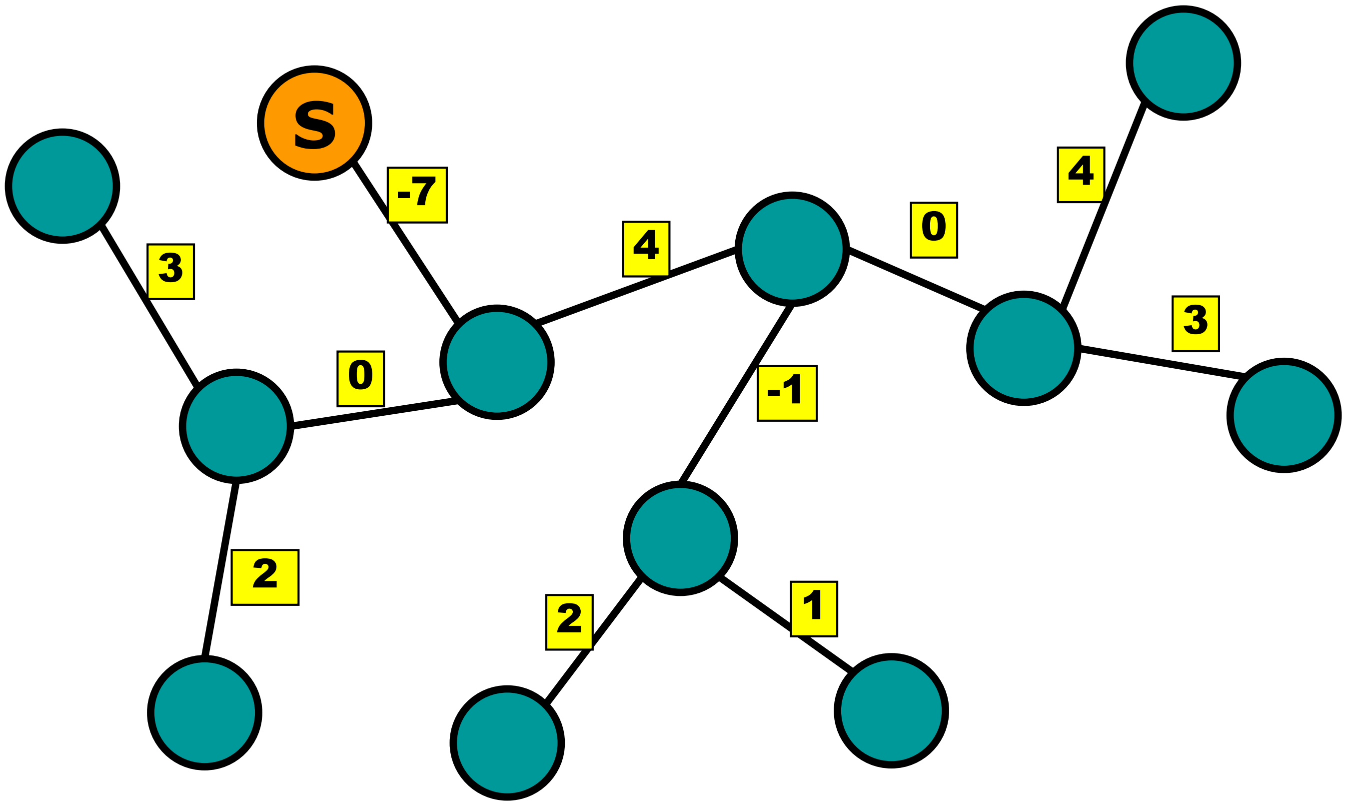
- A tree with a special designated root node

Our previous (recursive) definition of a *tree*:

- A node with zero, one, or more sub-trees
- a *rooted* tree

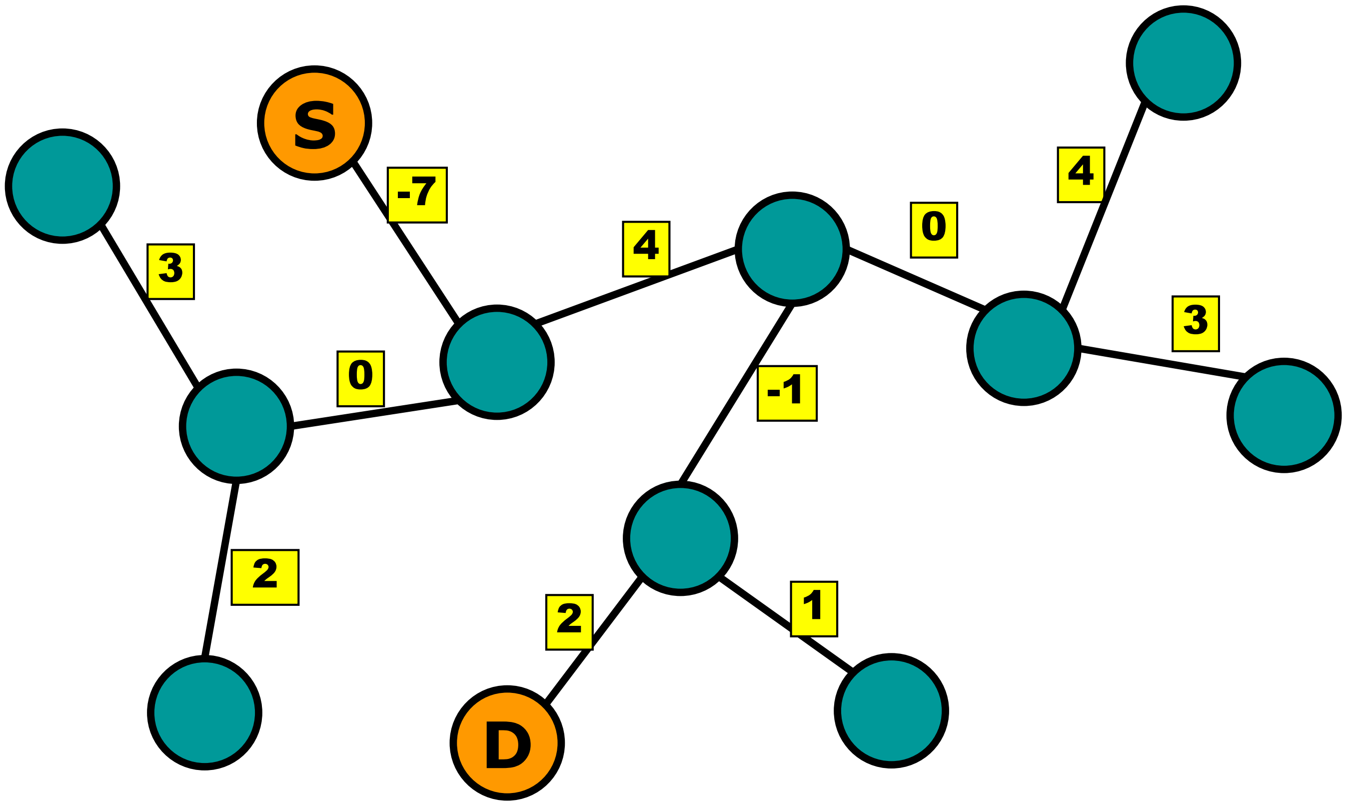
# Undirected Weighted Tree

**Assume you can only cross an edge once on your path.**



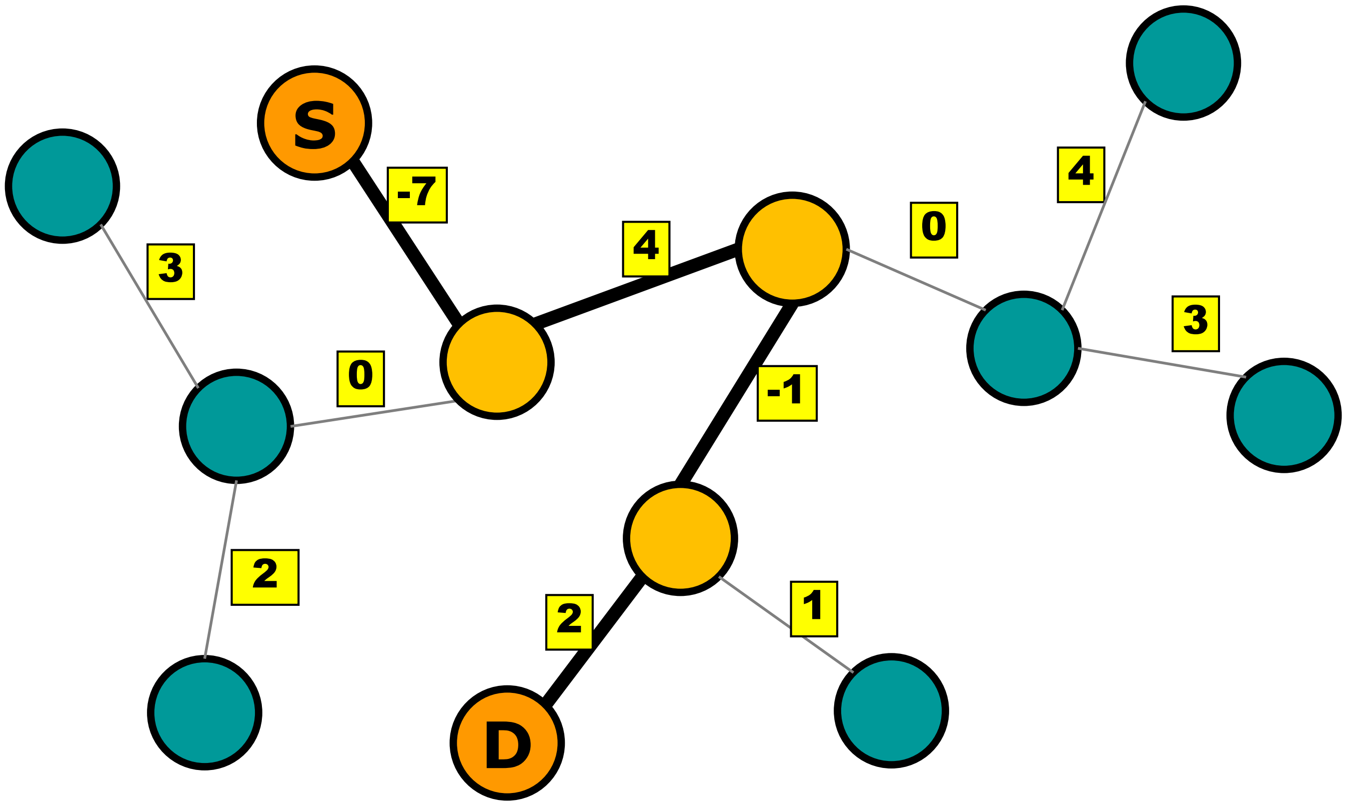
# Undirected Weighted Tree

**how many ways to get from S to D? (assume no backpedalling)**



# Undirected Weighted Tree

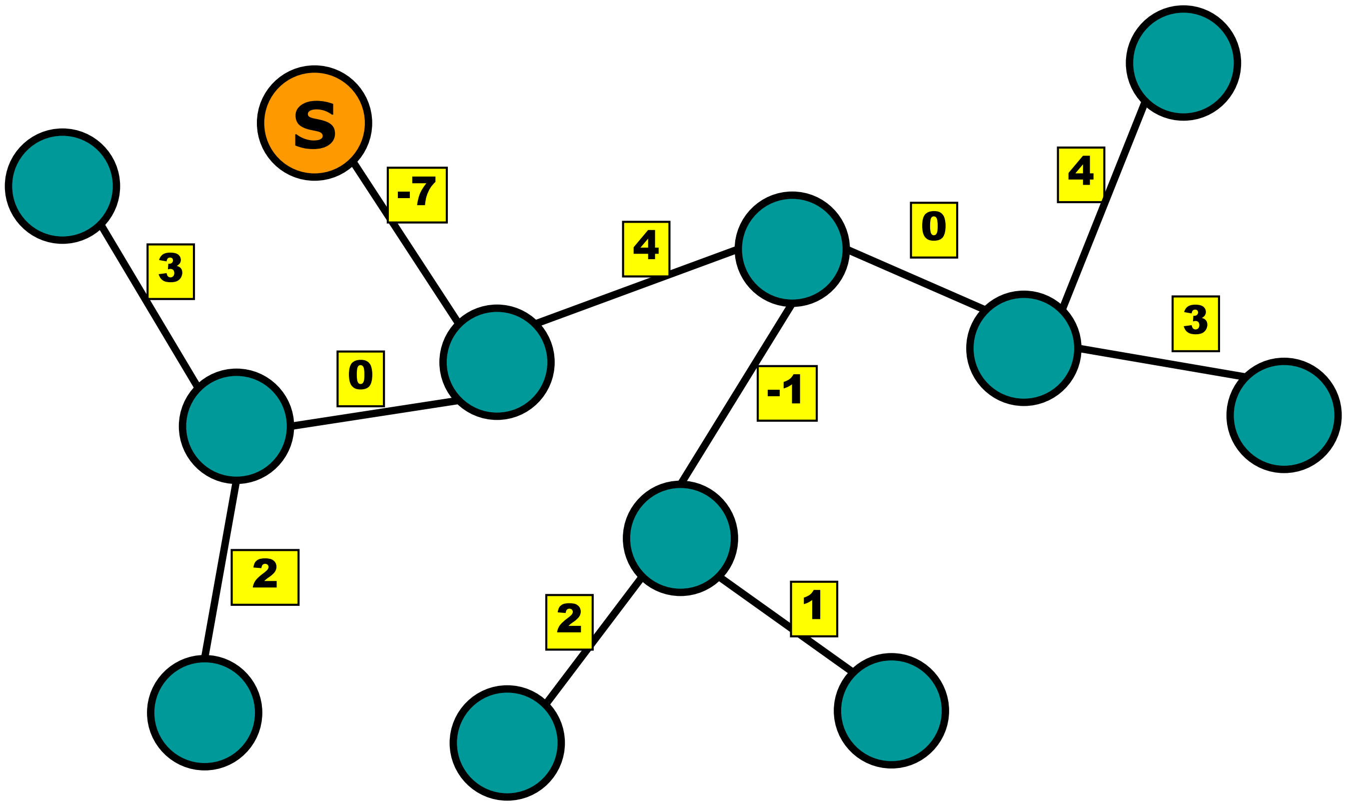
**Just 1 way! It's a tree!**



# Tree: source-to-all

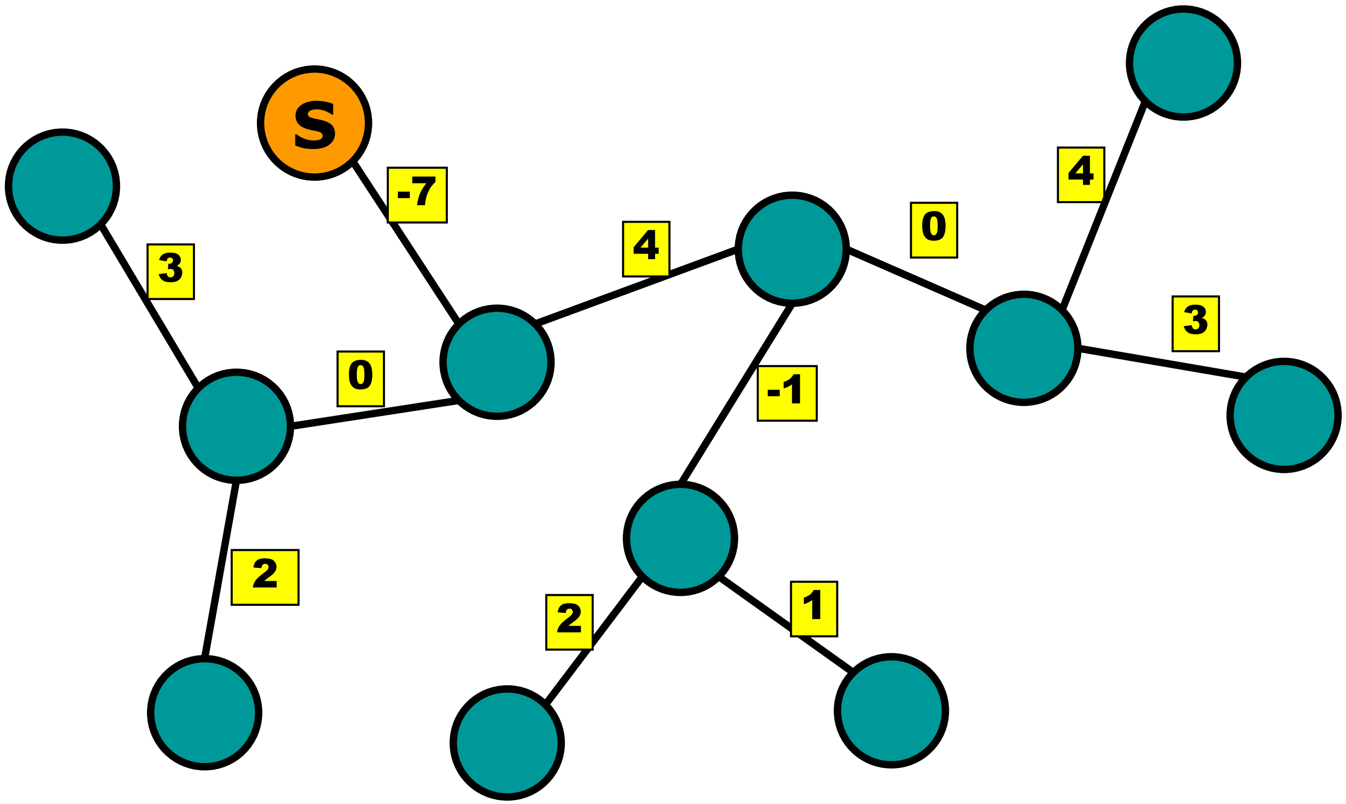
**In what order should we relax the nodes?**

