LECTURE 13: SINGLE-SOURCE SHORTEST PATHS

Harold Soh
harold@comp.nus.edu.sg

A POLL:

Poll Everywhere

https://bit.ly/2LvG9bq

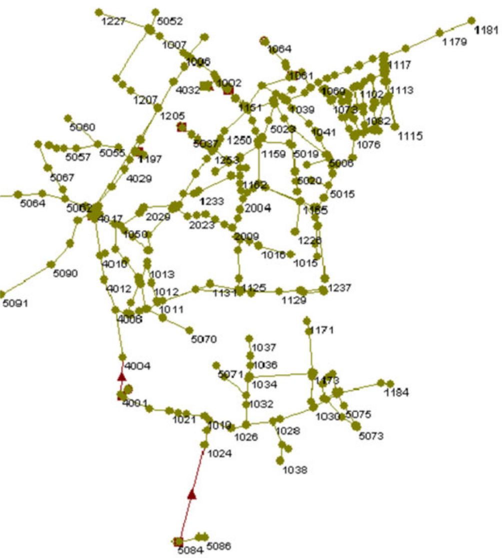
What wasn't clear in yesterday's lecture?

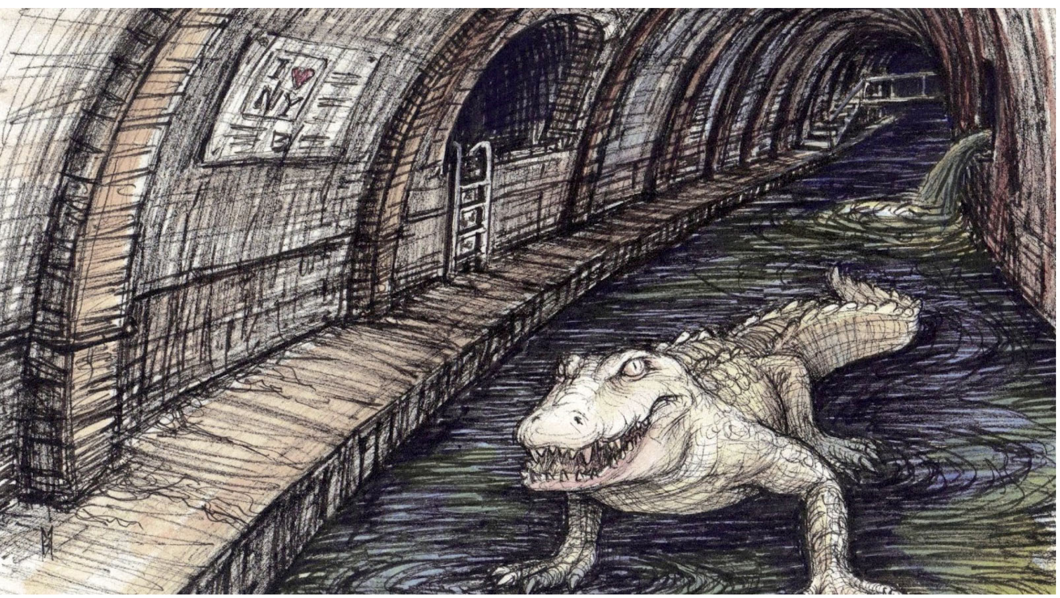
A. BFS and DFS
B. Topological Sort ("Breadth-first")
C. Topological Sort ("Depth-first")
D. B and C
E. A,B,C… All I don't understand. ☹
F. I understood it all. ☺

LEARNING OUTCOMES

By the end of the session, students should be able to:

- describe the **shortest path algorithm** for **unweighted graphs**
- explain the **Bellman-Ford algorithm**
- describe the time complexity of the **Bellman-Ford algorithm**
- Understand when **Bellman-Ford will fail**





PROBLEM: FINDING HERBERT!

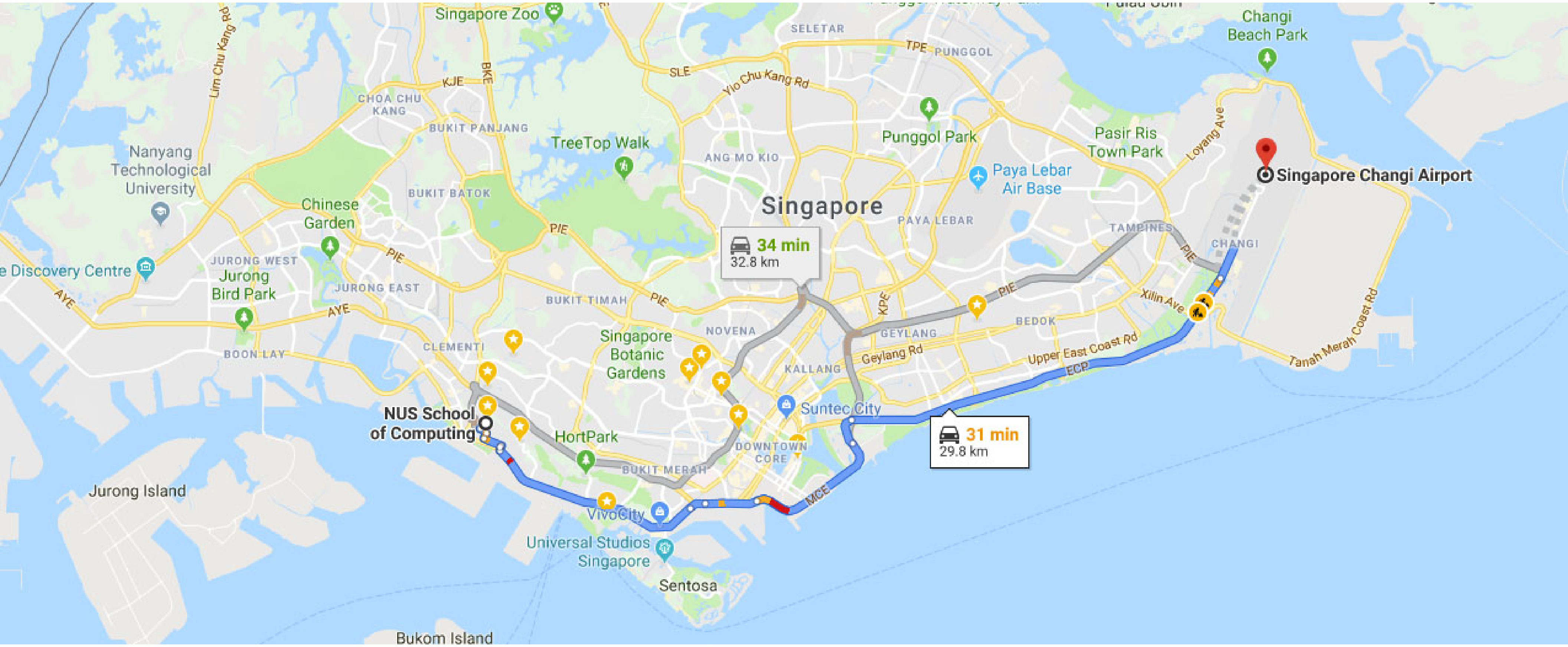
Herbert has gone missing!