**Use DFS or BFS**



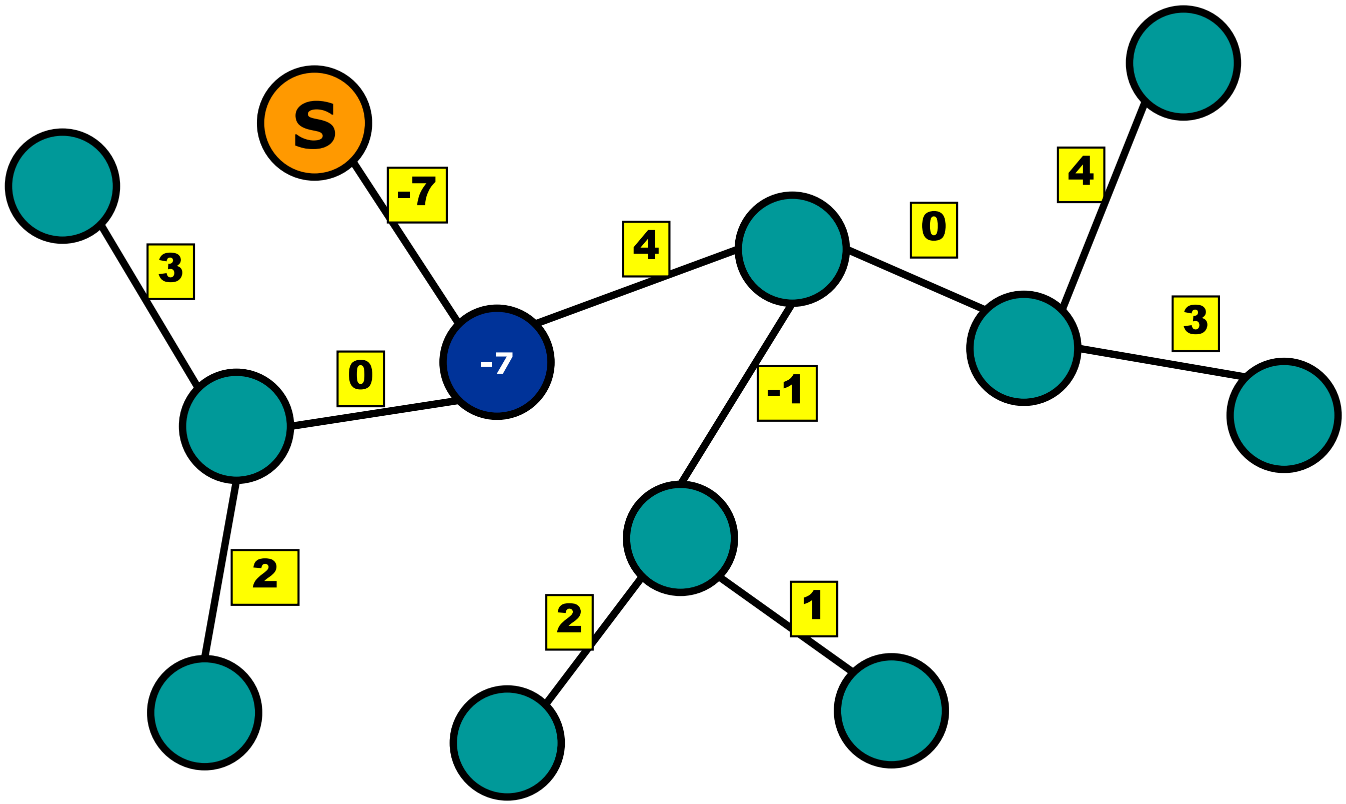
# Tree: source-to-all

**Relax in DFS Order**



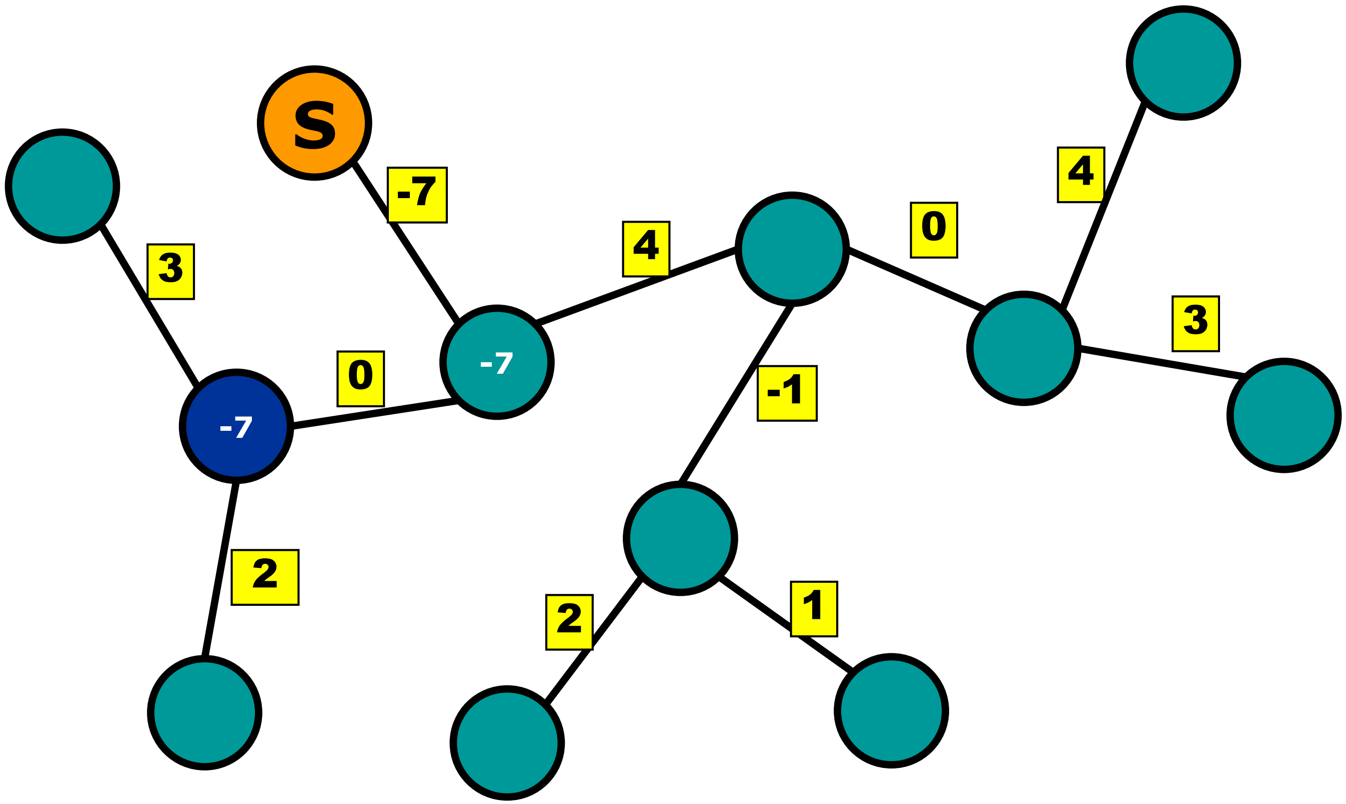
# Tree: source-to-all

**Relax in DFS Order**



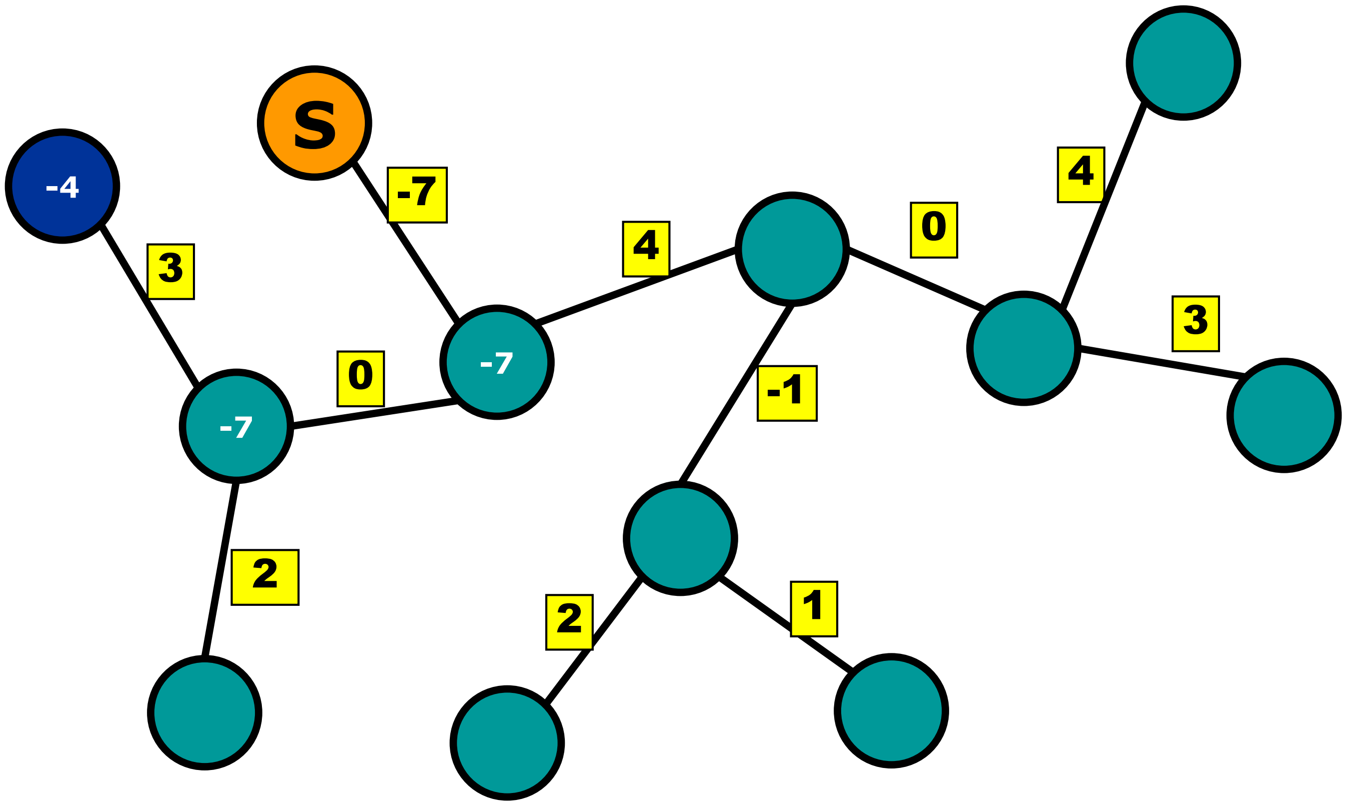
# Tree: source-to-all

## Relax in DFS Order



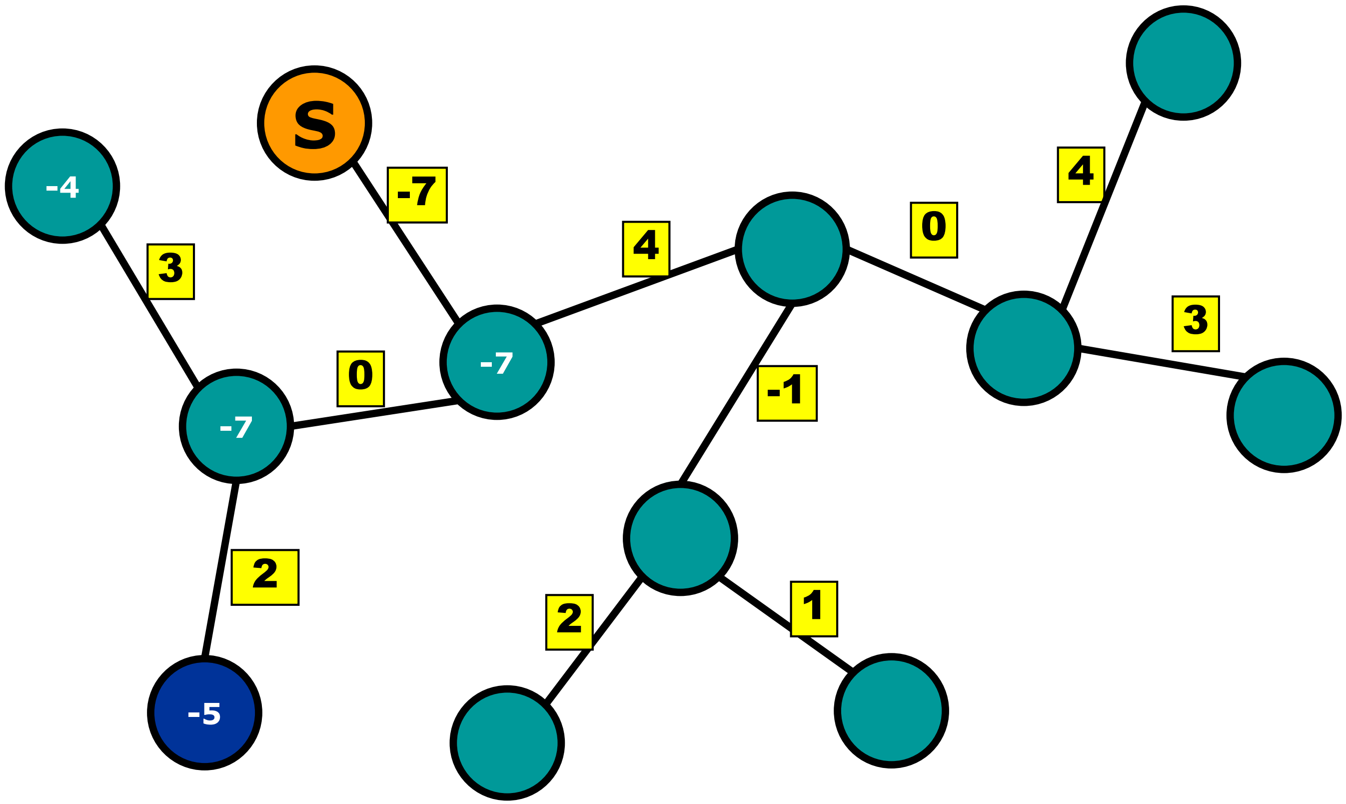
# Tree: source-to-all

**Relax in DFS Order**



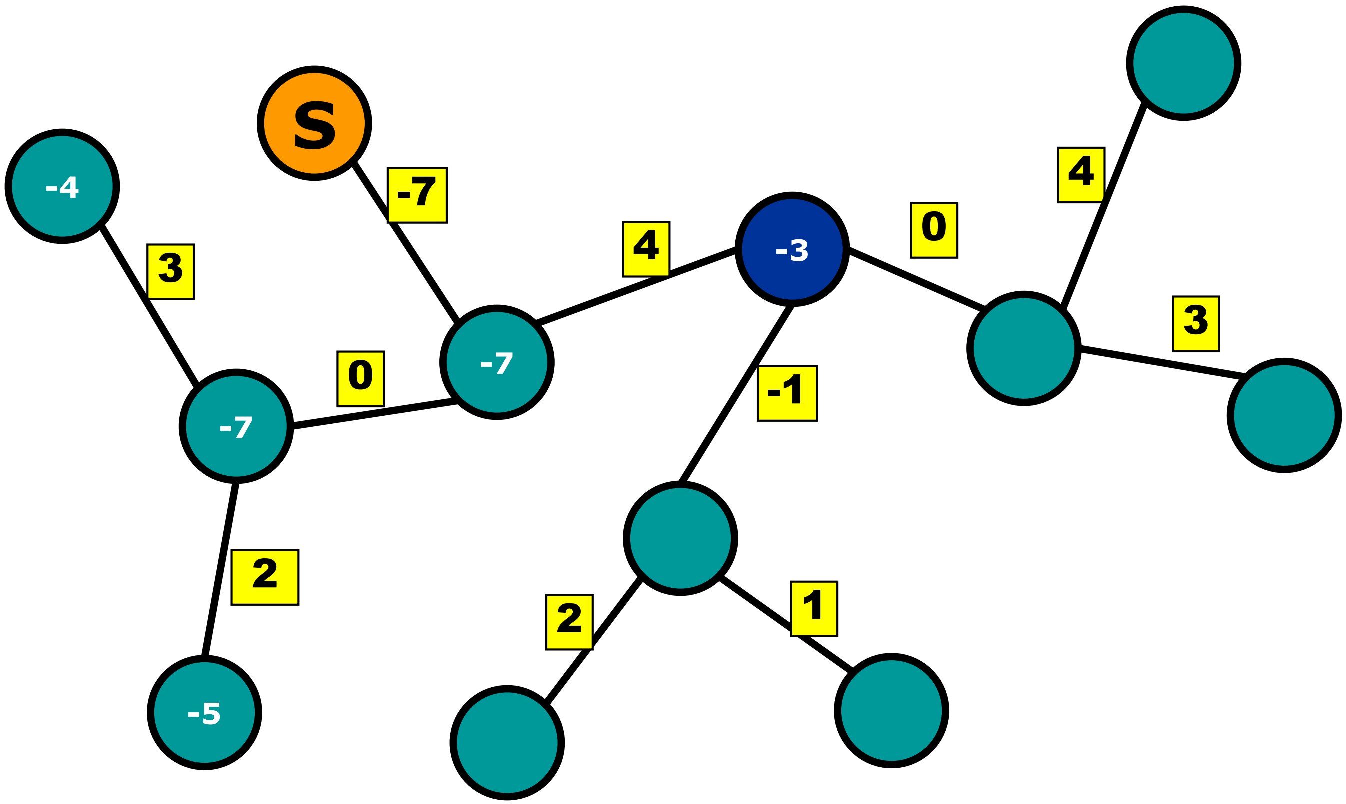
# Tree: source-to-all

## Relax in DFS Order



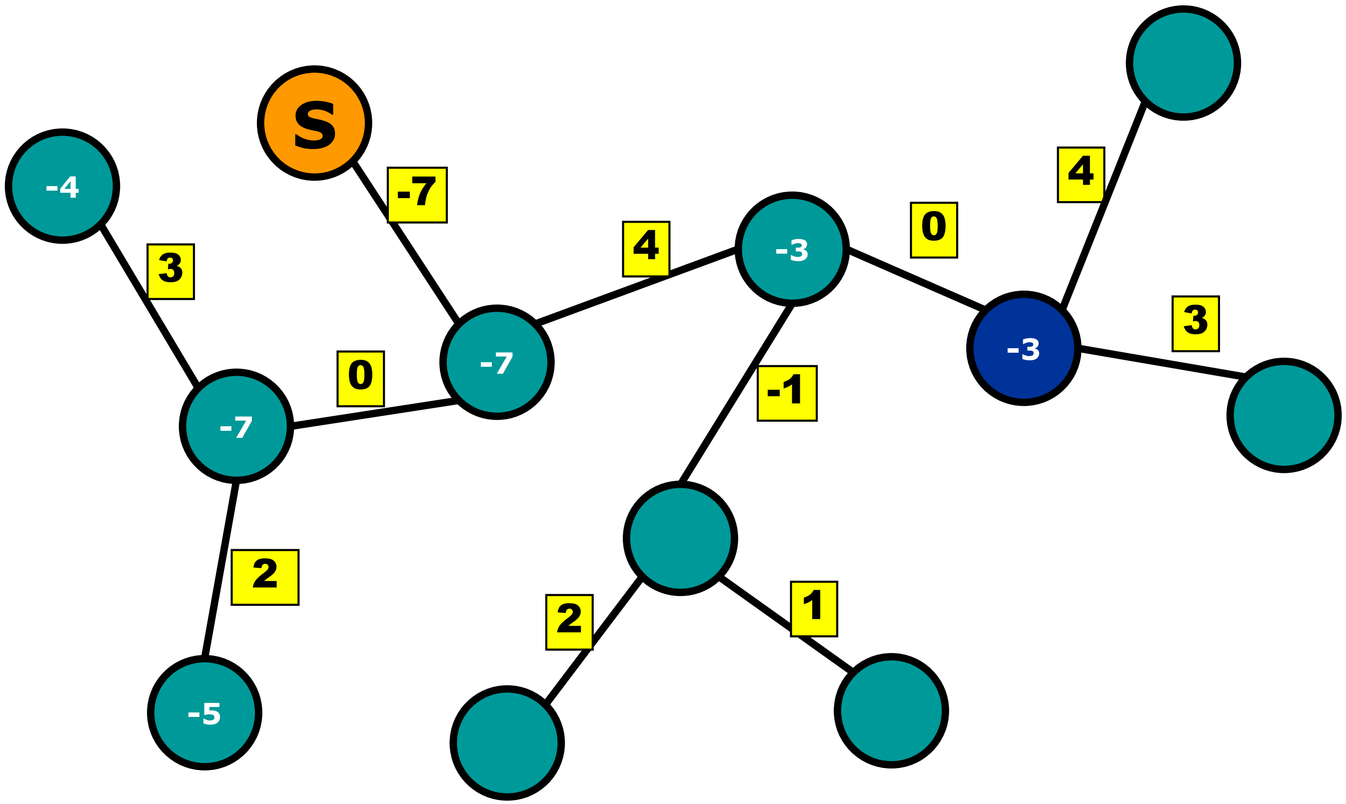
# Tree: source-to-all

**Relax in DFS Order**



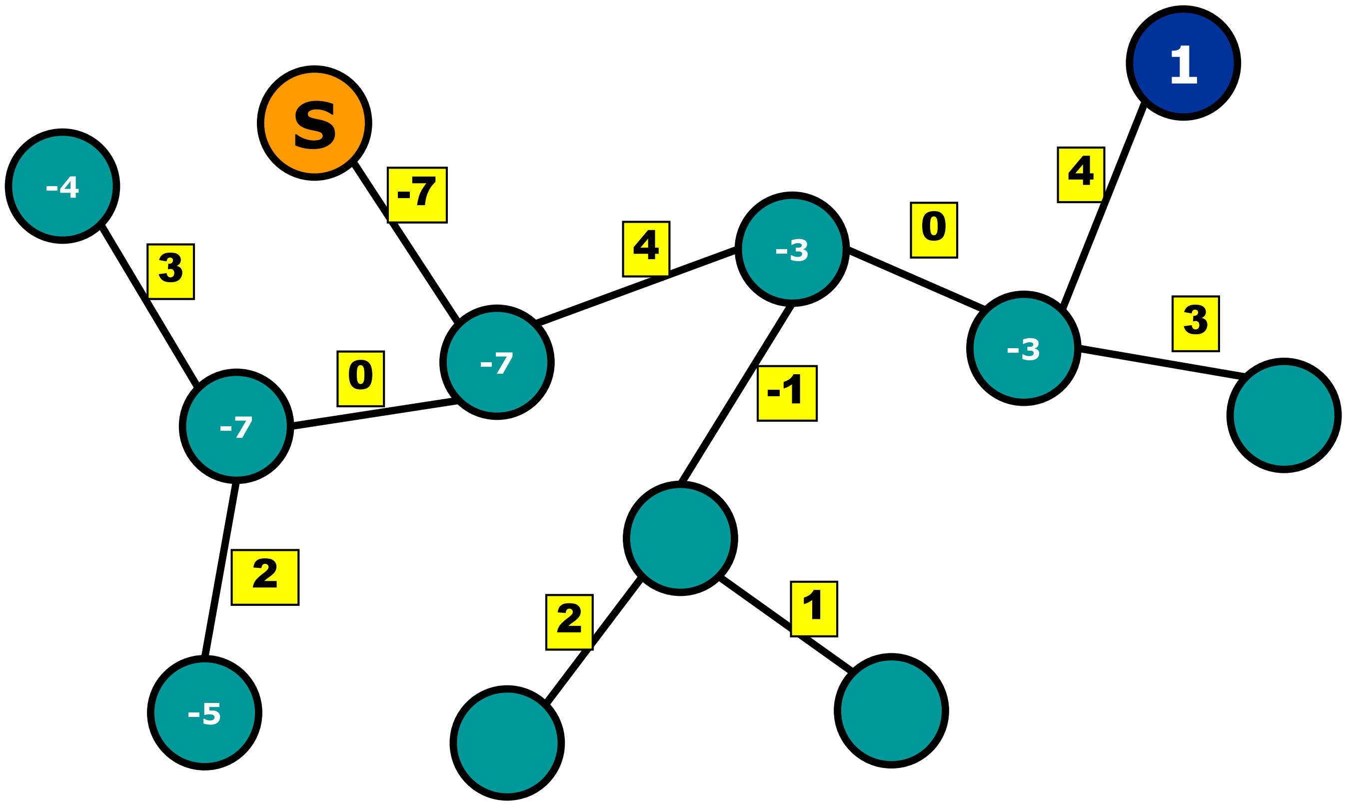
# Tree: source-to-all

**Relax in DFS Order**



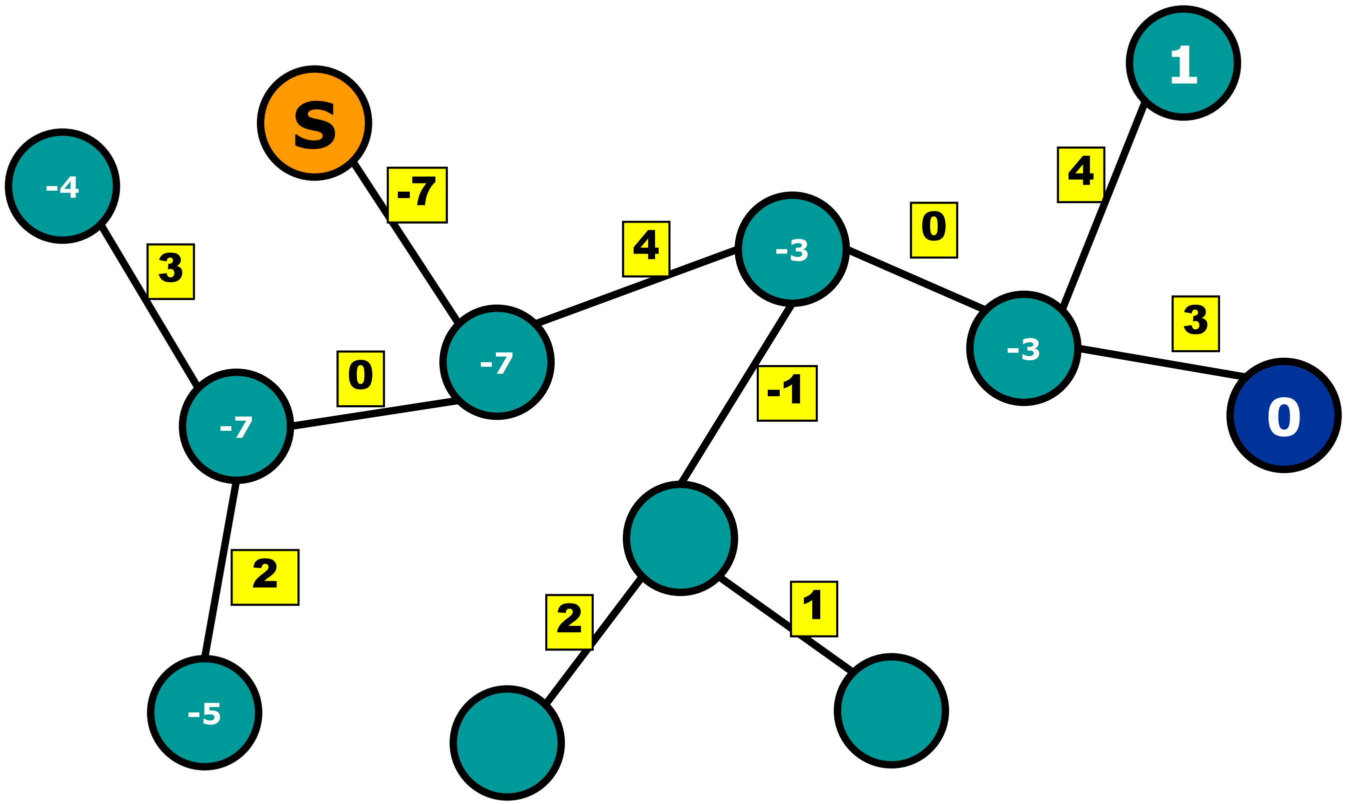
# Tree: source-to-all

## Relax in DFS Order



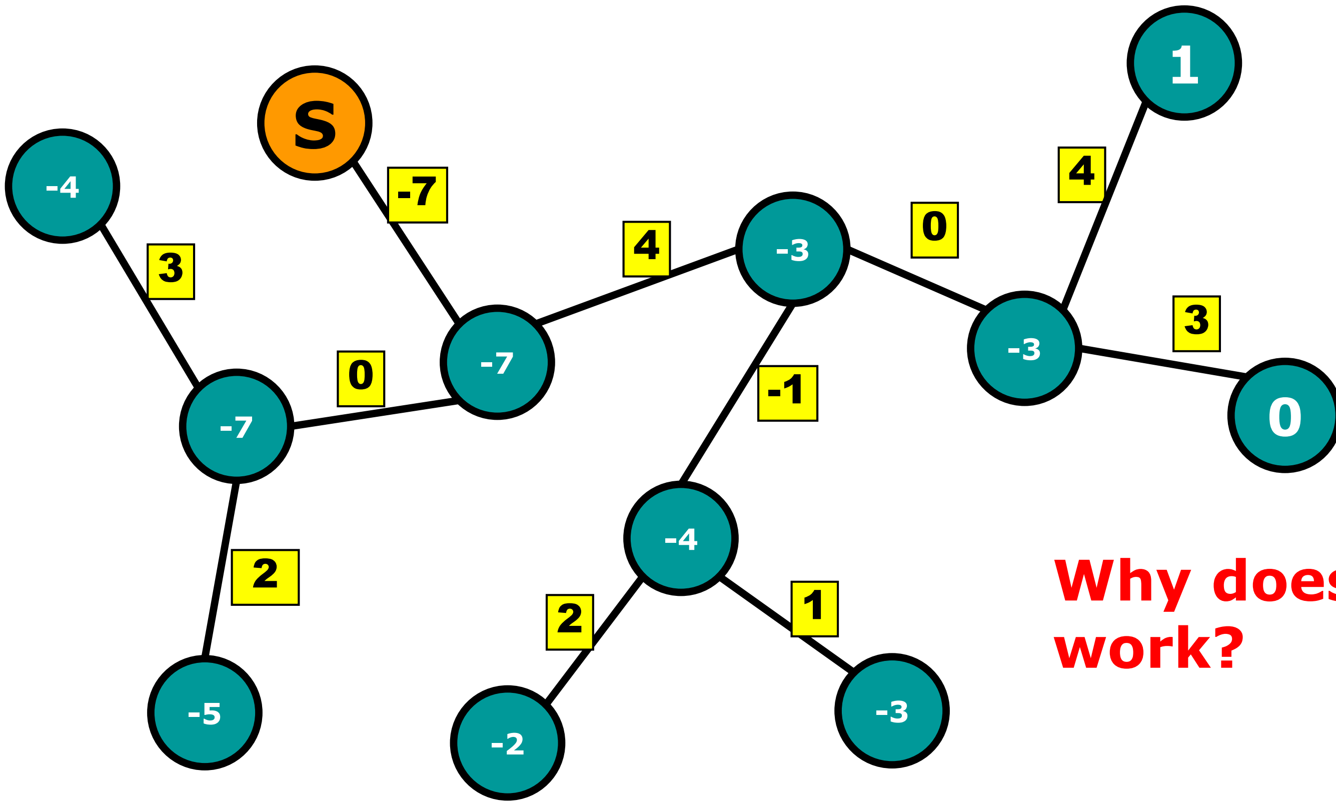
# Tree: source-to-all

**Relax in DFS Order**



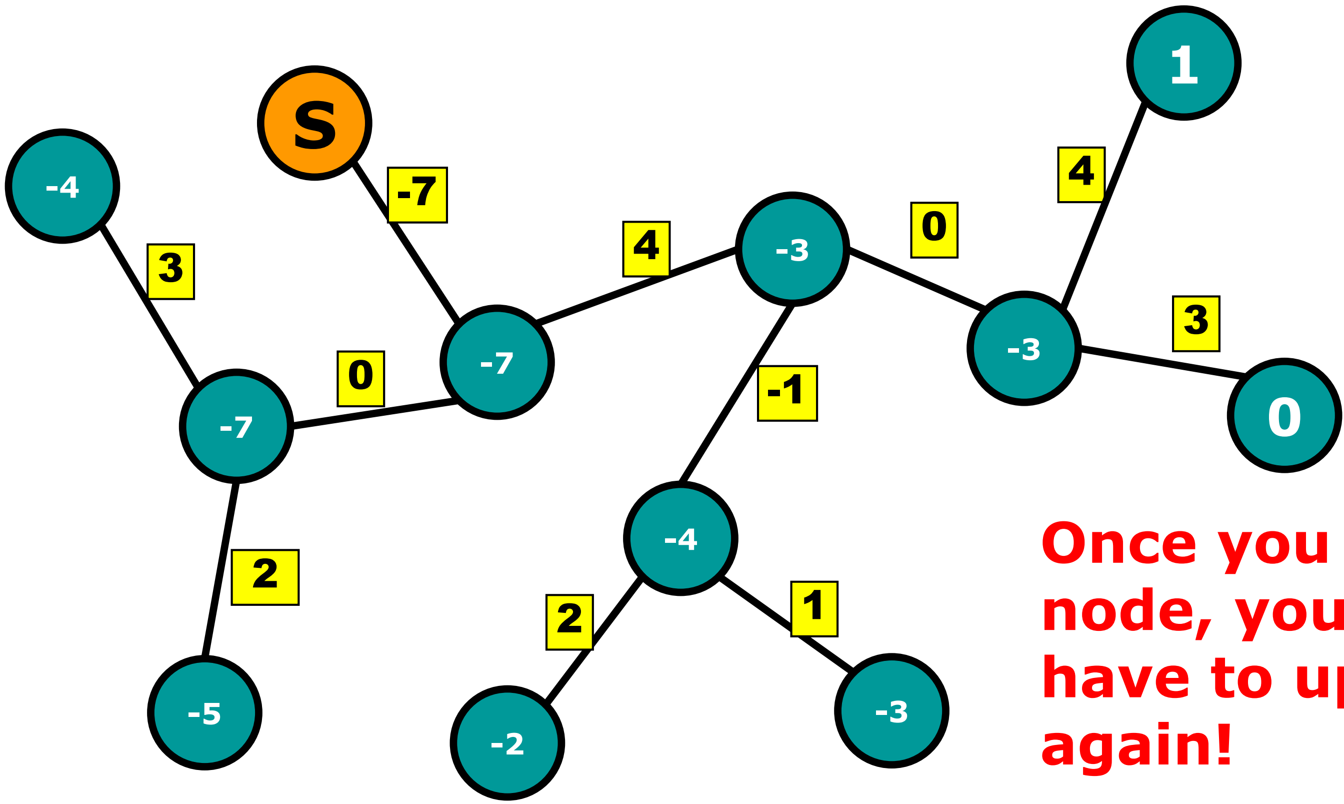
# Tree: source-to-all

**Relax in DFS Order**




**Why does this work?**

# Tree: source-to-all

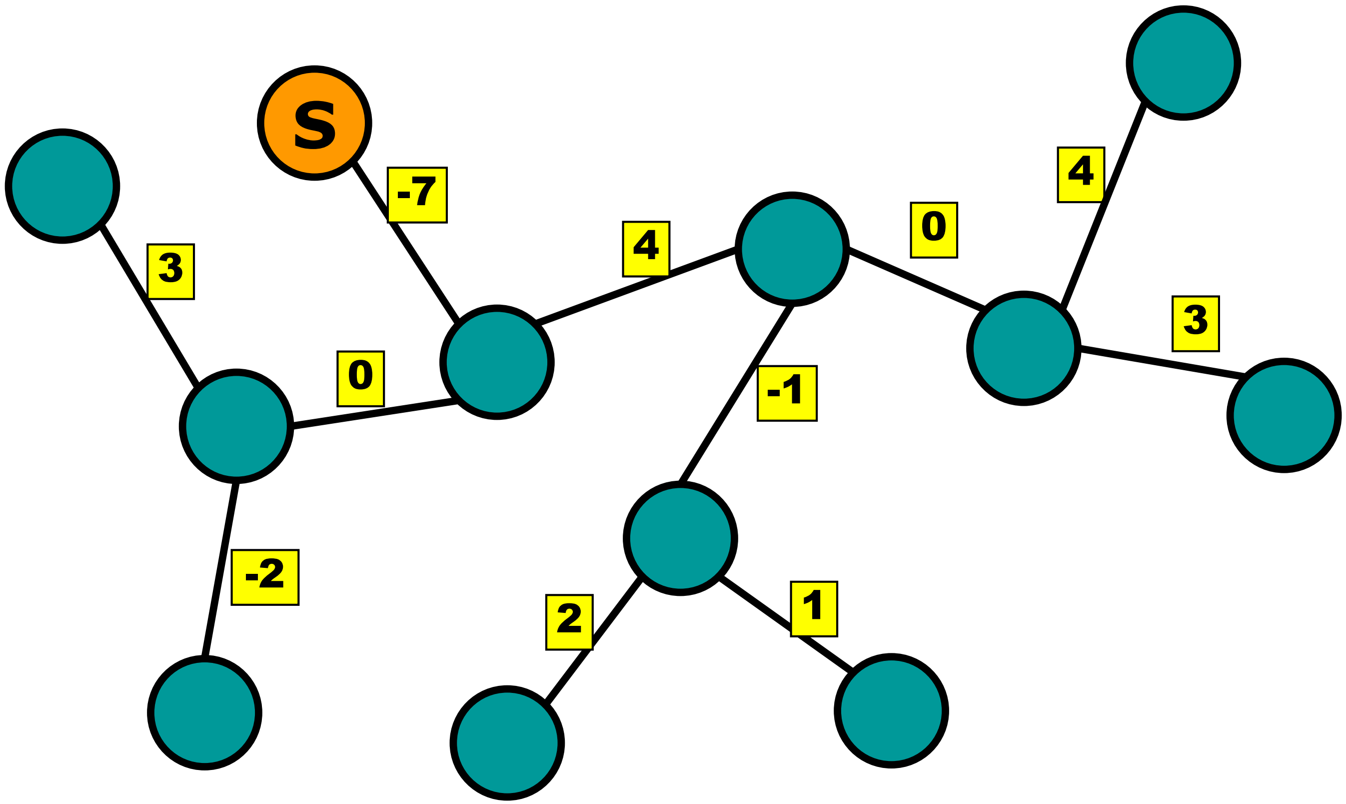