Last sighting: in the sewer system.

How can we *systematically* search for Herbert… before he gets destroyed by an alligator?

The tunnels may have different lengths!

ROUTING YOUR VEHICLE

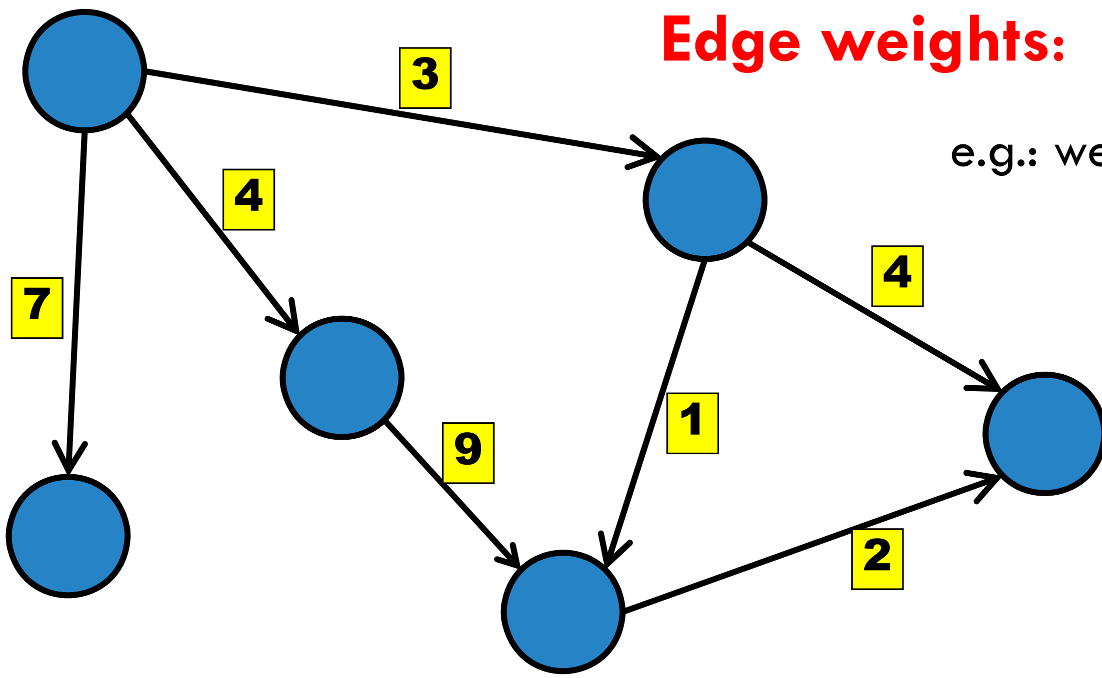


PATH TO ROUTE A PACKET OR A PACKAGE

Different edges have different costs:

- Time to send
- Cost to send
- Risk of going missing

WEIGHTED GRAPHS



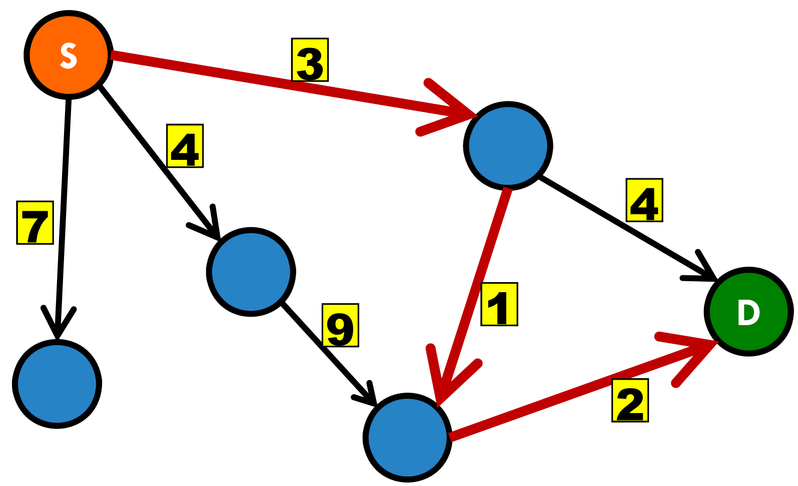
Edge weights: $w(e)\colon E \to \mathbb{R}$

e.g.: weight = distance

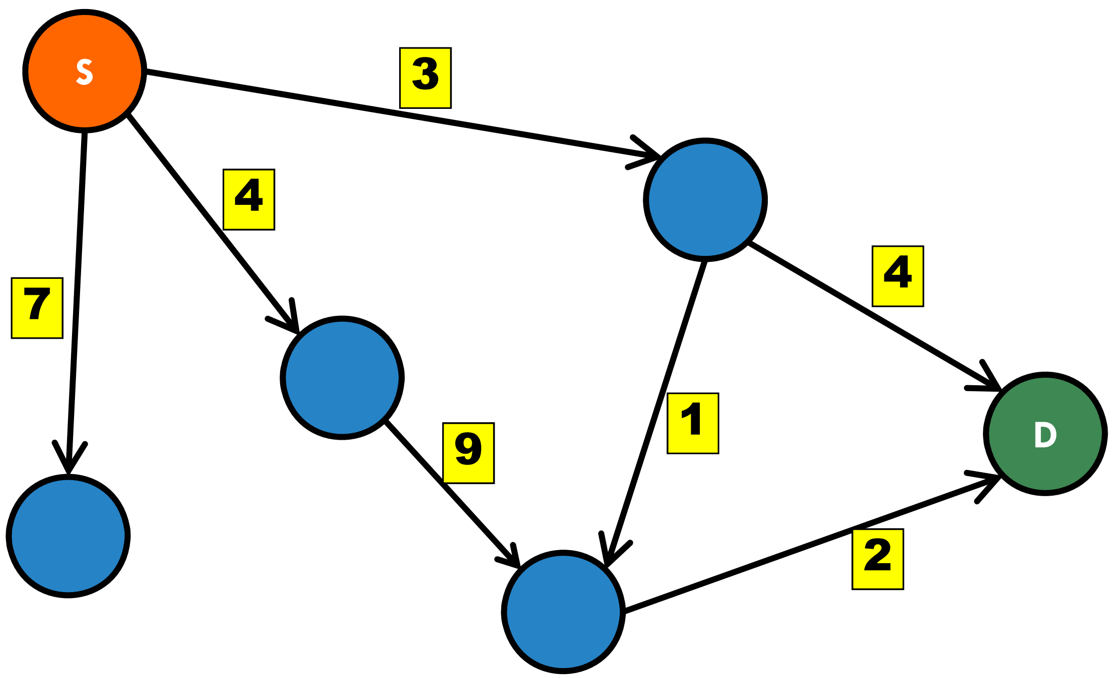
SHORTEST PATHS

Questions:

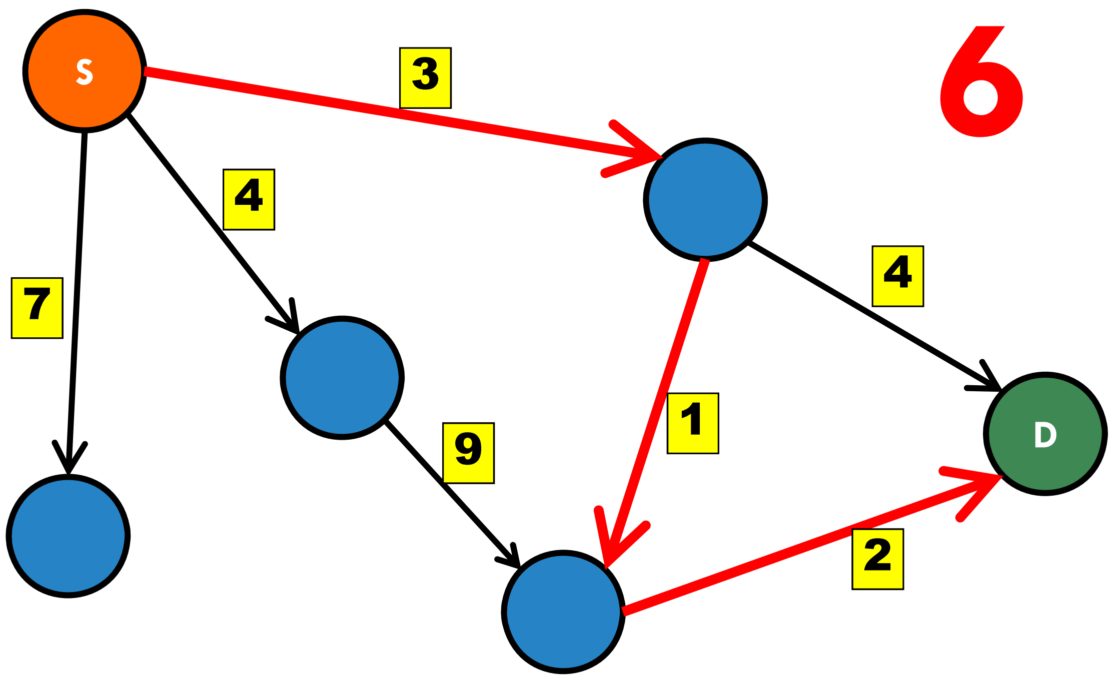
- How far is it from S to D?
- What is the shortest path from S to D?
- Find the shortest path from S to every node.
- Find the shortest path between every pair of nodes.



DISTANCE FROM THE SOURCE?