**Once you update a node, you never have to update it again!**

# Undirected Weighted Tree

Time Complexity?

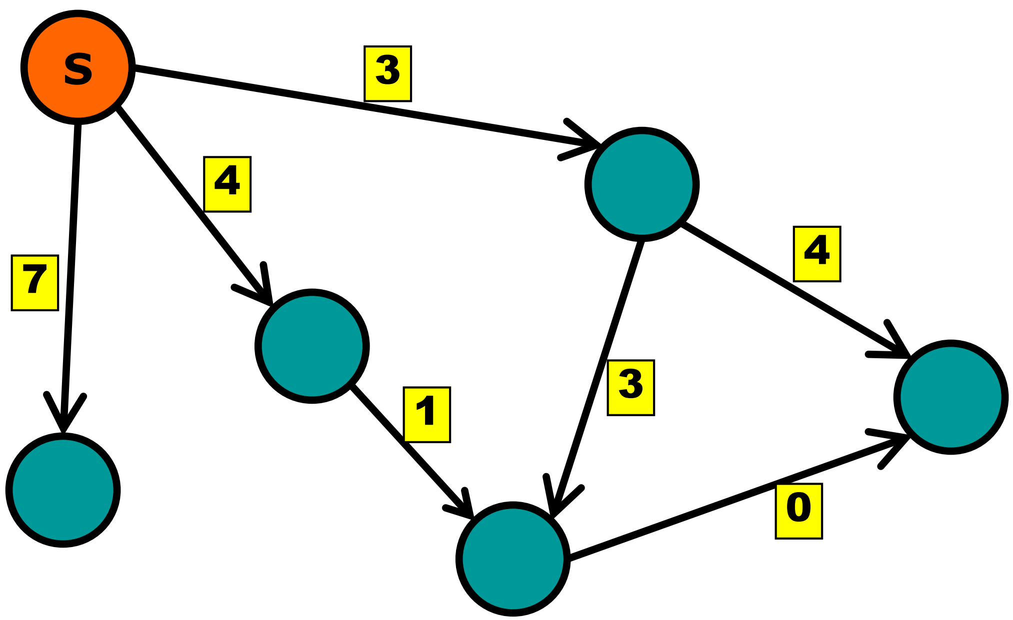
**every node only has one parent (except the root).**
$O(V) = O(E)$ **edges.**



# Special Cases

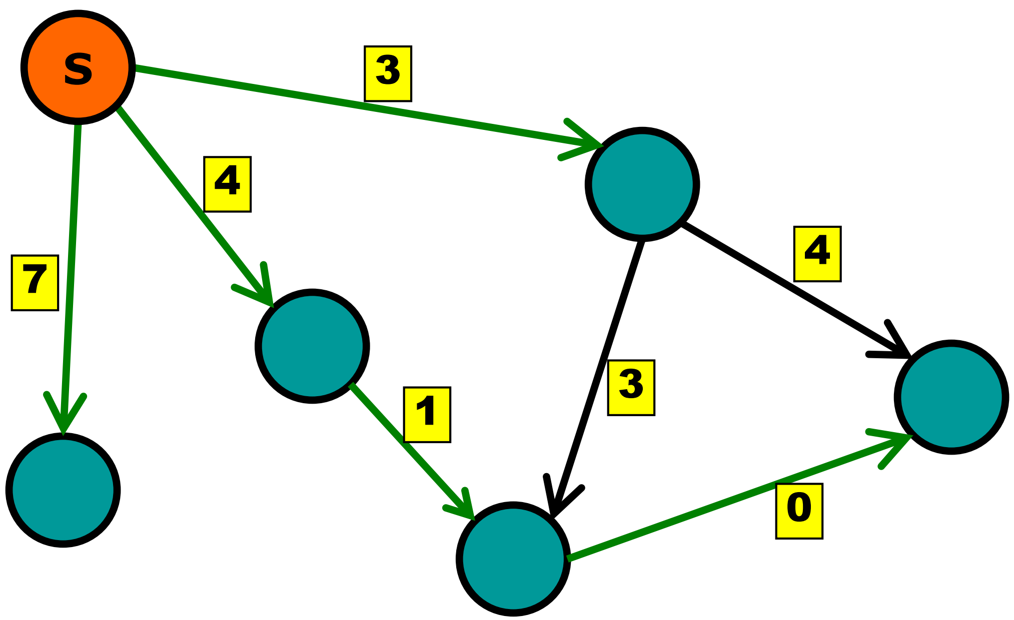
| Condition | Algorithm | Time Complexity |
| --- | --- | --- |
| No Negative Weight Cycles | Bellman-Ford Algorithm | $O(VE)$ |
| On Unweighted Graph (or equal weights) | BFS | $O(V+E)$ |
| No Negative Weights | Dijkstra's Algorithm | |
| Negative Weights | Modified Dijkstra's Algorithm | |
| On Tree | BFS / DFS | $O(V)$ |
| On DAG | Dynamic Programming | |

# General graph: non-negative edges



# General graph: non-negative edges

Shortest paths form a **tree**



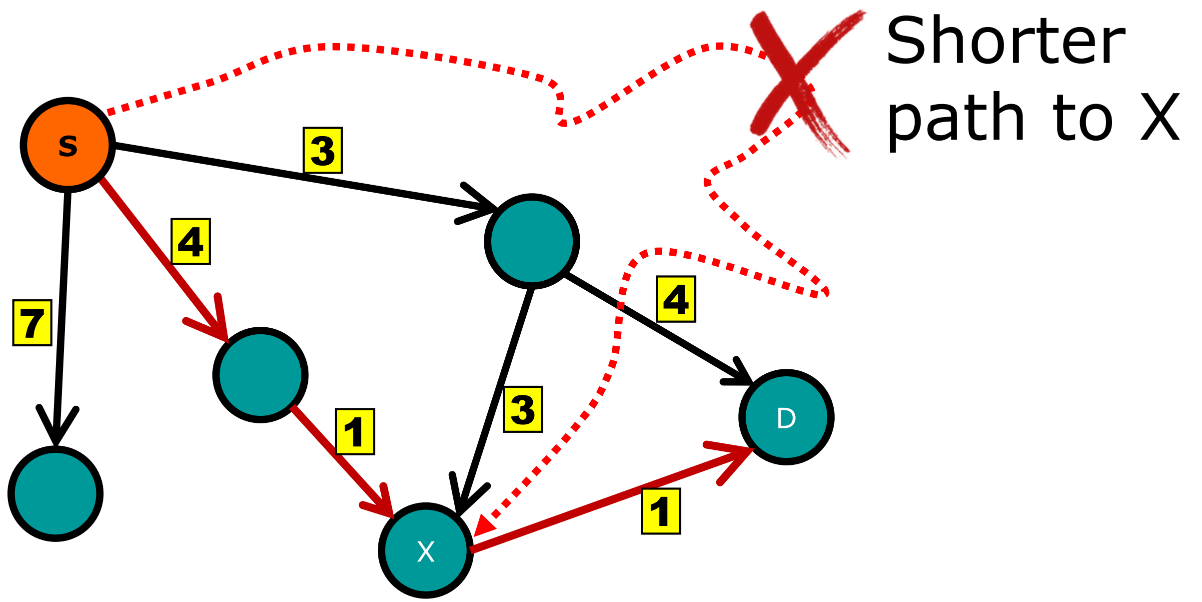
# General graph: non-negative edges

**Key property:**

If p is the shortest path from S to D,

and if p goes through X,

then p is also the shortest path from S to X (and from X to D).



# Dijkstra's Algorithm

**Key idea:**

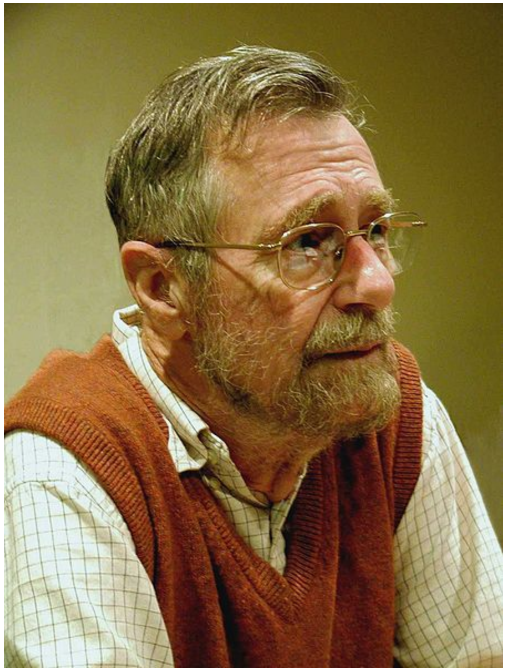
Relax the edges in the "right" order.