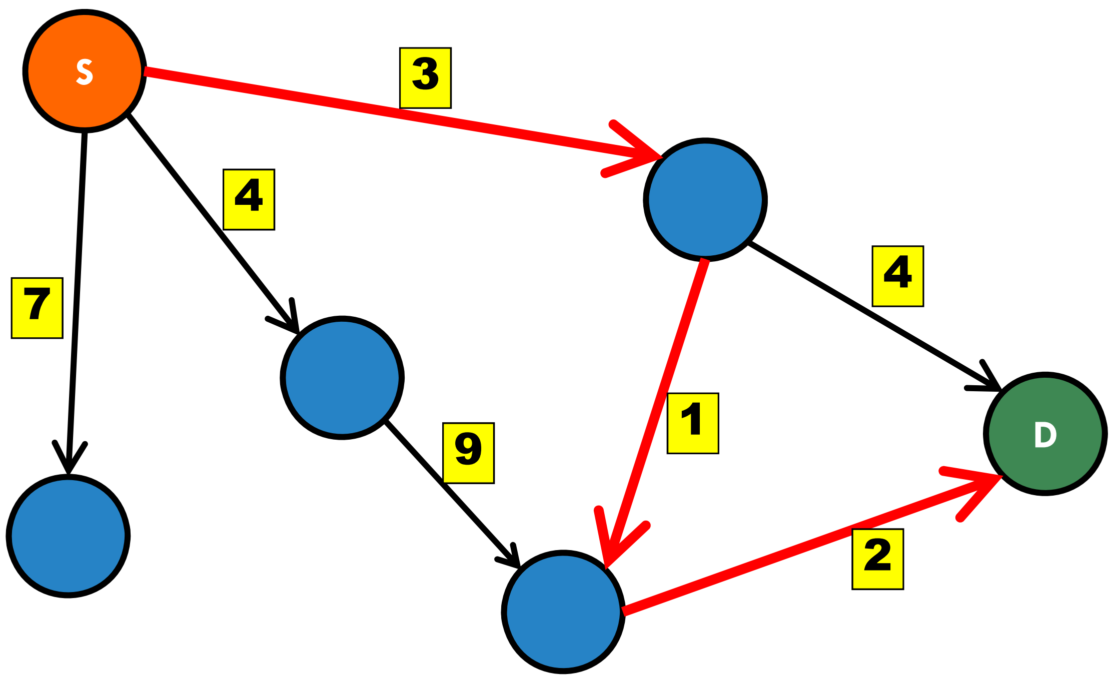


DISTANCE FROM THE SOURCE?

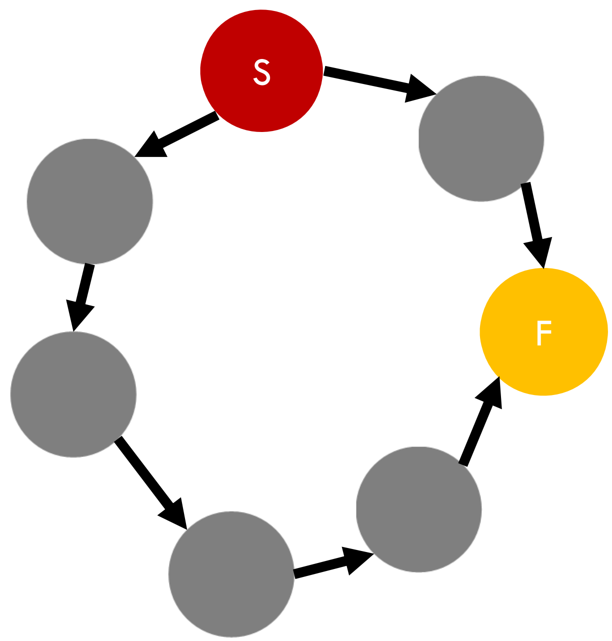


CAN WE USE **BFS**?

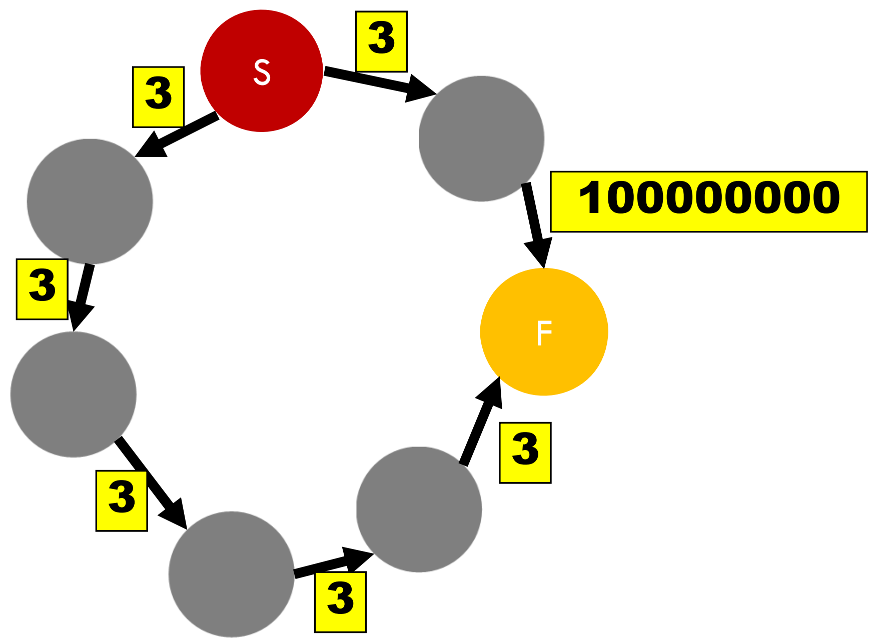
BFS finds minimum number of **HOPS** not minimum **DISTANCE**.