Only relax each edge **once**:

- $O(E)$ cost (for relaxation step)

# Edsger W. Dijkstra

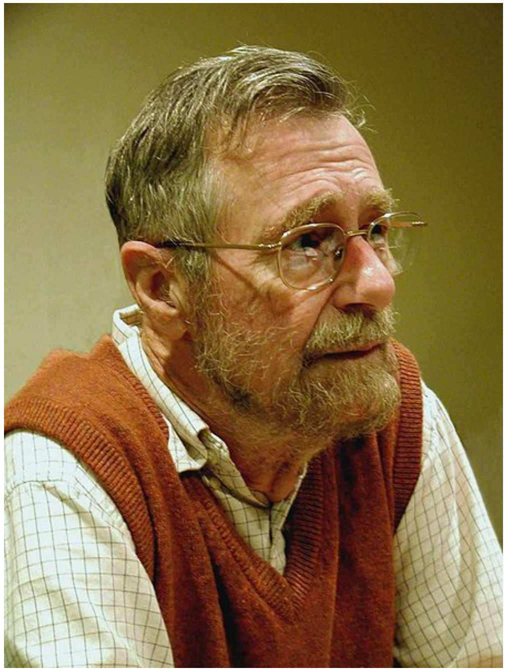
- *“Computer science is no more about computers than astronomy is about telescopes.”*
- *“The question of whether a computer can think is no more interesting than the question of whether a submarine can swim.”*
- *“There should be no such thing as boring mathematics.”*
- *“Elegance is not a dispensable luxury but a factor that decides between success and failure.”*
- *“Simplicity is prerequisite for reliability.”*



1930-2002

# Edsger W. Dijkstra

- *“It is practically impossible to teach good programming to students that have had a prior exposure to BASIC: as potential programmers they are mentally mutilated beyond hope of regeneration.”*
- *“The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offense.”*
- *“Object-oriented programming is an exceptionally bad idea which could only have originated in California.”*

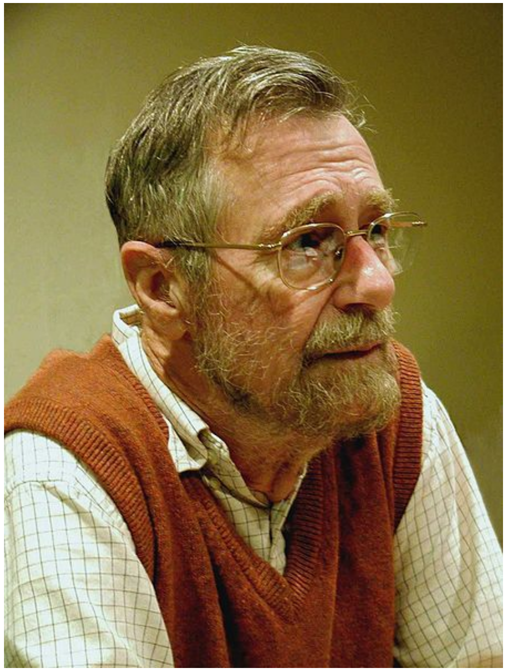


1930-2002

# Edsger W. Dijkstra

**From Wikipedia:**

- *His approach to teaching was unconventional …*
- *He invited the students to suggest ideas, which he then explored, or refused to explore because they violated some of his tenets.*
- *He conducted his final examinations orally, over a whole week.*
- *Each student was examined in Dijkstra's office or home, and an exam lasted several hours.*



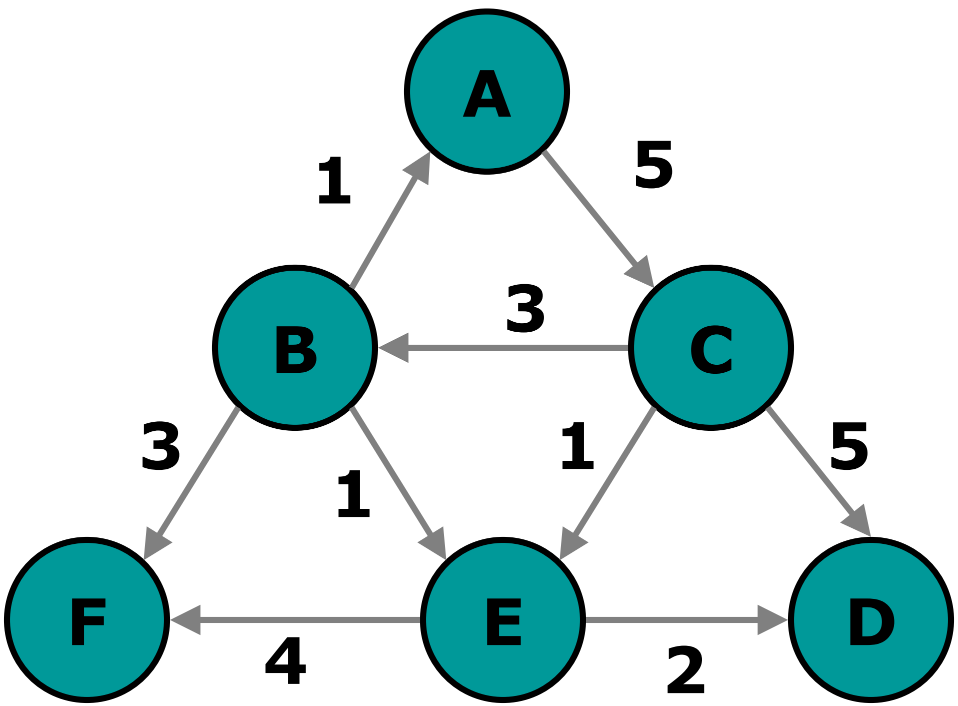
1930-2002

# Dijkstra's algorithm

**Basic idea:**

- Maintain distance <u>**estimate**</u> for every node
- Begin with empty shortest-path-tree
- Repeat:
  - Consider vertex with minimum **estimate**
  - Add vertex to **shortest-path-tree**
  - Relax all outgoing edges

# Dijkstra's algorithm

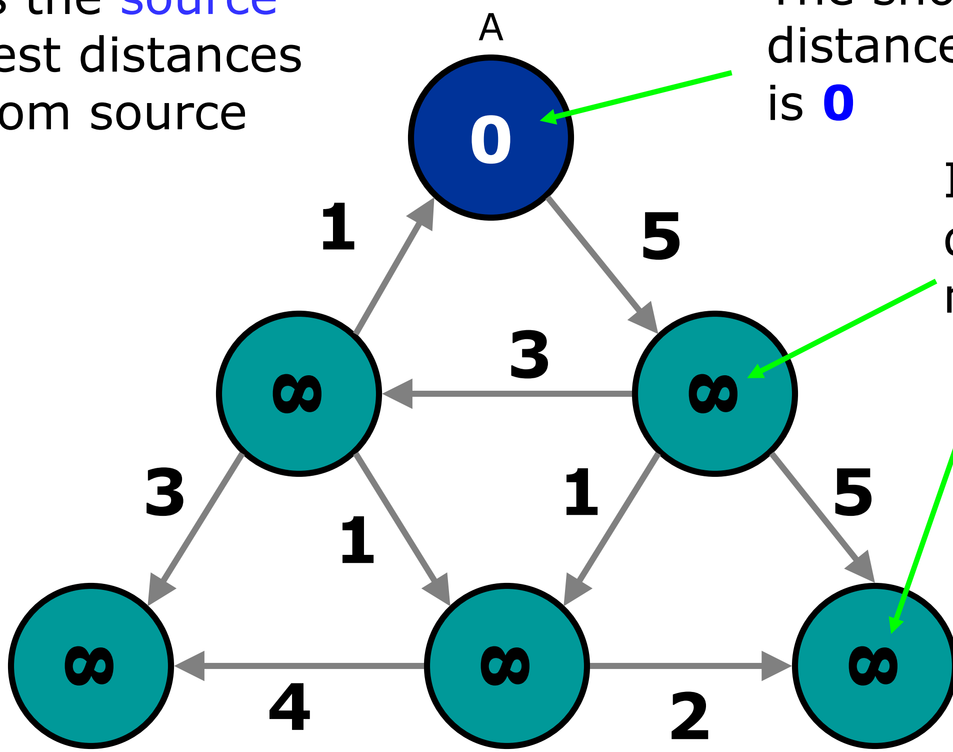


# Dijkstra's algorithm

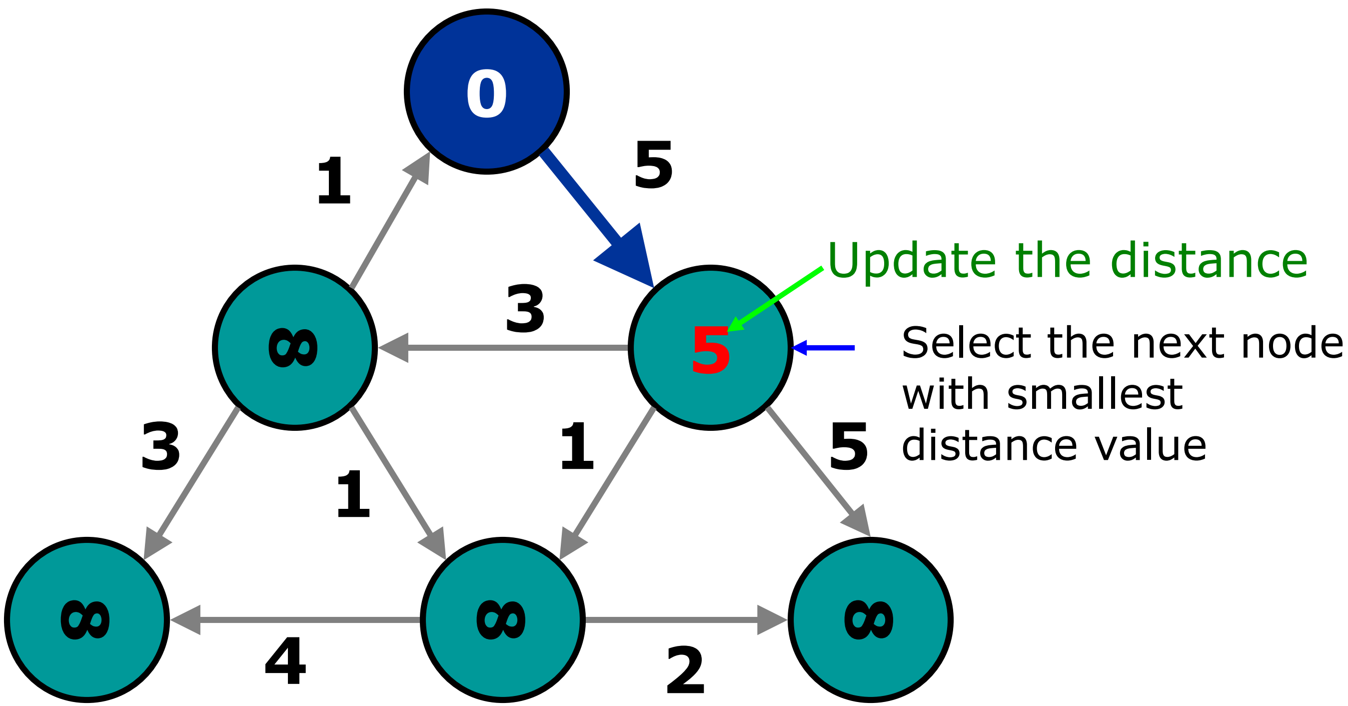
Use node A as the source i.e. find shortest distances of all nodes from source node A

The shortest distance of A from A is **0**

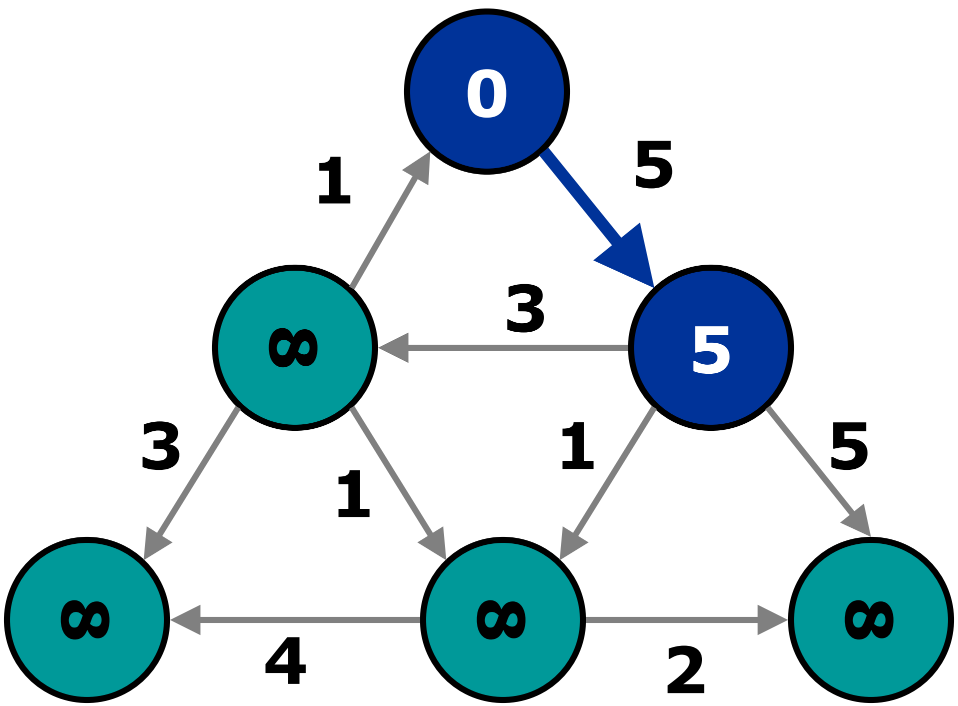
Initialize the distances of others nodes as **infinite**



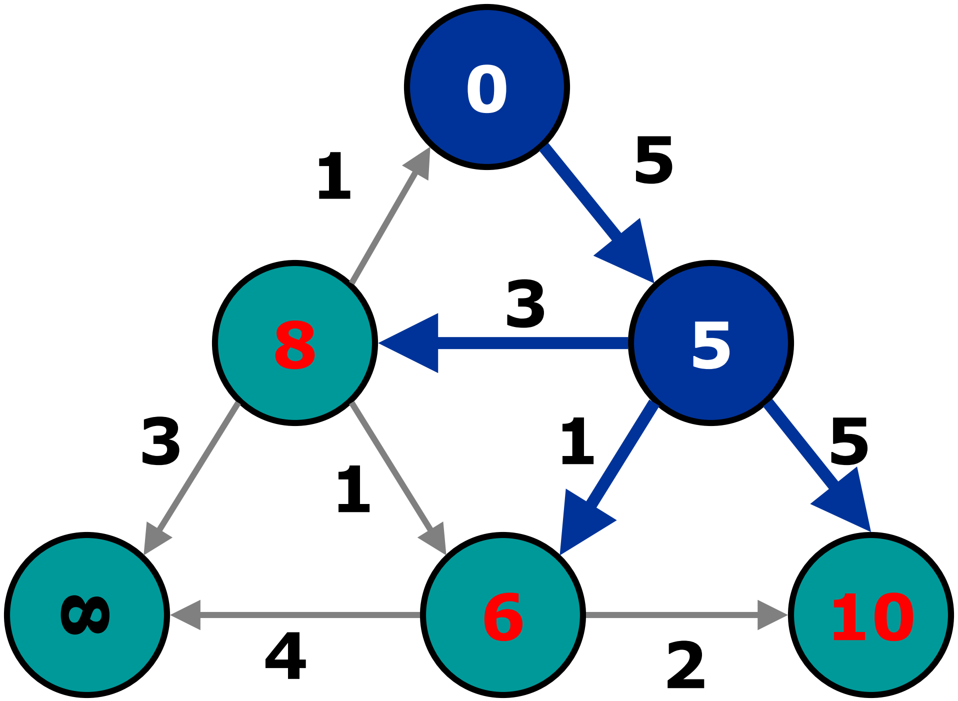
# Dijkstra's algorithm



# Dijkstra's algorithm

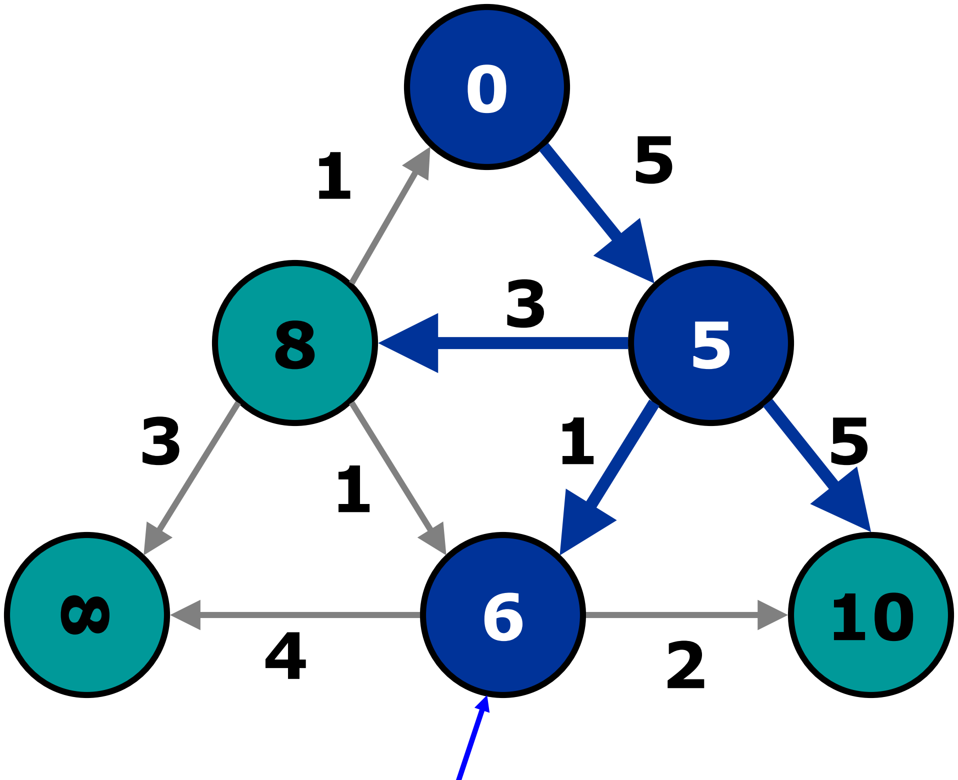


# Dijkstra's algorithm



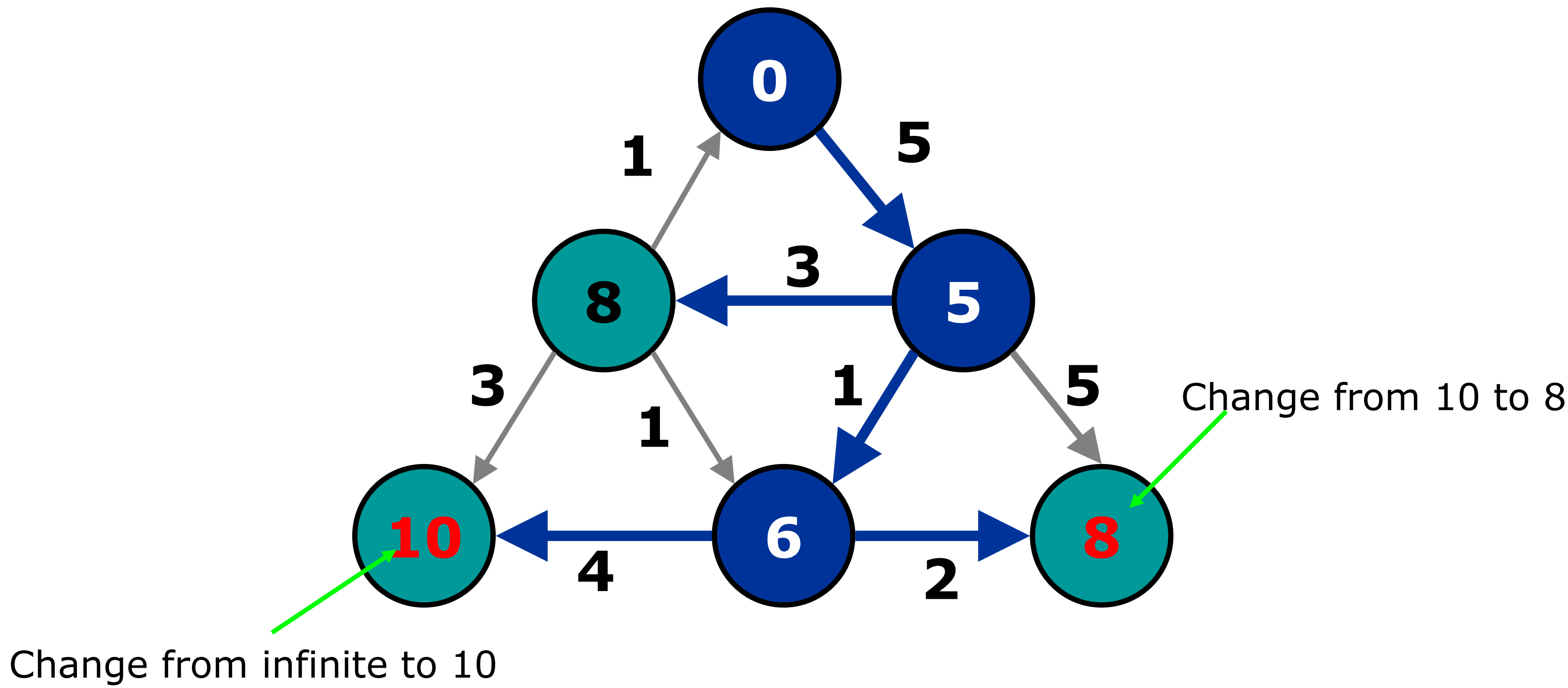

(1) Update distance for 3 nodes

# Dijkstra's algorithm

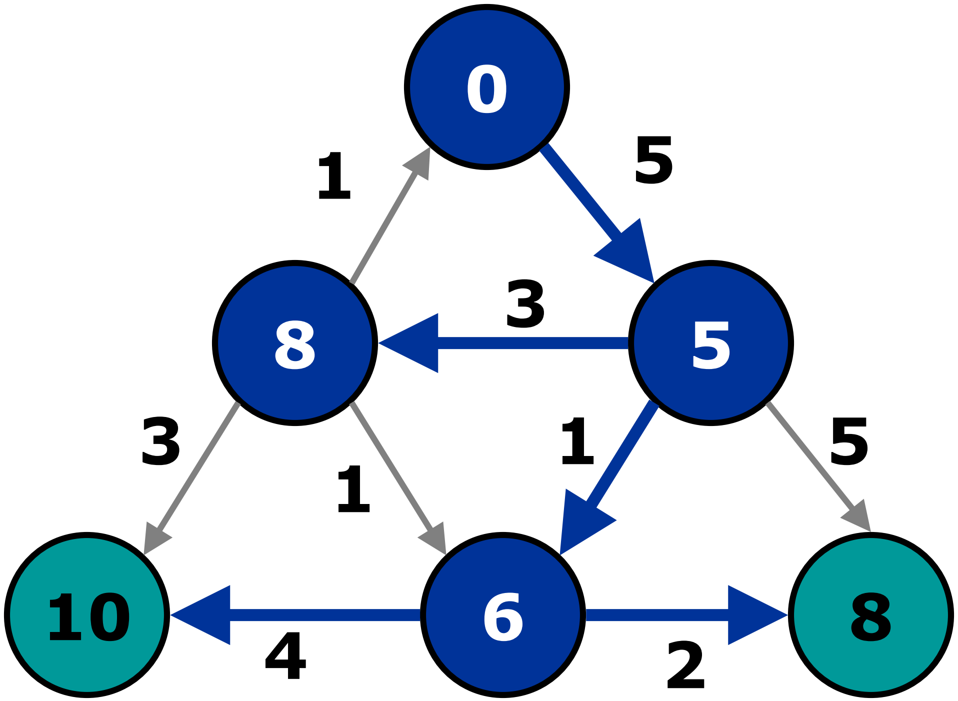



(2) Select the next node with smallest distance value

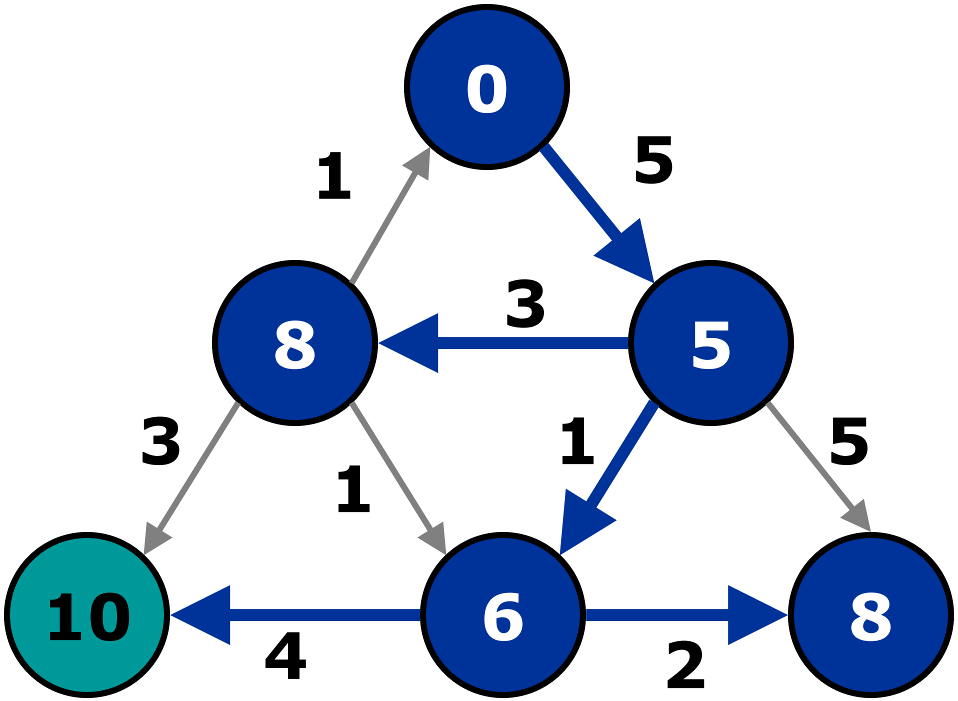
# Dijkstra's algorithm



# Dijkstra's algorithm

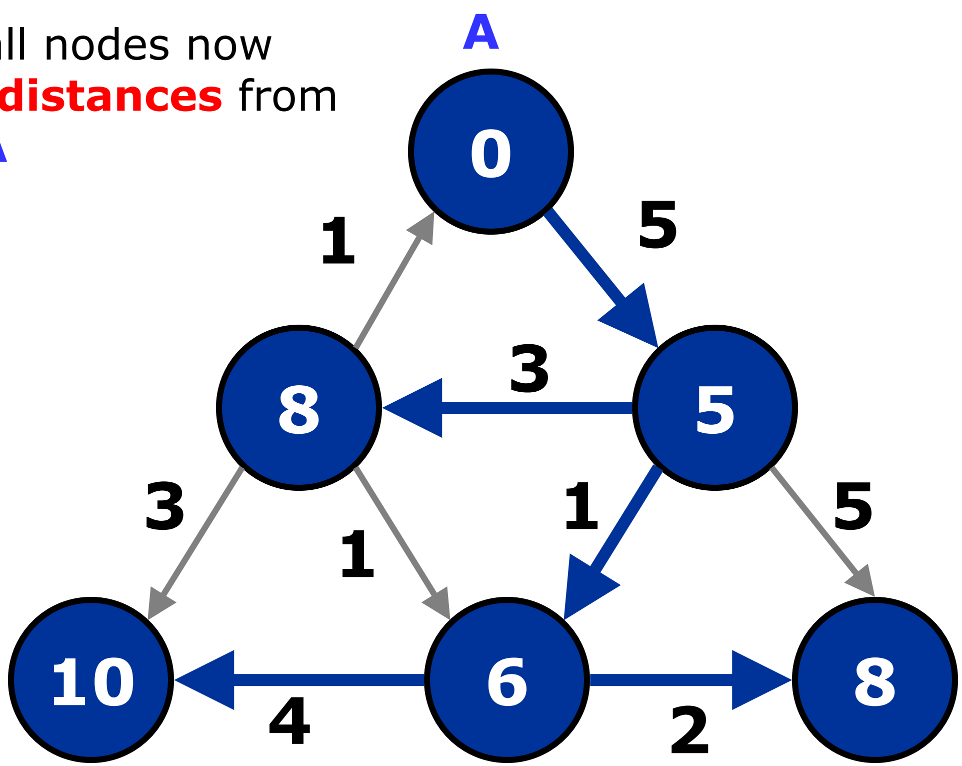


# Dijkstra's algorithm



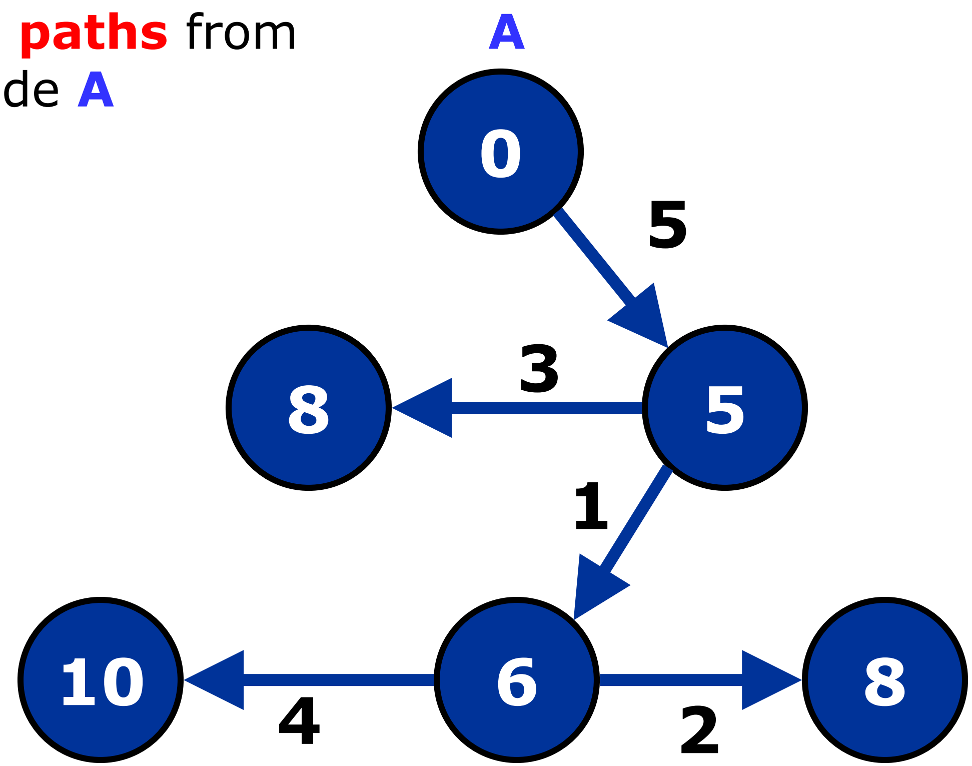
# Dijkstra’s algorithm

The distances of all nodes now are the **shortest distances** from the source node **A**



# Dijkstra's algorithm

The **shortest paths** from the source node **A**



# Dijkstra's algorithm

```
color all vertices yellow
                // yellow nodes are those not yet processed
foreach vertex w
   distance(w) = INFINITY
distance(s) = 0            //source node distance is 0
```

# Dijkstra's algorithm

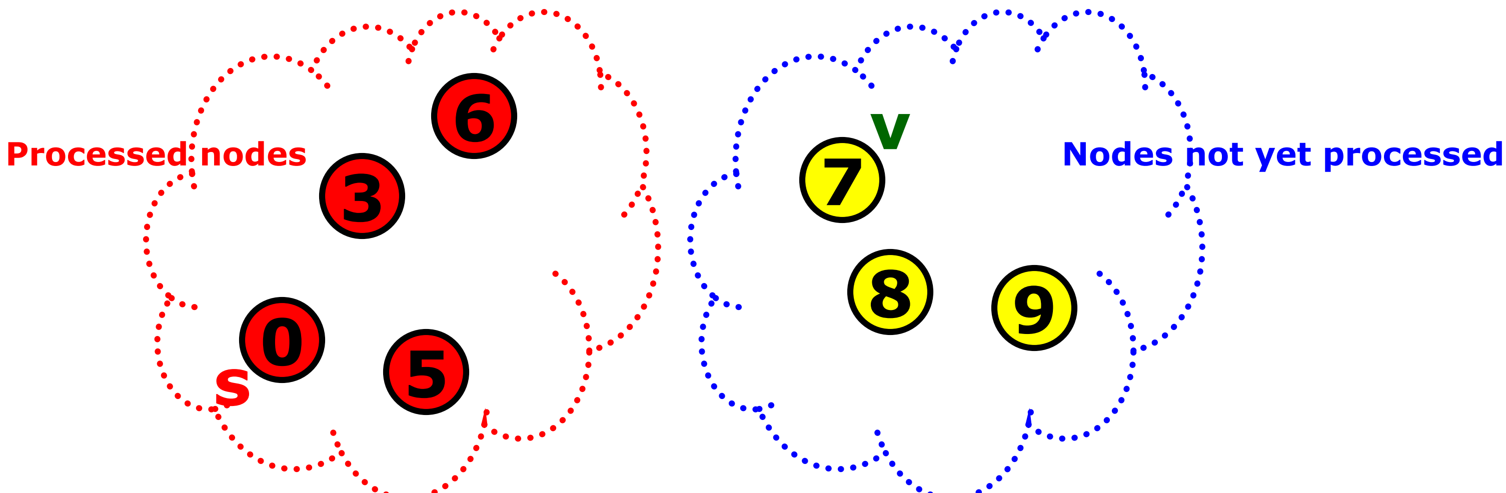
**while** there are yellow vertices //unprocessed nodes are yellow

v = yellow vertex with min distance(v)

color v red // red vertices are vertices with shortest distances from s found

**foreach** yellow neighbour w of v

**relax**(v,w)



# Time Complexity

```
color all vertices yellow
foreach vertex w
   distance(w) = INFINITY
distance(s) = 0
while there are yellow vertices
   v = yellow vertex with min distance(v)
   color v red
   foreach yellow neighbour w of v
        relax(v,w)
```

# Time Complexity

- Initialization takes O(V) time. // V = no of nodes
- Picking the vertex with minimum distance(v) can take O(V) time, and relaxing the neighbours take O(adj(v)) time. // adj(v) = adjacent nodes of v
- The sum of these over all vertices is $O(V^2+E)$.

  // Because sum adj(v) = E where E is the no of edges
- Can we improve this if we improve the running time for picking the minimum distance()? Yes, use priority queue to pick the minimum.

# Using priority queue

```
foreach vertex w
    distance(w) = INFINITY
distance(s) = 0
pq = new PriorityQueue(V)     // minimum heap
              //with all vertices and their distances (as keys)
while pq is not empty
    v = pq.deleteMin()    // O(log V)
    foreach neighbour w of v
        relax(v,w)
```

Since priority queue supports efficient minimum picking operation, we can use a priority queue here to improve the running time. Note that we no longer color vertices here. Yellow vertices in the previous pseudocode are now vertices that are in the priority queue.

# Time Complexity - Initialization

```
foreach vertex w
    distance(w) = INFINITY
distance(s) = 0
pq = new PriorityQueue(V)
                // with all vertices and their distances (as keys)
```

Initialization still takes O(V)

# Time Complexity - Main loop

```
while pq is not empty
   v = pq.deleteMin()
   foreach neighbour w of v
      relax(v,w)
```

We have to be more careful with the analysis of the main loop. We know that each deleteMin() takes O(log V) time. But relax(v,w) is no longer O(1).

Note: Need to expand the relax(v,w) using priority queue

# Time Complexity - Main loop

```
while pq is not empty
    v = pq.deleteMin()                    // O(log V)
    foreach neighbour w of v              // adj(v)
        d = distance(v) + weight(v,w)
        if distance(w) > d then
            distance(w) = d
            pq.decreaseKey(w, d)          // O(log V)
            parent(w) = v
```

- If we expand the code for relax(), we will see that we cannot simply update distance(v), since distance(v) is a key in the priority queue.
- Here, we use an operation called decreaseKey() that updates the key value of distance(v) in the priority queue.

# Time Complexity - Main loop

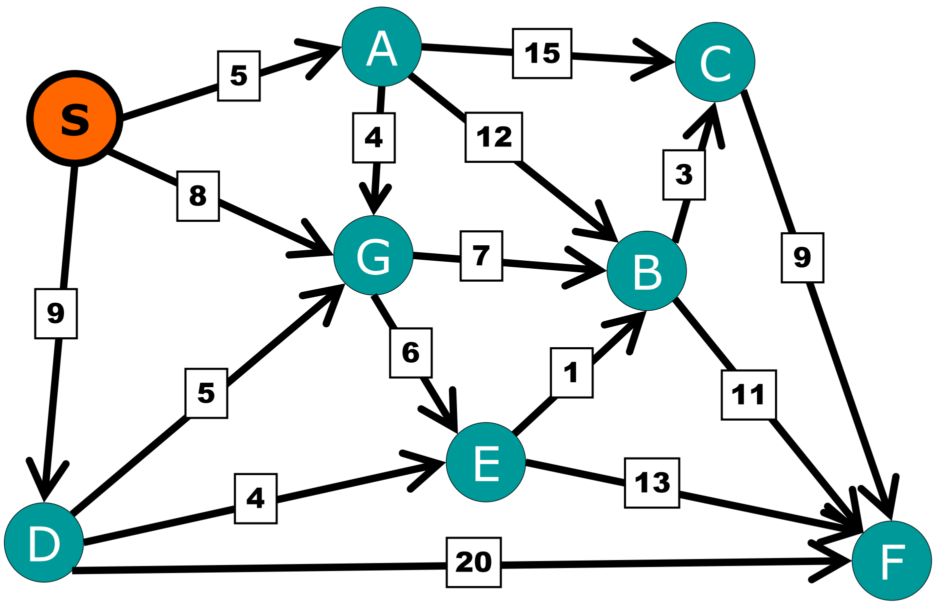
- decreaseKey() can be done in O(log V) time. How?
- The time complexity for this version of Dijkstra's algorithm takes:

  = sum (O(log V) + adj(v) * O(log V)) over all vertices

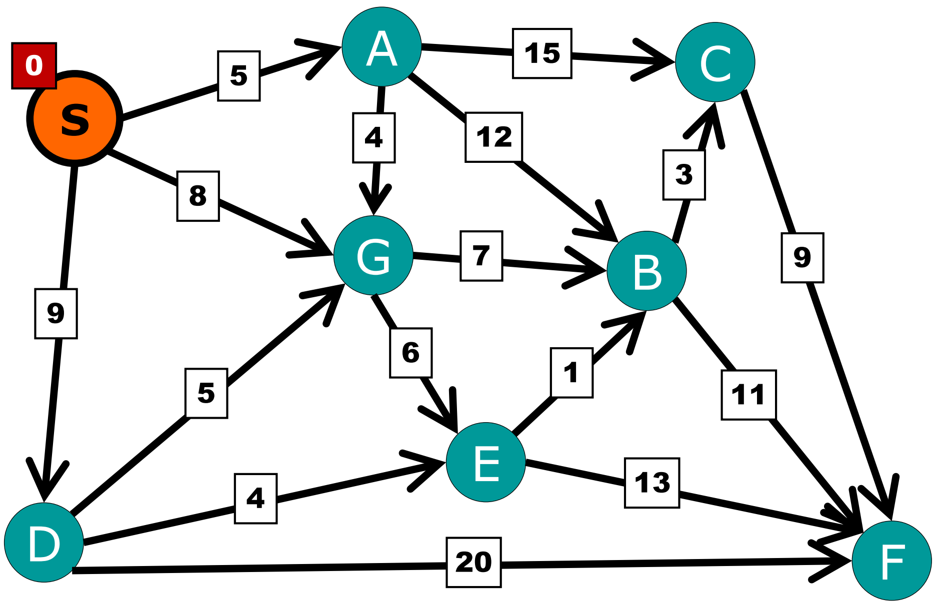
  = O(V log V + E log V) = O((V+E) log V)

  since the total number of adjacent nodes of all nodes is the total number of edges.

# Dijkstra's algorithm



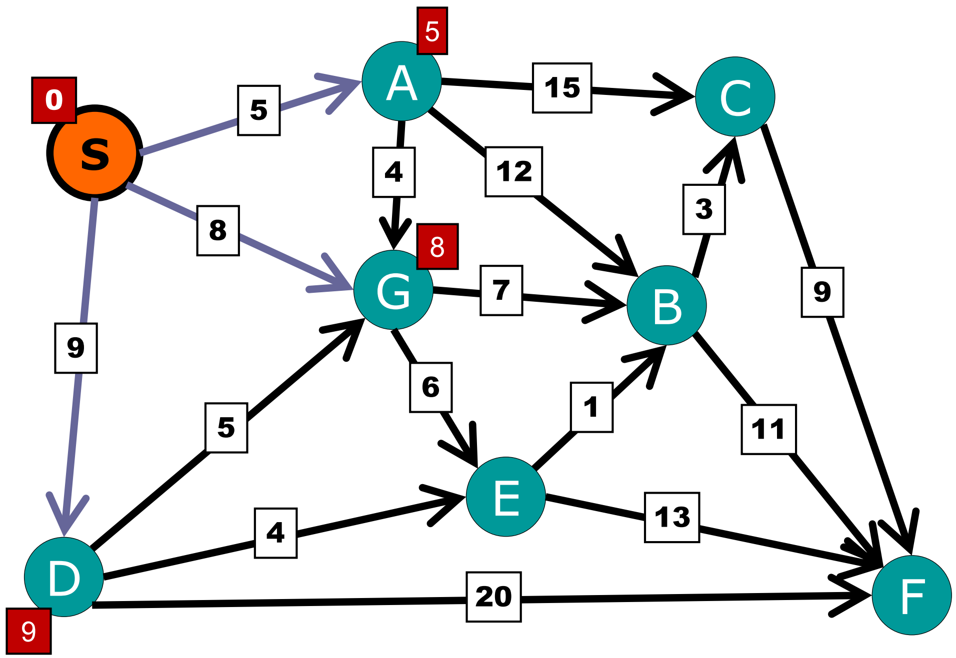
# Dijkstra's algorithm




| Vertex | Dist. |
| --- | --- |
| S | 0 |
| | |
| | |
| | |
| | |
| | |
| | |

Step 1: Add source

# Dijkstra's algorithm

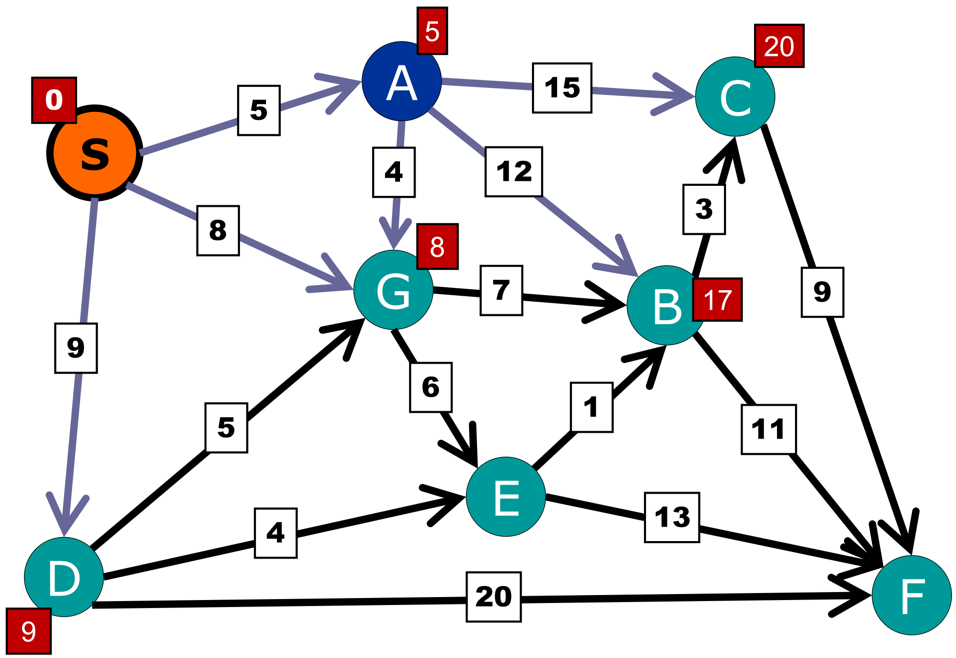



| Vertex | Dist. |
|---|---|
| A | 5 |
| G | 8 |
| D | 9 |
| | |
| | |
| | |
| | |

Step 1: Add source

Step 2: Remove S and relax.

# Dijkstra's algorithm



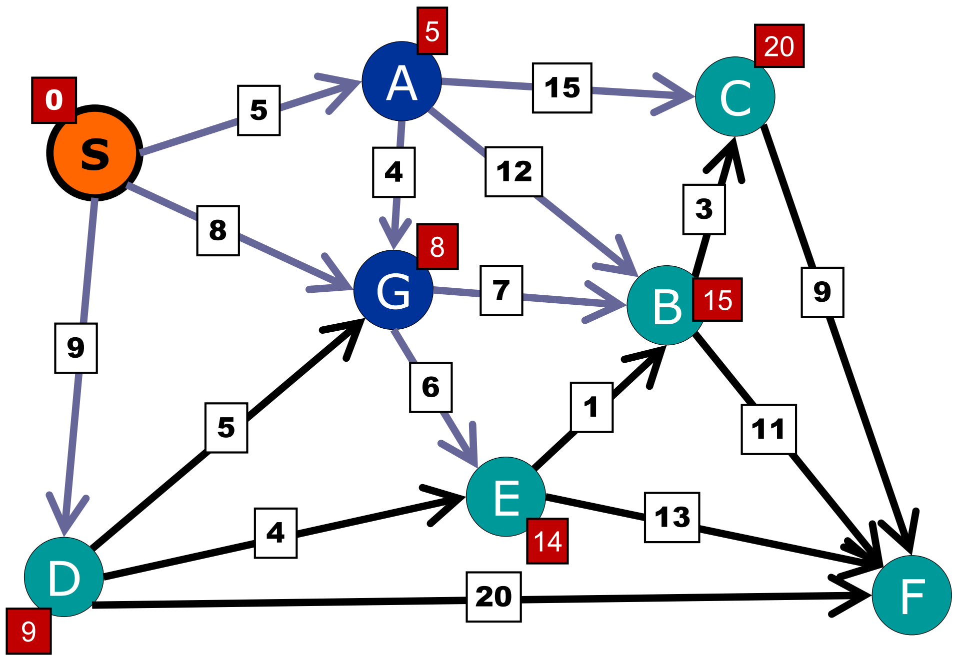

| Vertex | Dist. |
|---|---|
| G | 8 |
| D | 9 |
| B | 17 |
| C | 20 |
| | |
| | |

Step 1: Add source

Step 2: Remove S and relax.