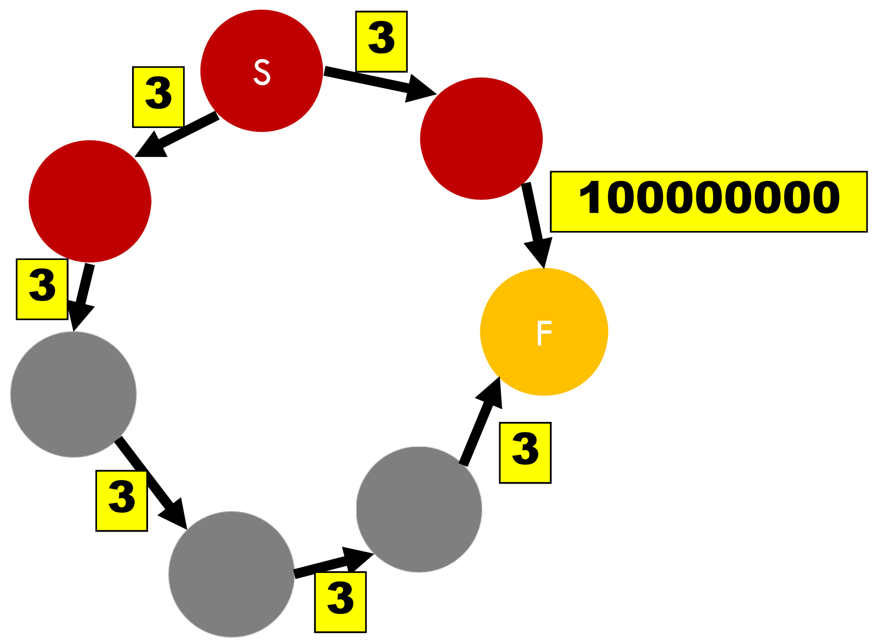
AN EXAMPLE: BFS



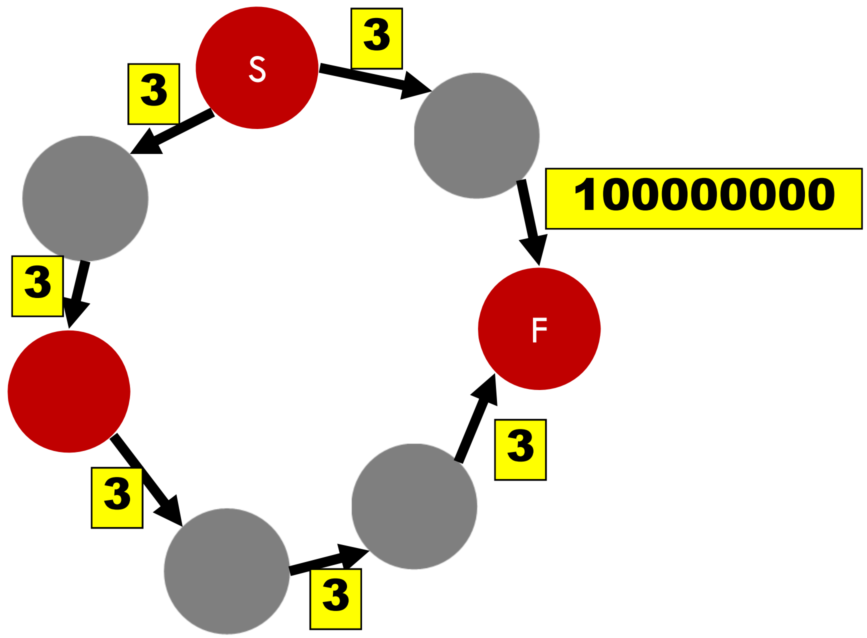
AN EXAMPLE: BFS



AN EXAMPLE: BFS

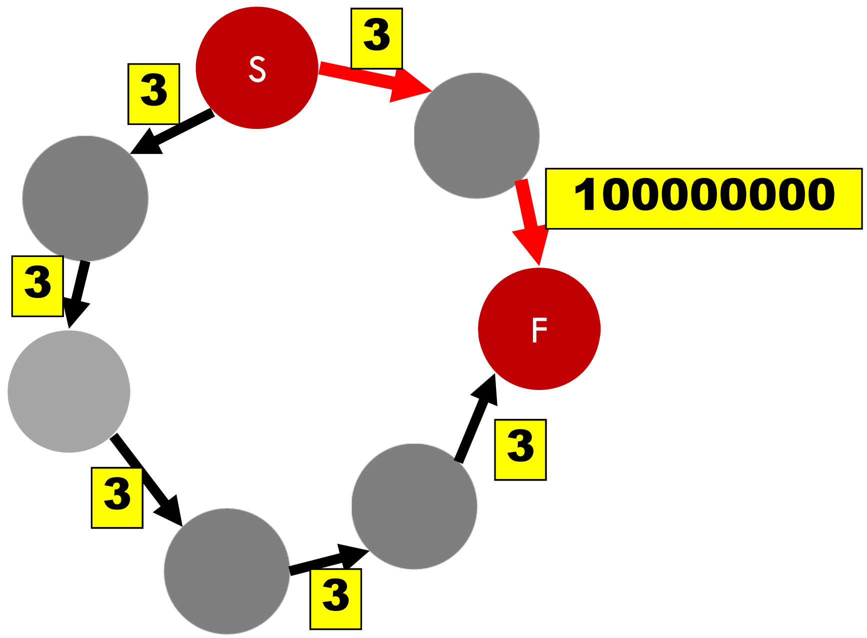


AN EXAMPLE: BFS



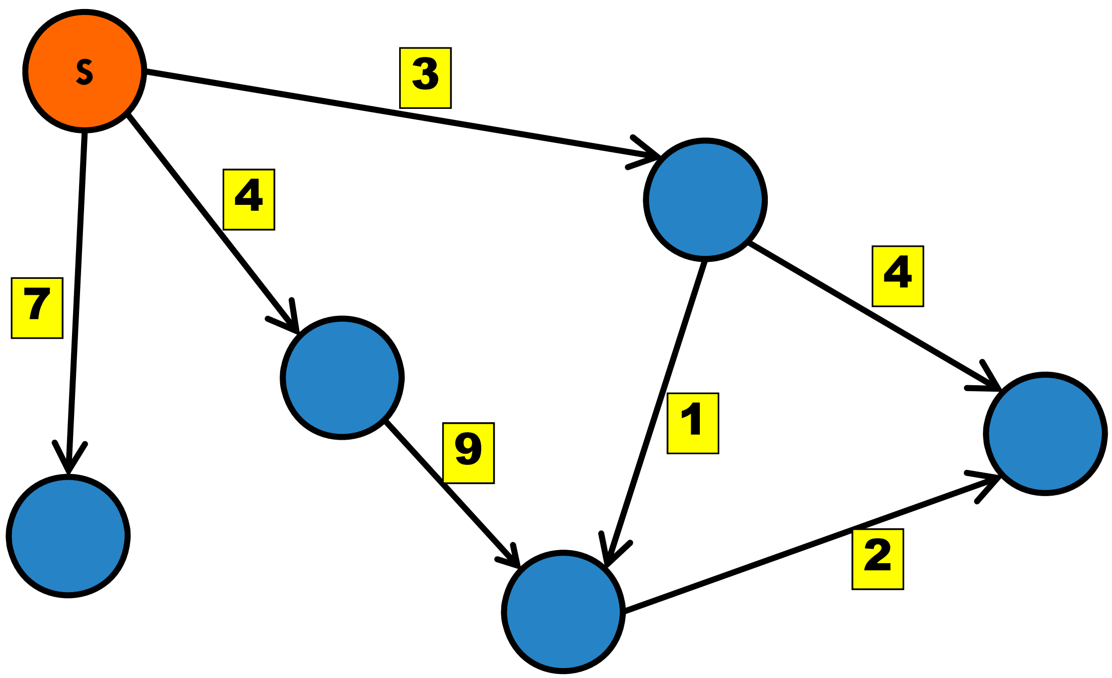
AN EXAMPLE: BFS

BFS finds minimum number of **HOPS** not minimum **DISTANCE**.

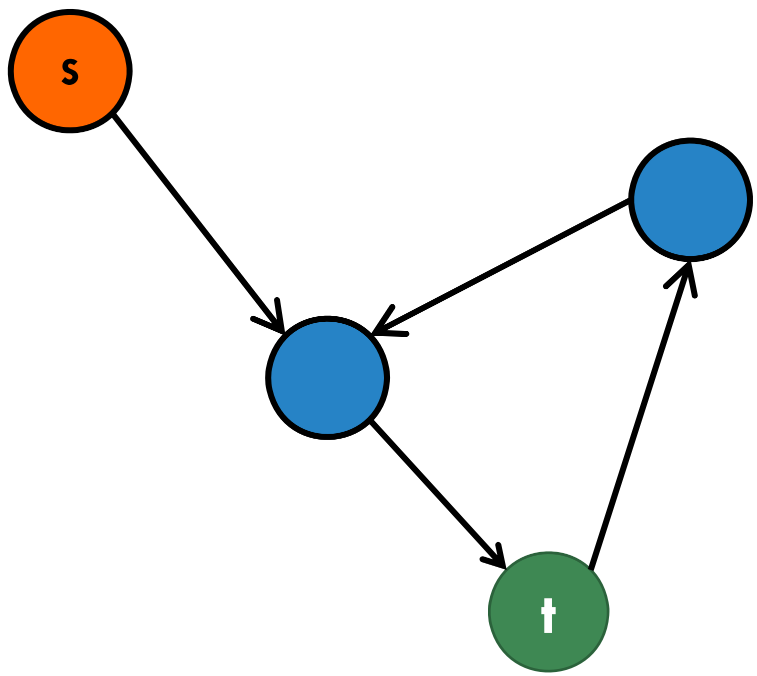


SHORTEST PATHS

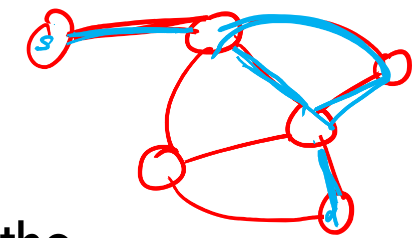
Notation: $\delta(u, v)$ = minimum distance from u to v



IF ALL WEIGHTS ARE POSITIVE, CAN THE SHORTEST PATH CONTAIN A CYCLE?



IF ALL WEIGHTS ARE POSITIVE, CAN THE SHORTEST PATH CONTAIN A CYCLE?



Lemma 1: If G = (V, E) contains **only positive weights** then the shortest path p from source vertex s to a vertex v must be a **simple path**.

e e e e

A **simple path** is defined as path $p = \{v_0, v_1, v_2, \dots, v_k\}$ where $(v_i, v_{i+1}) \in E, \forall\, 0 \le i \le (k-1)$ and there is **<u>no</u>** repeated vertex along this path.

WHY PROVE STUFF?

Algorithmic Thinking

Asymptotic Analysis

Software Carpentry

Theoretical CS

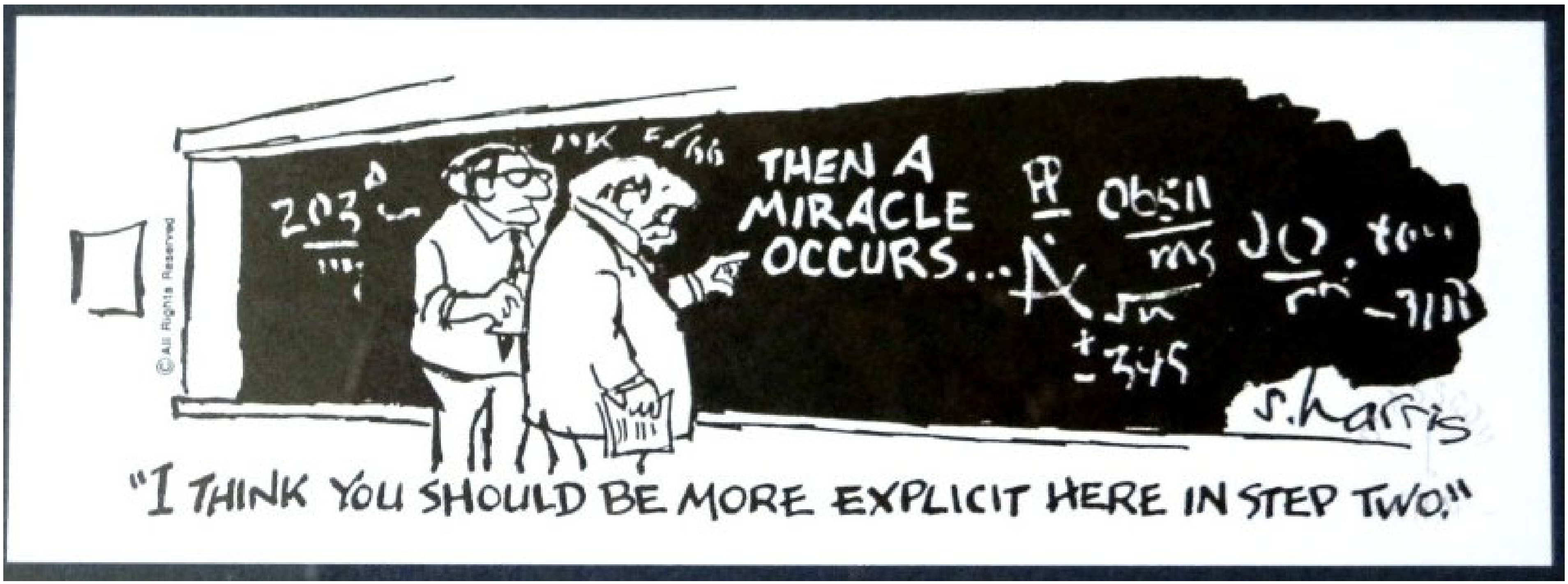
Programming Languages

Data Structures

Computer Architecture

Computer Security

WHY PROVE STUFF?



"I THINK YOU SHOULD BE MORE EXPLICIT HERE IN STEP TWO."

PROOF METHODS

Direct Proof

Mathematical Induction

Contradiction

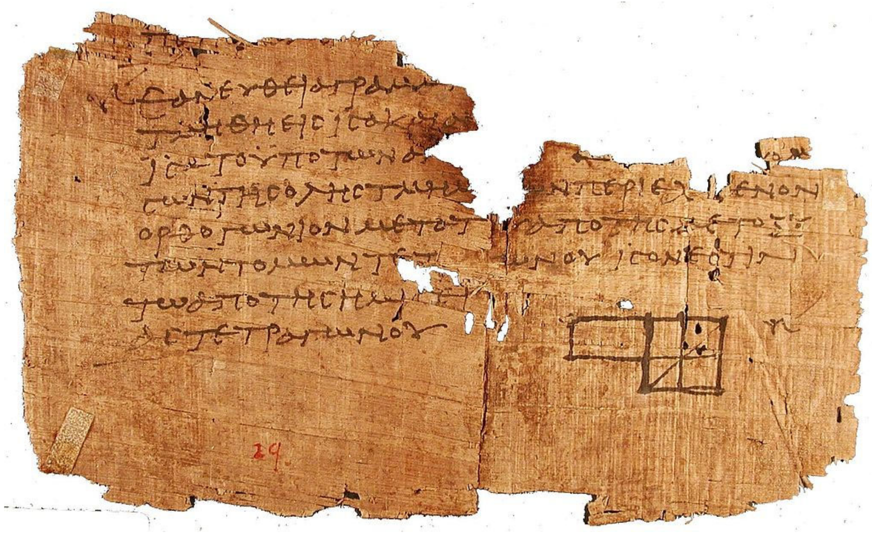
Contraposition

Proof by construction

Proof by exhaustion

Probabilistic proof

…



Page Fragment from Euclid's Elements of Geometry (75-125 A.D)

PROOF METHODS

Direct Proof

Mathematical Induction

Contradiction

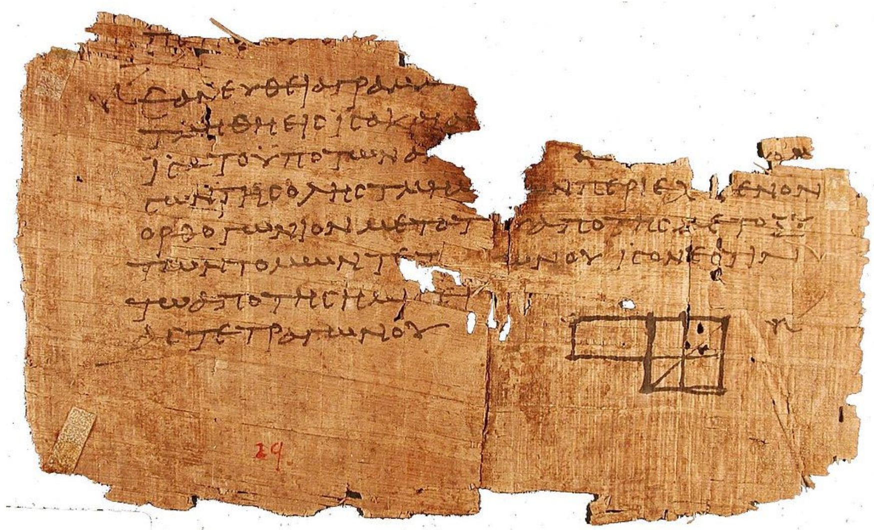
Contraposition

Proof by construction

Proof by exhaustion

Probabilistic proof

…



Page Fragment from Euclid's Elements of Geometry (75-125 A.D)

IF ALL WEIGHTS ARE POSITIVE, CAN THE SHORTEST PATH CONTAIN A CYCLE?