Step 3: Remove A and relax.

# Dijkstra's algorithm




| Vertex | Dist. |
|---|---|
| D | 9 |
| E | 14 |
| B | 15 |
| C | 20 |
| | |
| | |

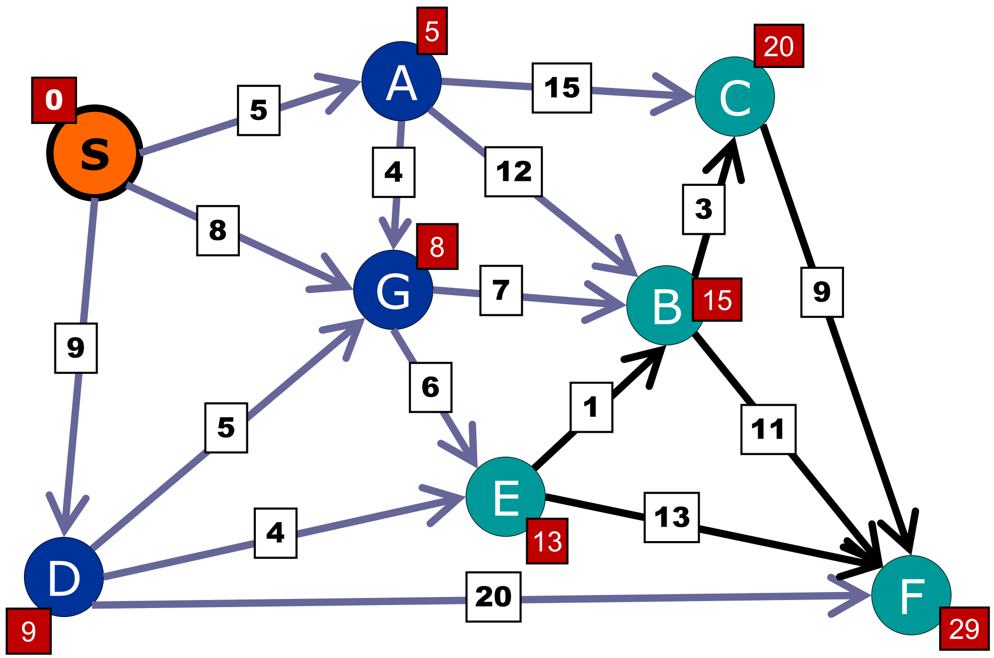
Step 1: Add source

Step 2: Remove S and relax.

Step 3: Remove A and relax.

Step 4: Remove G and relax.

# Dijkstra's algorithm




| Vertex | Dist. |
|---|---|
| E | 13 |
| B | 15 |
| C | 20 |
| F | 29 |
| | |

Step 1: Add source

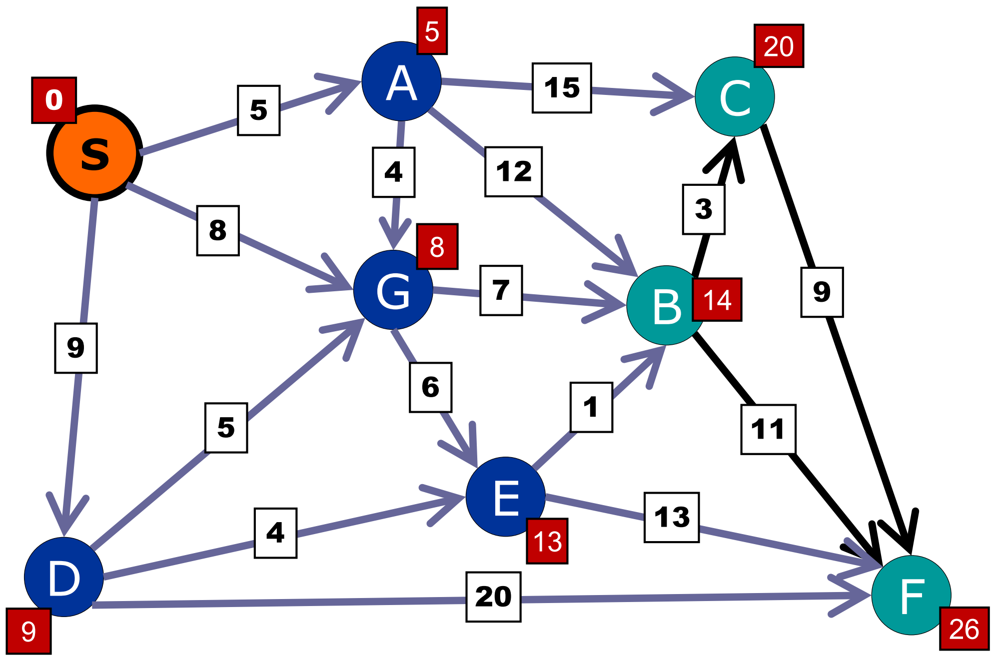
Step 2: Remove S and relax.

Step 3: Remove A and relax.

Step 4: Remove G and relax.

Step 5: Remove D and relax.

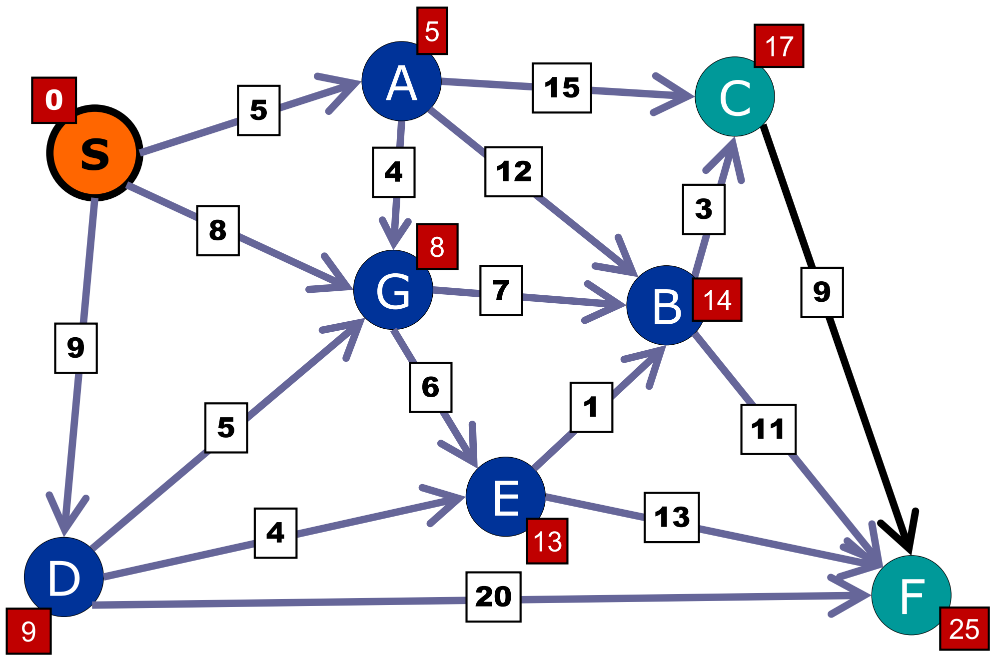
# Dijkstra's algorithm




| Vertex | Dist. |
|---|---|
| **B** | **14** |
| C | 20 |
| **F** | **26** |
| | |

Step 6: Remove E and relax.

# Dijkstra’s algorithm

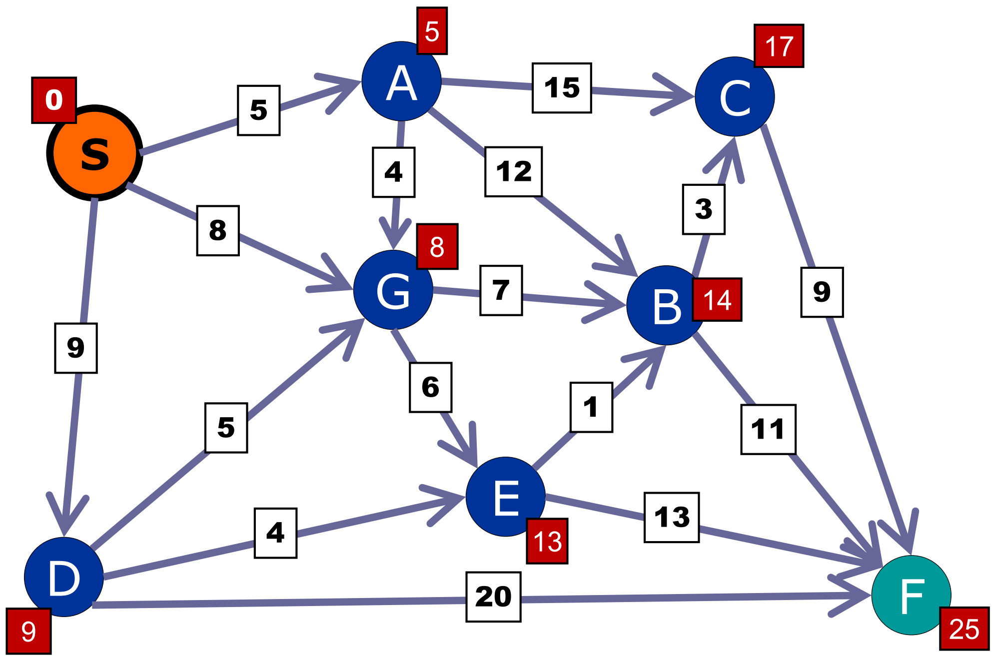



| Vertex | Dist. |
|---|---|
| C | 17 |
| F | 25 |
| | |

Step 6: Remove E and relax.

Step 7: Remove B and relax.

# Dijkstra's algorithm




| Vertex | Dist. |
|---|---|
| F | 25 |
| | |

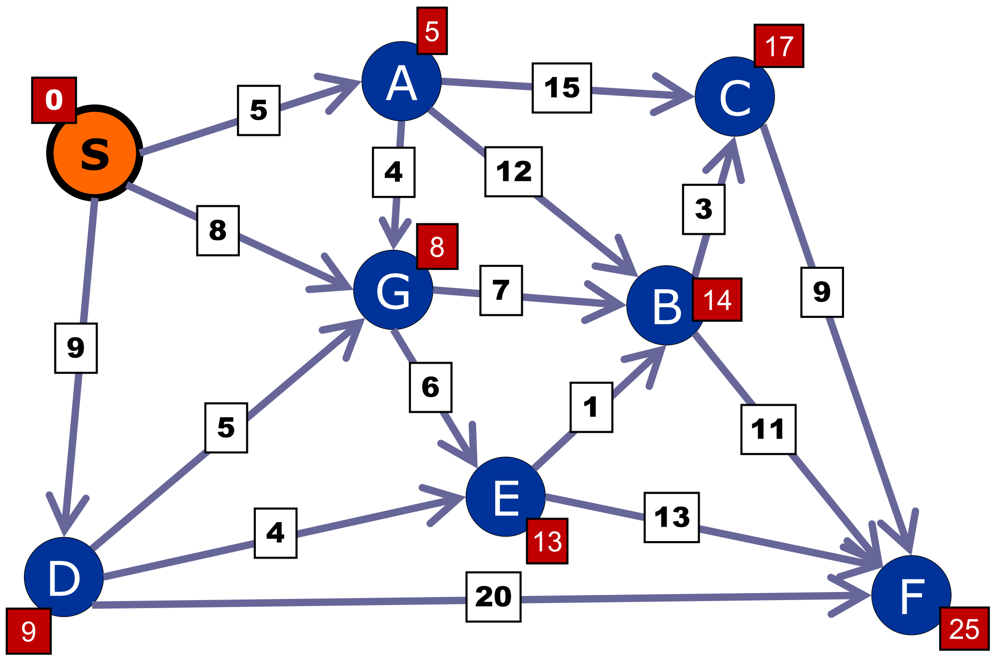
Step 6: Remove E and relax.

Step 7: Remove B and relax.

Step 8: Remove C and relax.

# Dijkstra's algorithm

| Vertex | Dist. |
| --- | --- |
| | |




Step 6: Remove E and relax.

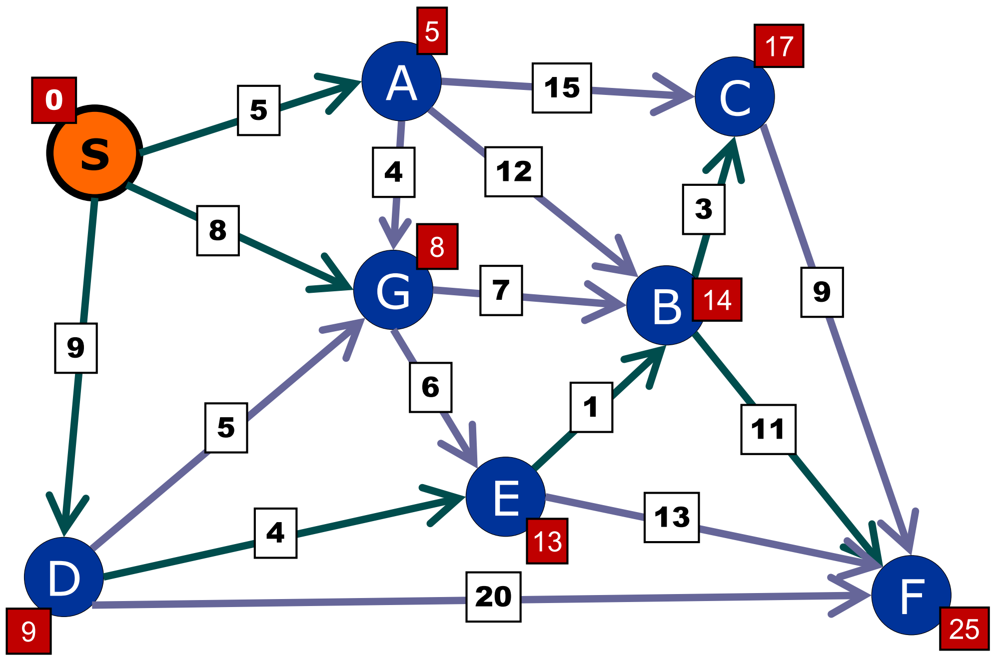
Step 7: Remove B and relax.

Step 8: Remove C and relax.

Step 9: Remove F and relax.

# Dijkstra's algorithm

| Vertex | Dist. |
| --- | --- |
| | |




Step 6: Remove E and relax.

Step 7: Remove B and relax.

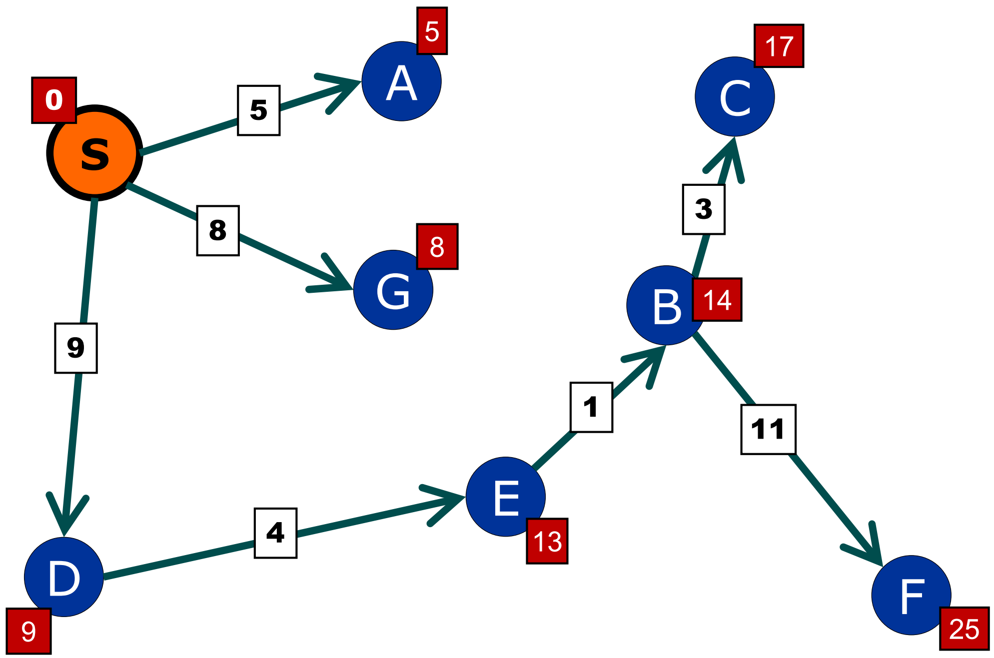
Step 8: Remove C and relax.

Step 9: Remove F and relax.

Done!

# Dijkstra's algorithm

| Vertex | Dist. |
|---|---|
| | |

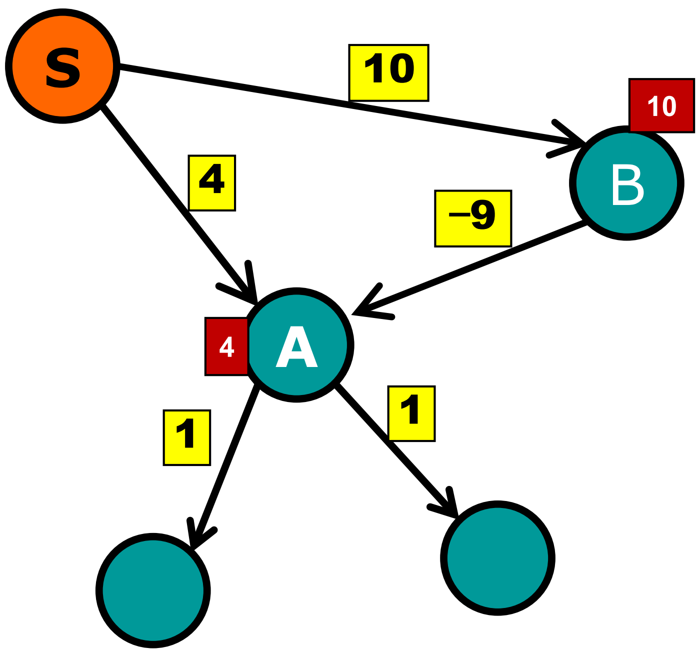



Step 10: Enjoy your Shortest-Path Tree. ☺

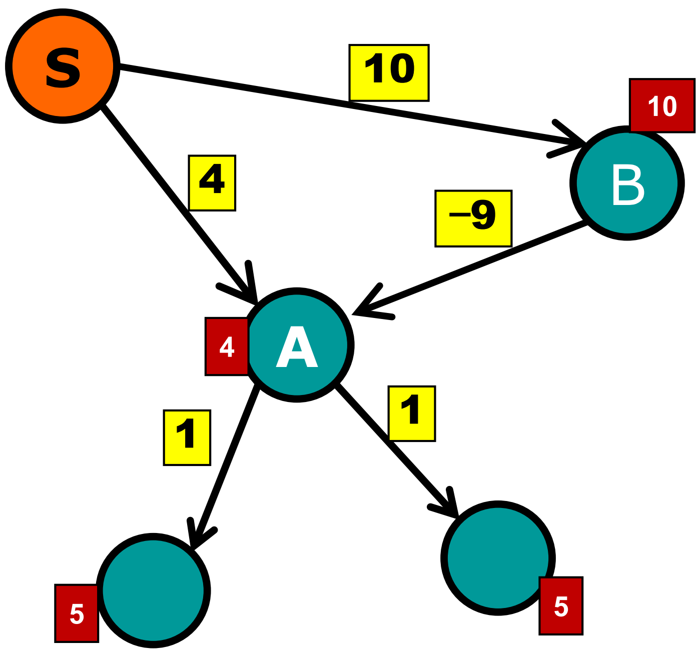
# Special Cases

| Condition | Algorithm | Time Complexity |
|---|---|---|
| No Negative Weight Cycles | Bellman-Ford Algorithm | $O(VE)$ |
| On Unweighted Graph (or equal weights) | BFS | $O(V+E)$ |
| No Negative Weights | Dijkstra’s Algorithm | |
| Negative Weights | Modified Dijkstra’s Algorithm | |
| On Tree | BFS / DFS | $O(V)$ |
| On DAG | Dynamic Programming | |

# Negative Weights



# Negative Weights




**Step 1:** Remove A.
Relax A.
Mark A done.

…

**Step 4:** Remove B.
Relax B.
Mark B done.

**Oops:** We need to update A.

Dijkstra's algorithm does not work on graphs with negative weights

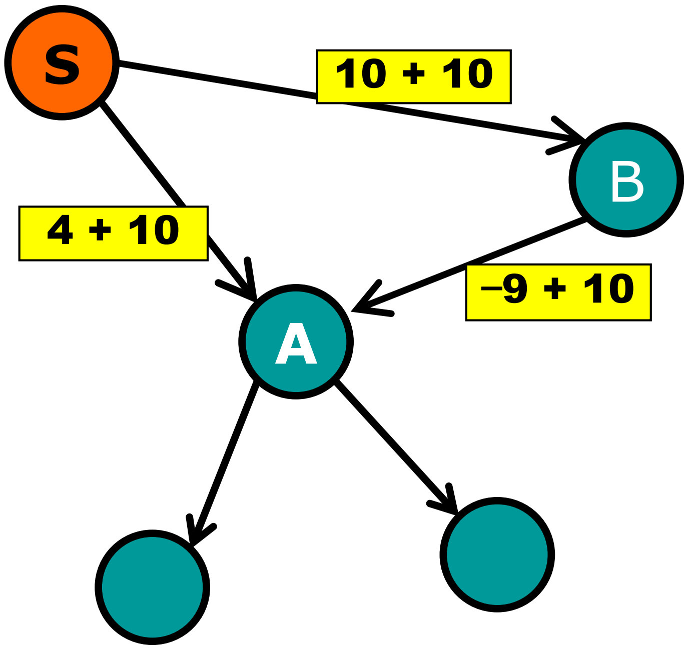
# Modified Dijkstra's Algorithm

- Can be used for graphs with at least one negative weight edge
- Dijkstra's algorithm can also be implemented differently. The O((**V**+**E**) log **V**) **Modified Dijkstra's algorithm** can be used for directed weighted graphs that may have negative weight edges but no negative weight cycle.
- Such input graph appears in some practical cases, e.g. travelling using an **electric car** that has battery and our objective is to find a path from source vertex **s** to another vertex that minimizes overall **battery usage**. As usual, during acceleration (or driving on flat/uphill road), the electric car **uses** (positive) energy from the battery. However, during braking (or driving on downhill road), the electric car **recharges** (or use negative) energy to the battery. There is no negative weight cycle due to kinetic energy loss.

# Modified Dijkstra's Algorithm

- The key idea is the modification done to C++ STL priority_queue to allow it to perform the required 'DecreaseKey' operation efficiently, i.e. in O(log **V**) time.
- The technique is called *'Lazy Update*' - leave the 'outdated/weaker/bigger-valued information' in the Min Priority Queue instead of deleting it straightaway. As the items are ordered from smaller values to bigger values in a Min PQ, we are guaranteeing ourselves that we will encounter the smallest/most-up-to-date item first before encountering the weaker/outdated item(s) later - which can be easily ignored.
- Refer to Visualgo for example

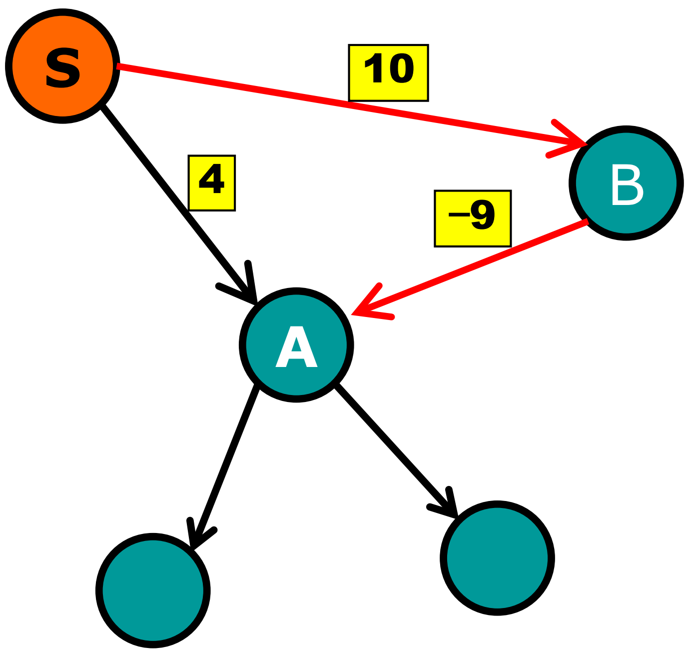
# Negative Weights




Can we re-weight the edges with some constant (10)?

A. yeah!
B. Nope.. wouldn't work.
C. 

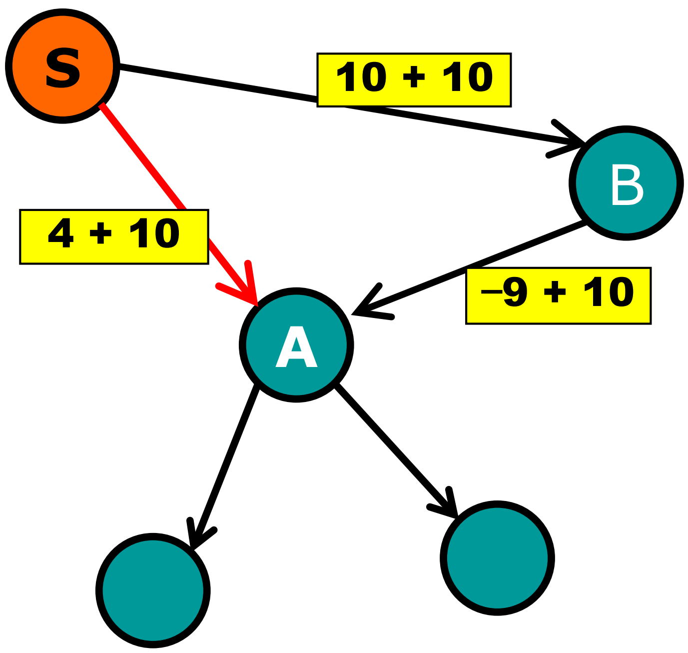
# Reweighting?




Path S-B-A: 1

Path S-A: 4

# Reweighting?

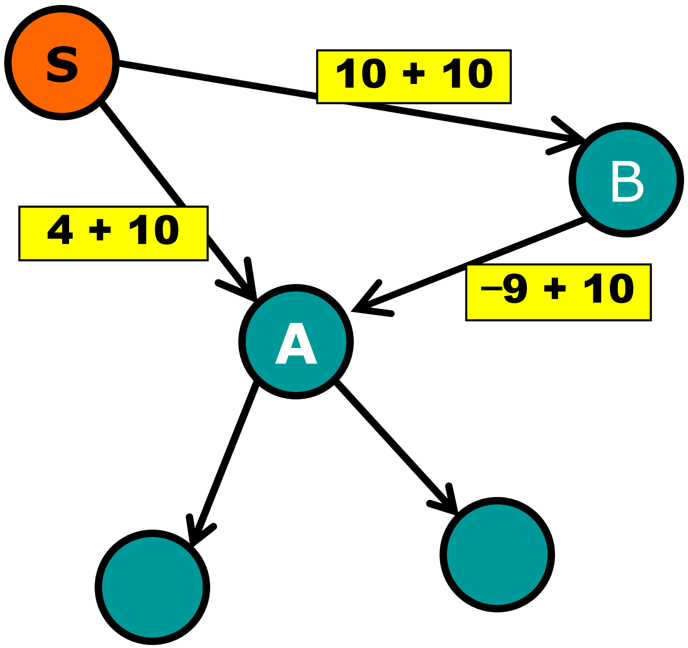



Path S-B-A: 21

Path S-A: 14

**The shortest path is no longer preserved!**

# Negative Weights




Can we re-weight the edges with some constant?

A. yeah!

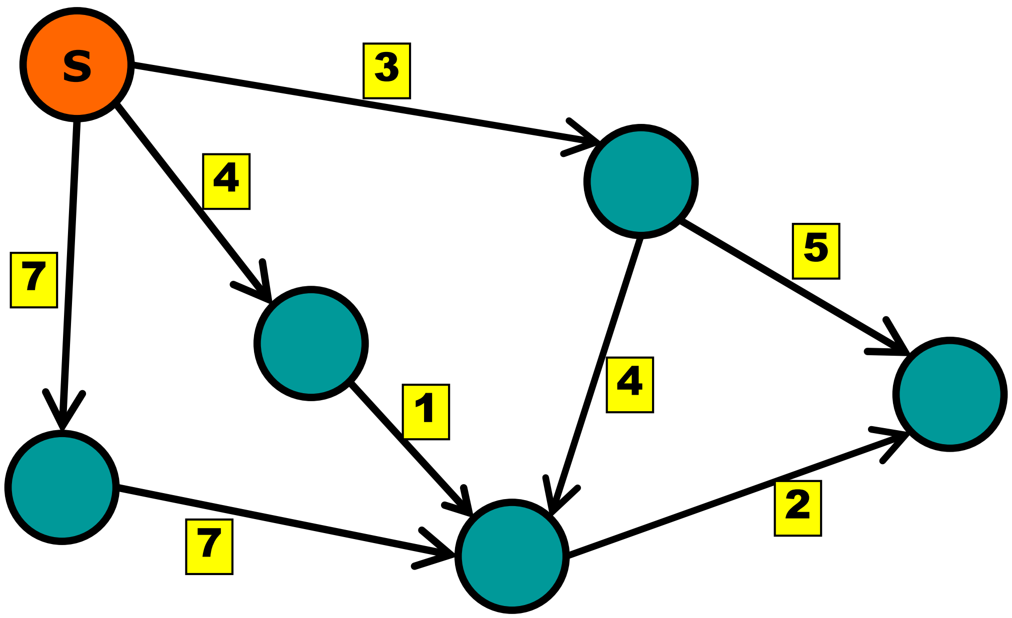
**B. Nope.. wouldn't work.**

C. 

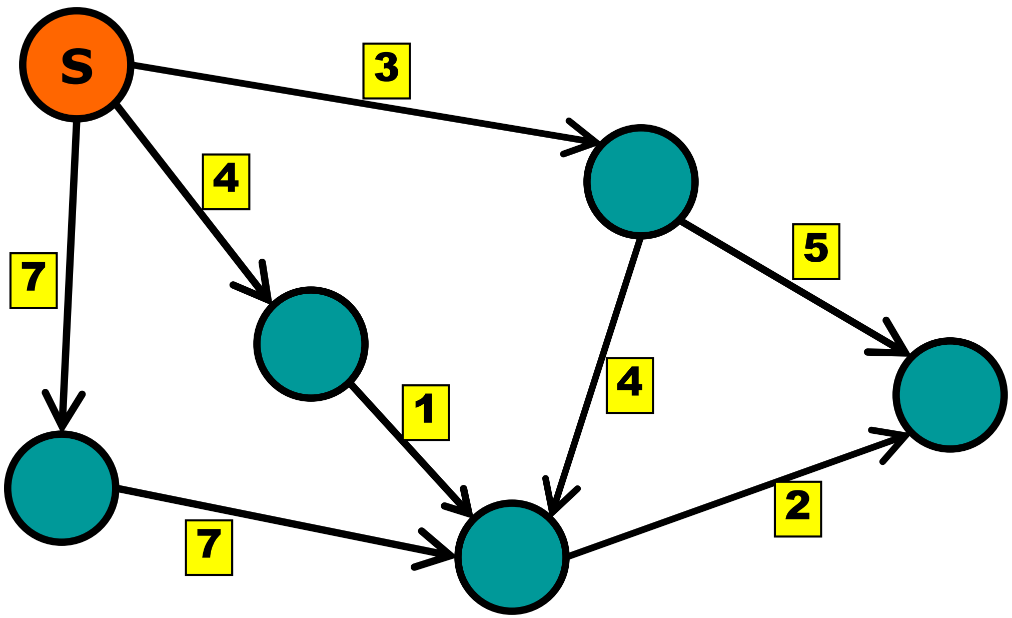
# Special Cases

| Condition | Algorithm | Time Complexity |
| --- | --- | --- |
| No Negative Weight Cycles | Bellman-Ford Algorithm | $O(VE)$ |
| On Unweighted Graph (or equal weights) | BFS | $O(V+E)$ |
| No Negative Weights | Dijkstra's Algorithm | $O((V+E)log\,V)$ |
| Negative Weights | Modified Dijkstra's Algorithm | $O((V+E)log\,V)$ |
| On Tree | BFS / DFS | $O(V)$ |
| On DAG | Dynamic Programming | |

# Directed Acyclic Graph (DAG)



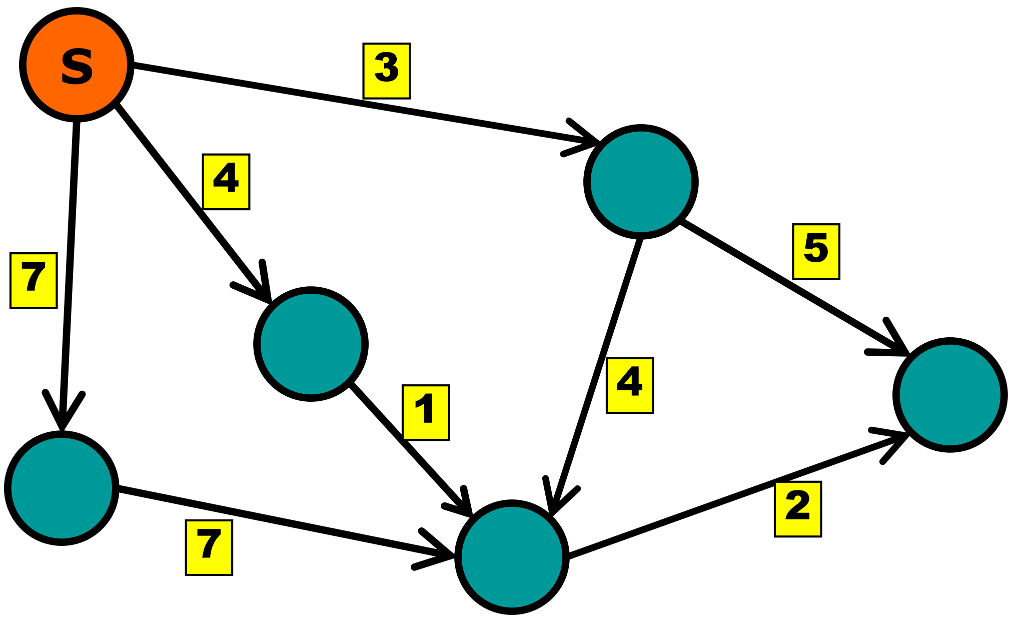
# Directed Acyclic Graph (DAG)




what relaxation order should we use for a DAG?

A. Random.. any order is the same.
B. BFS order
C. Topological sort order
D. 

# Directed Acyclic Graph (DAG)

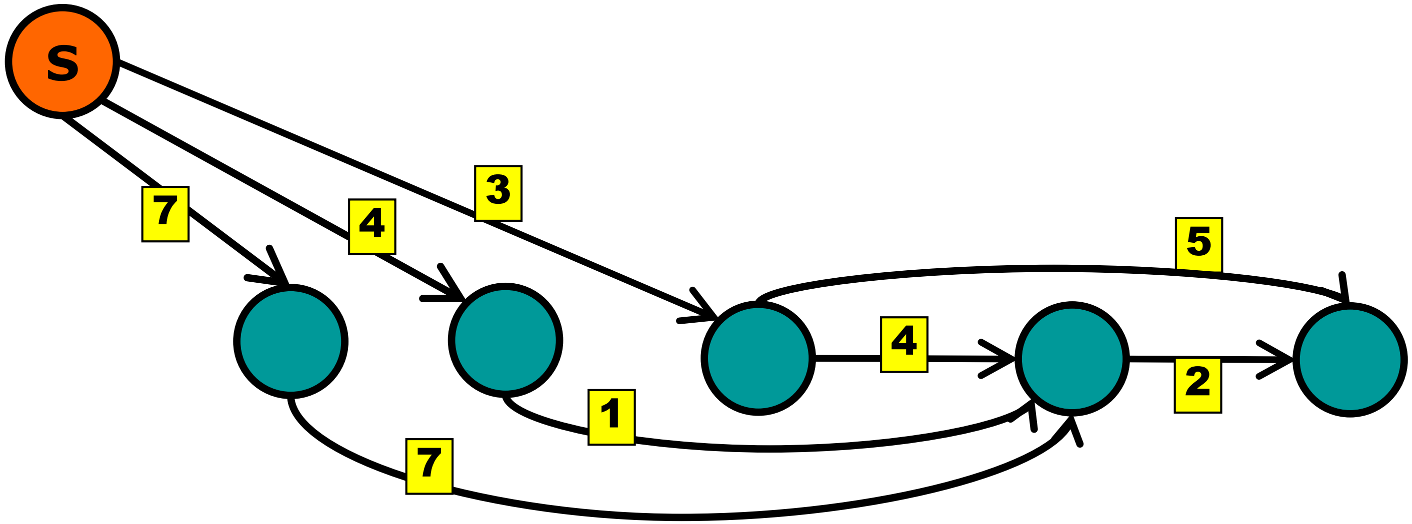



what relaxation order should we use for a DAG?

A. Random.. any order is the same.
B. BFS order
**C. Topological sort order**
D. 

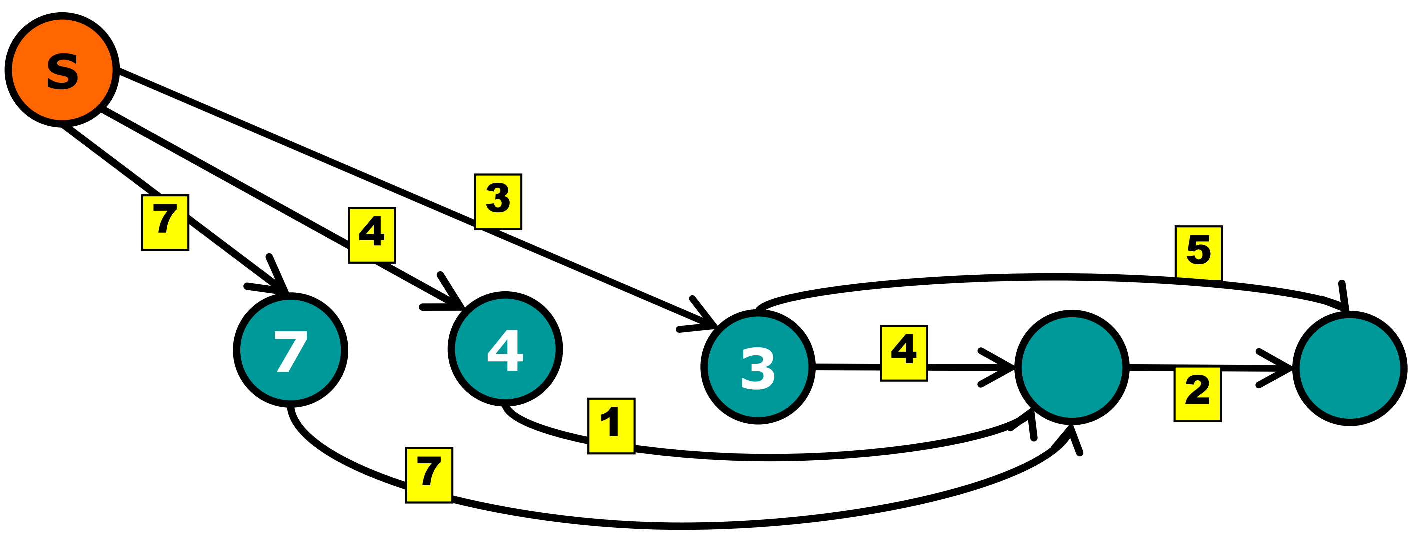
# Directed Acyclic Graph (DAG)

1. Topological sort
2. Relax in order.



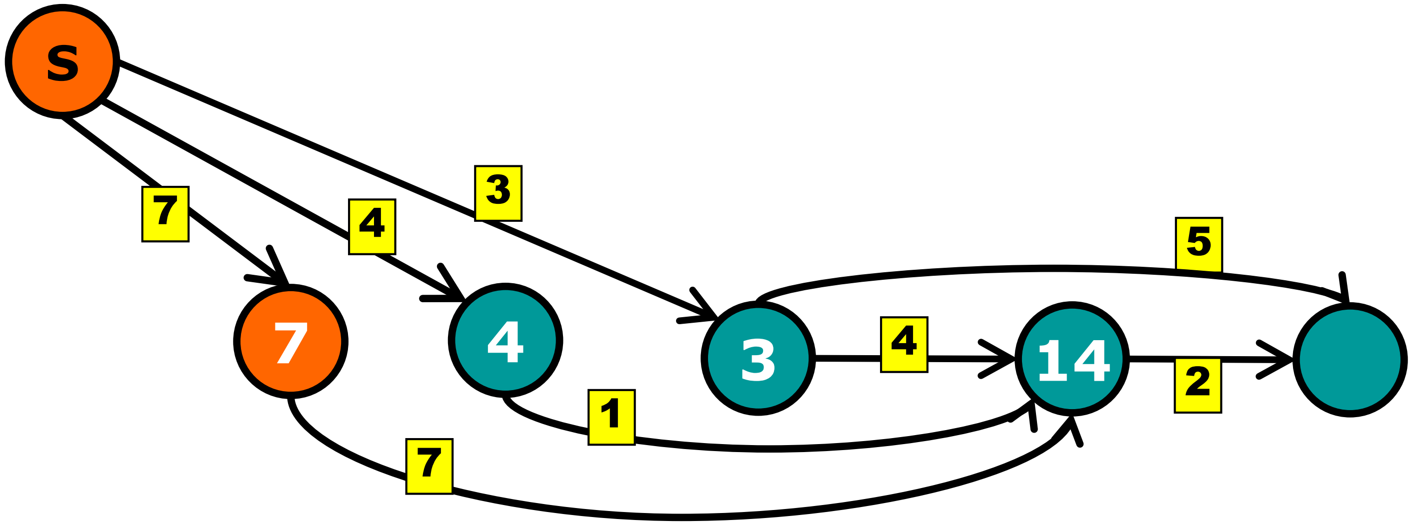
# Directed Acyclic Graph (DAG)

1. Topological sort
2. Relax in order.



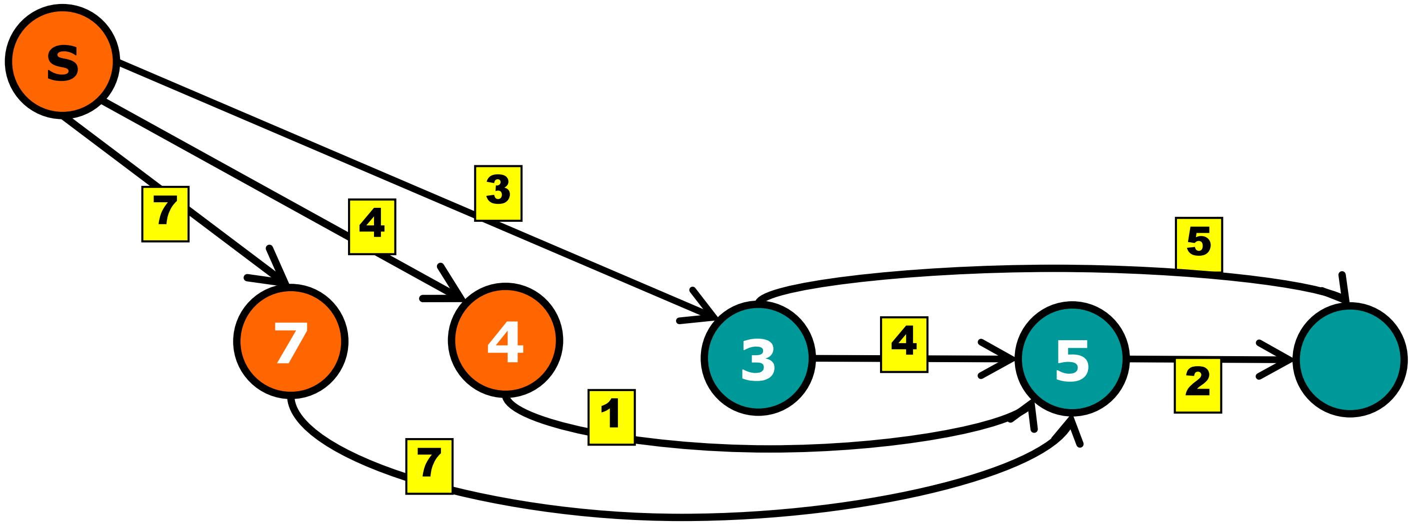
# Directed Acyclic Graph (DAG)

1. Topological sort
2. Relax in order.



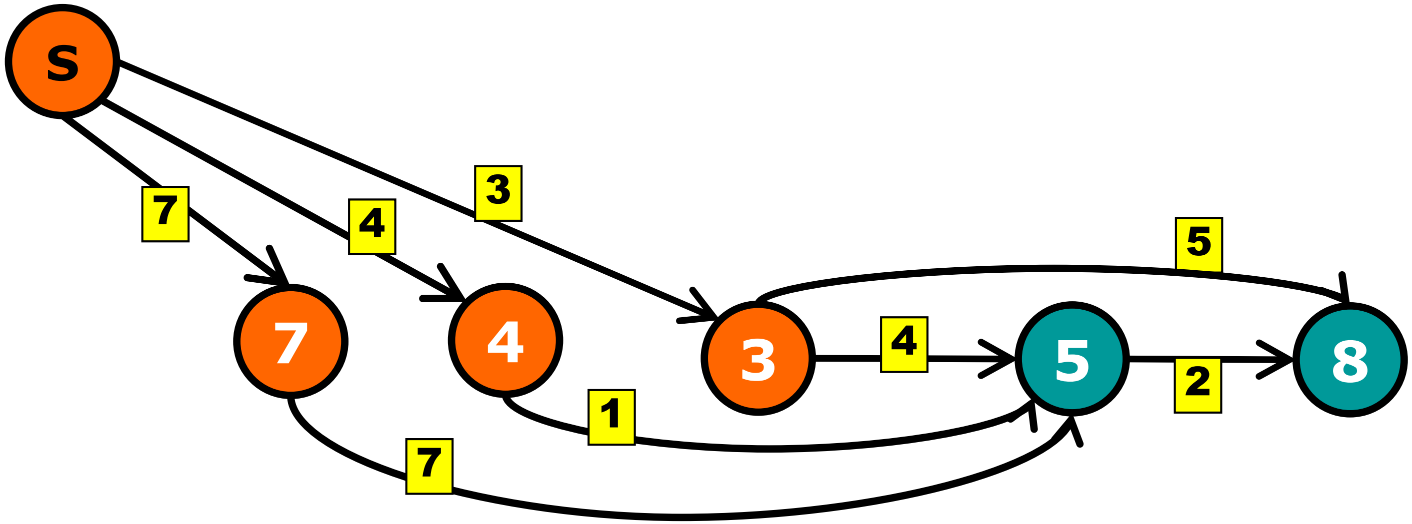
# Directed Acyclic Graph (DAG)

1. Topological sort
2. Relax in order.



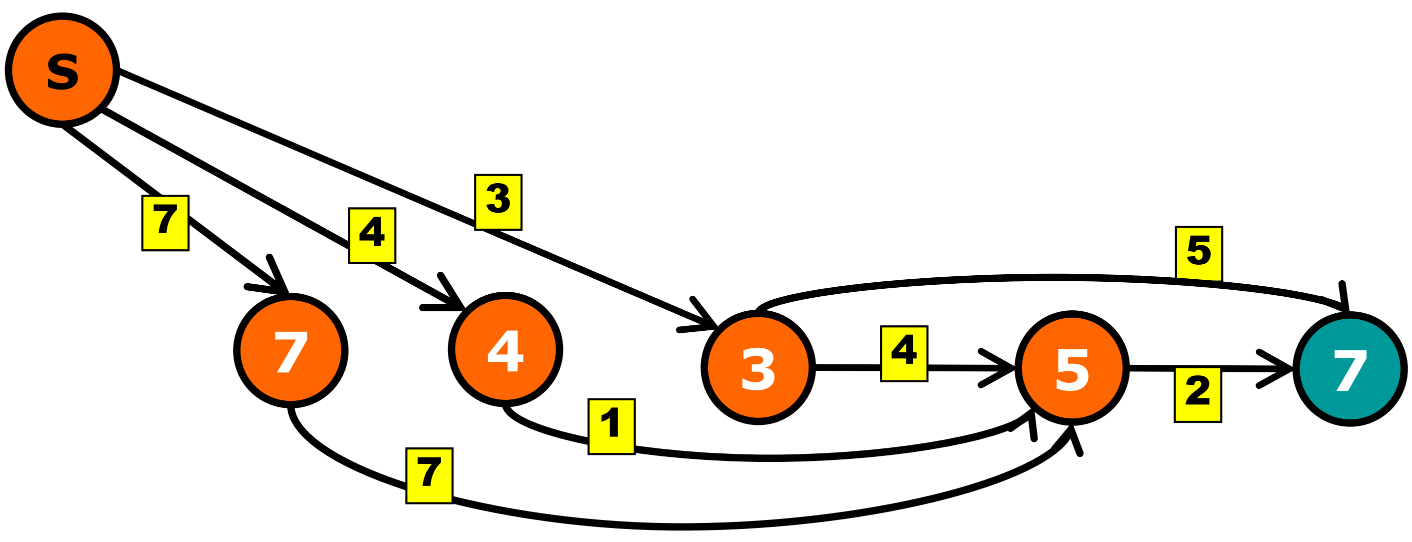
# Directed Acyclic Graph (DAG)

1. Topological sort
2. Relax in order.



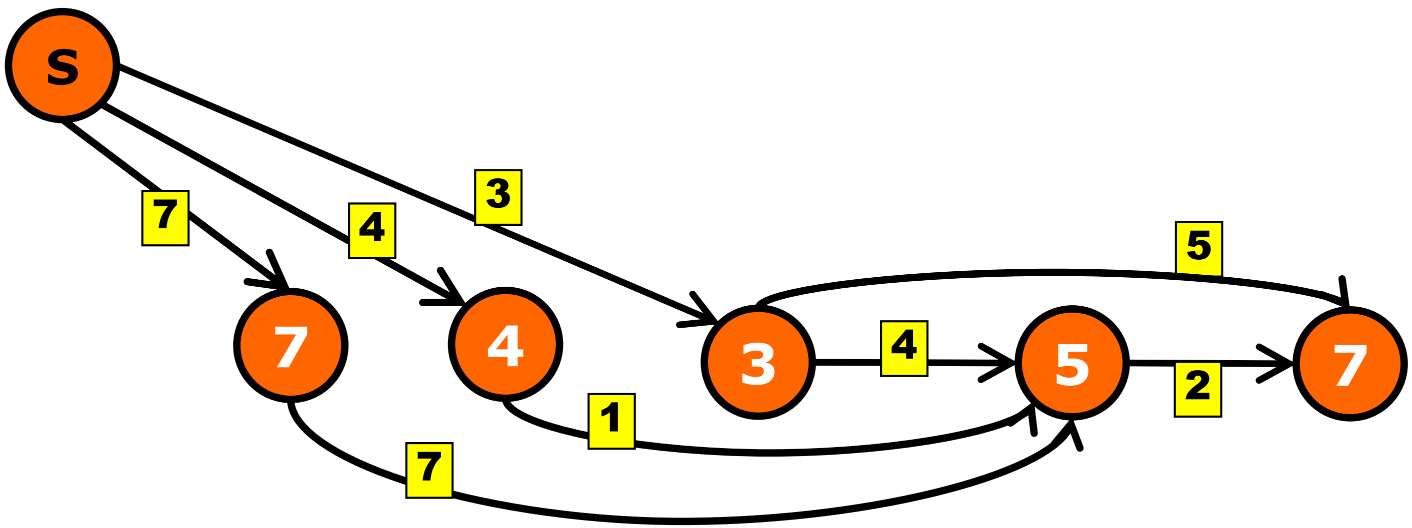
# Directed Acyclic Graph (DAG)

1. Topological sort
2. Relax in order.

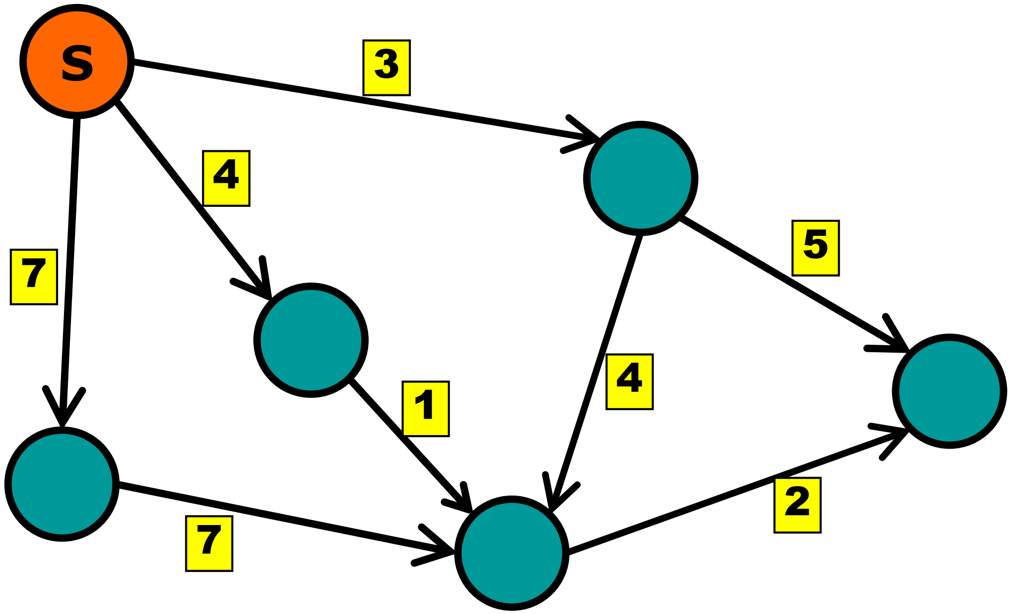


# Directed Acyclic Graph (DAG)

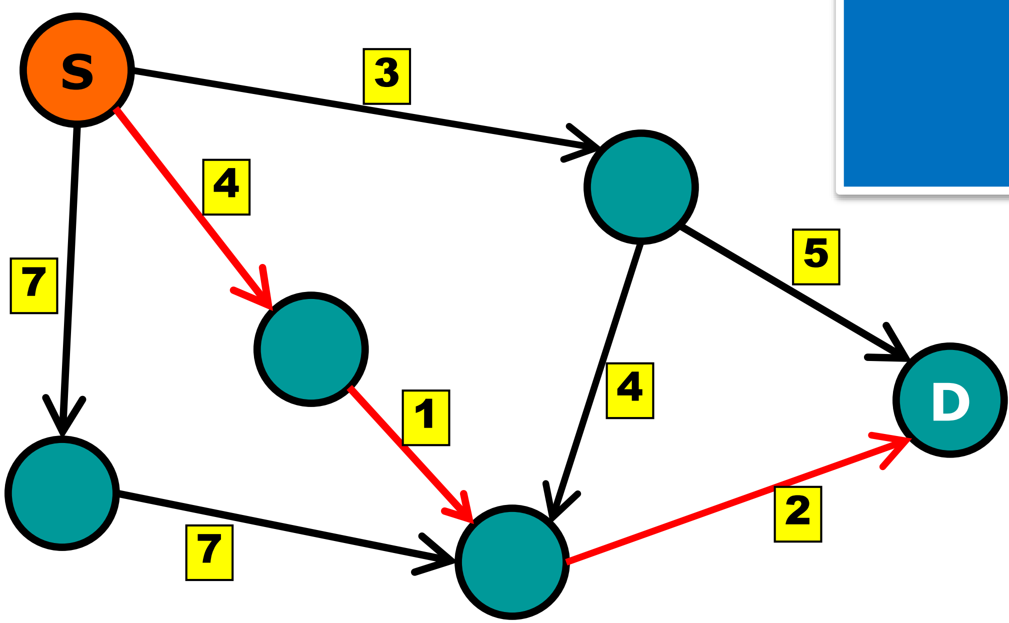
1. Topological sort
2. Relax in order.



# Why Topological Order?

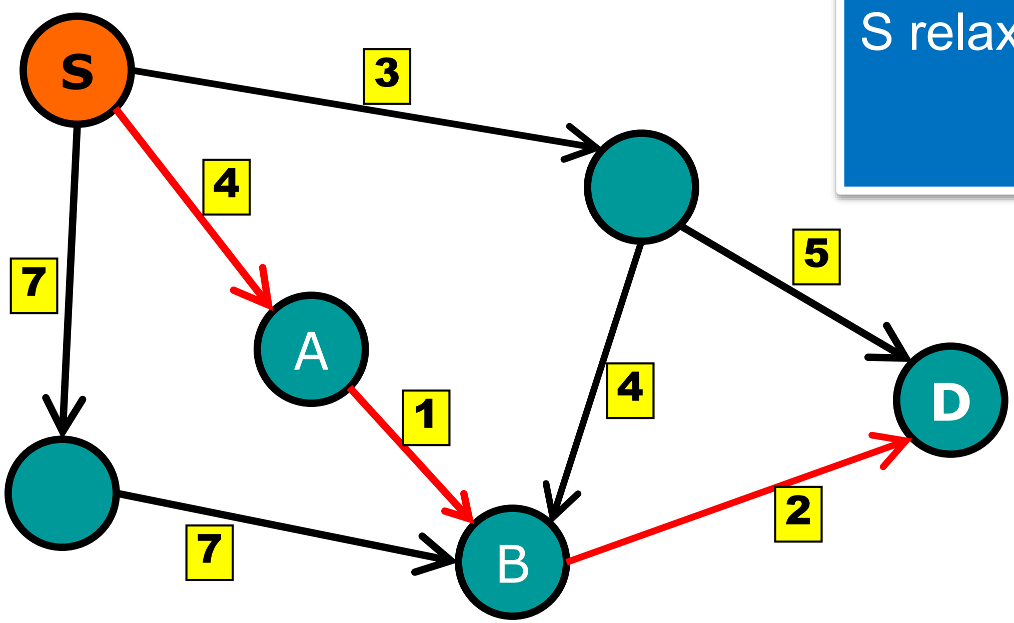


# Why Topological Order?




Fix S-D shortest path.

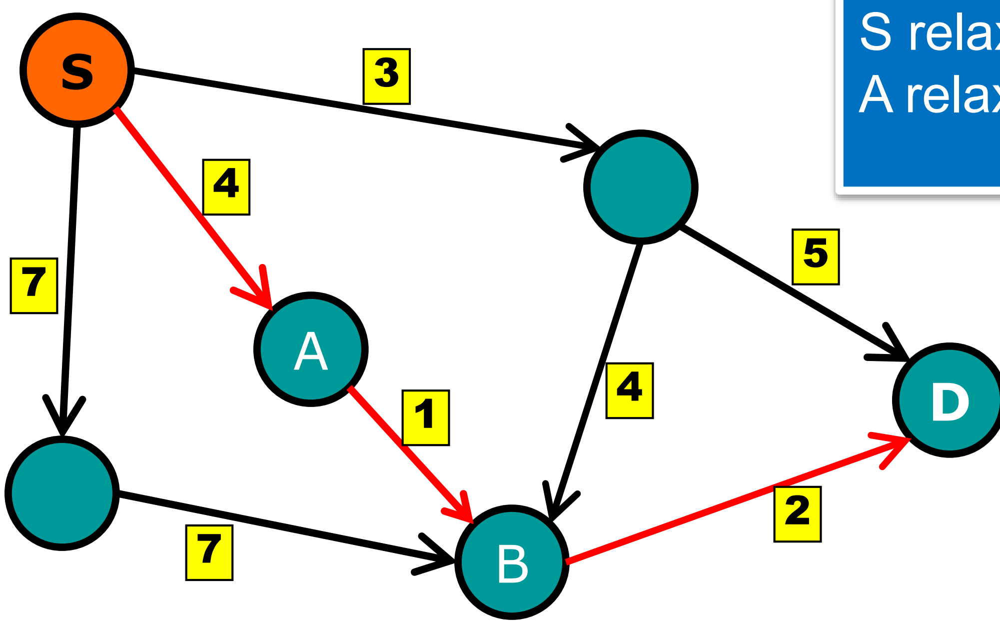
# Why Topological Order?




Fix S-D shortest path.

S relaxed before A.

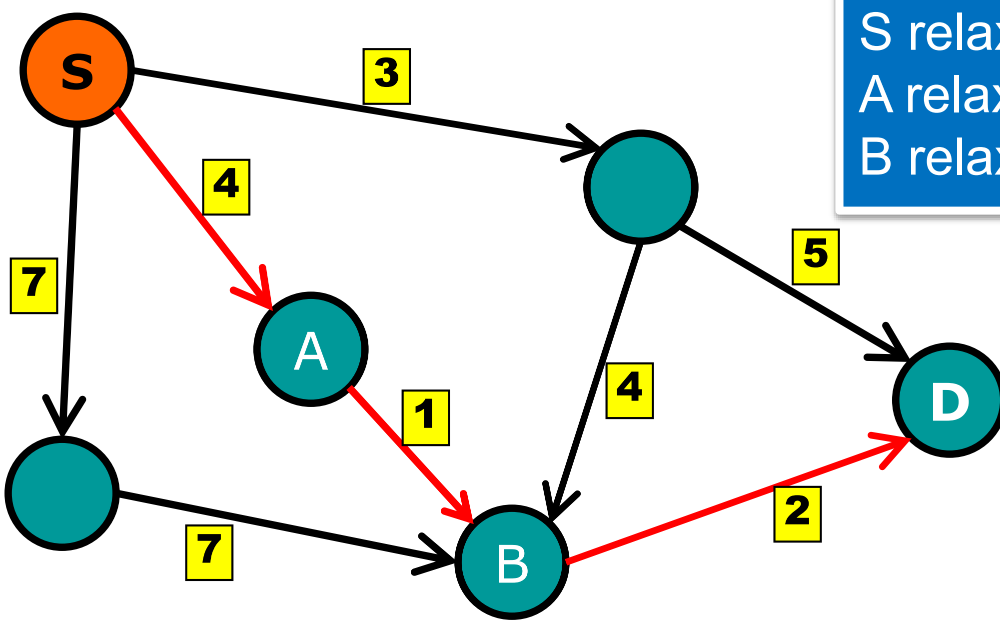
# Why Topological Order?




Fix S-D shortest path.

S relaxed before A.
A relaxed before B.

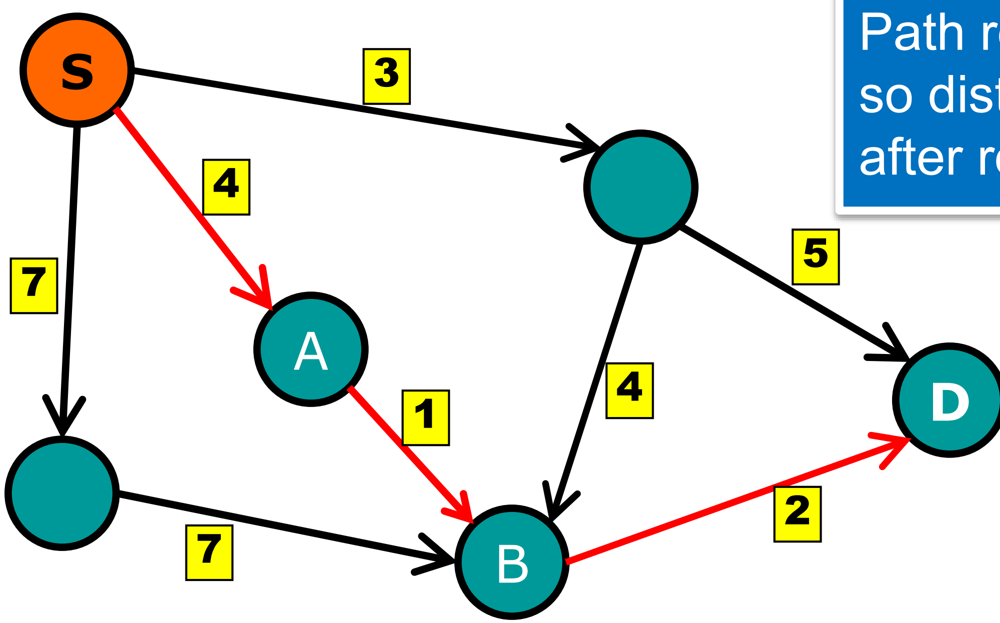
# Why Topological Order?




Fix S-D shortest path.

S relaxed before A.
A relaxed before B.
B relaxed before D.

# Why Topological Order?




Fix S-D shortest path.

Path relaxed in-order, so distance is correct after relaxation.

# Special Cases

| Condition | Algorithm | Time Complexity |
|---|---|---|
| No Negative Weight Cycles | Bellman-Ford Algorithm | $O(VE)$ |
| On Unweighted Graph (or equal weights) | BFS | $O(V + E)$ |
| No Negative Weights | Dijkstra's Algorithm | $O((V + E) log\, V)$ |
| Negative Weights | Modified Dijkstra's Algorithm | $O((V + E) log\, V)$ |
| On Tree | BFS / DFS | $O(V)$ |
| On DAG | Topological Sort (also called one-pass Bellman-Ford) | $O(V + E)$ |

# Summary

- Looked at various SSSP algorithms for special graphs that can run faster
- Described each special graph and the algorithm used
- Described Dijkstra’s algorithm and its modification
- Analyzed the computational complexity of Dijkstra’s algorithm

*Acknowledgement: some slides courtesy of Dr Harold Soh