Lemma 1: If $G = (V, E)$ contains **only positive weights** then the shortest path p from source vertex s to a vertex v must be a **simple path.**

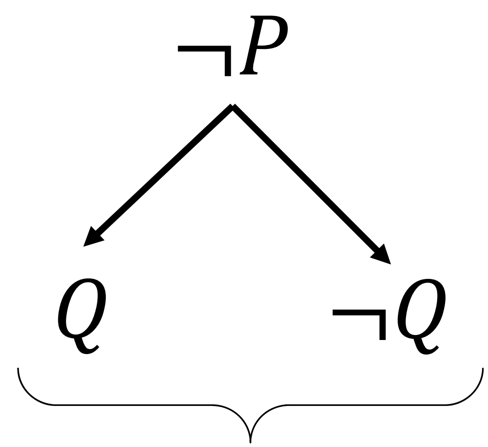
A **simple path** is defined as path $p = \{v_0, v_1, v_2, \dots, v_k\}$ where $(v_i, v_{i+1}) \in E, \forall\, 0 \le i \le (k-1)$ and there is **<u>no</u>** repeated vertex along this path.

PROOF SKETCH OF LEMMA 1

(By Contradiction)

Strategy:

- Assume some statement P to be *false*, i.e. not-P
- Show that if not-P, then two contradictory statements Q and not-Q are reached.
- Since both Q and not-Q cannot be true, not-P is false!
- So, P must be true



Contradiction!

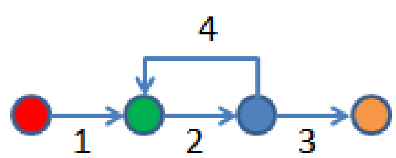
Lemma 1: If G = (V, E) contains **only positive weights** then the shortest path p from source vertex s to a vertex v must be a **simple path.**

PROOF SKETCH OF LEMMA 1

Suppose the shortest path p is **not** a simple path

Then p must contain *at least* one cycle

Suppose there is a cycle c in p with positive weight:



If we remove c then we will have a "shorter" shortest path.

Contradiction! (we said at the beginning that p is the shortest path)

Conclusion: p is a simple path.

IF ALL WEIGHTS ARE POSITIVE, CAN THE SHORTEST PATH CONTAIN A CYCLE?

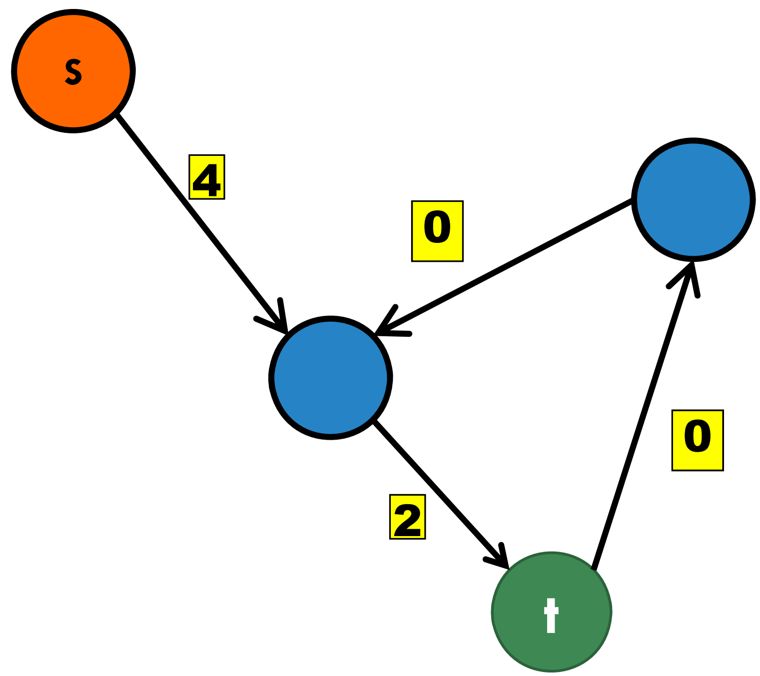
Lemma 1: If $G = (V, E)$ contains **only positive weights** then the shortest path p from source vertex s to a vertex v must be a **simple path.**

A **simple path** is defined as path $p = \{v_0, v_1, v_2, \dots, v_k\}$ where $(v_i, v_{i+1}) \in E, \forall\, 0 \le i \le (k-1)$ and there is **no** repeated vertex along this path.

HOW ABOUT NON-POSITIVE WEIGHTS?

What about 0 weight?

- 0-cycles can occur but can be removed.



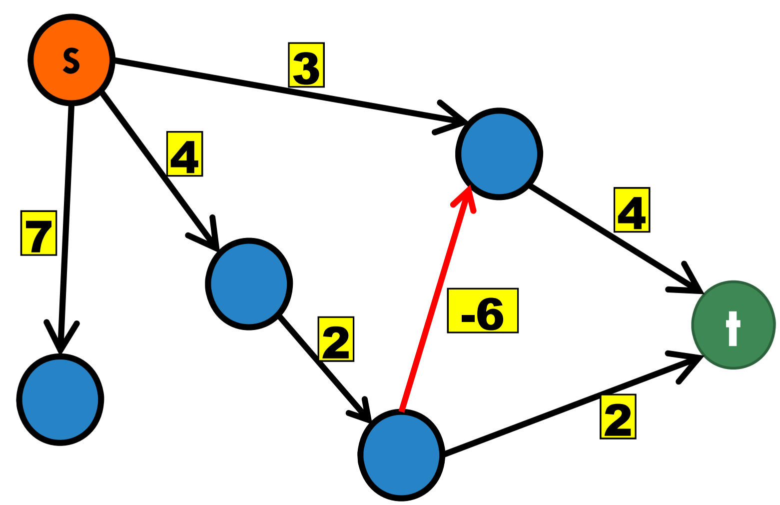
HOW ABOUT NON-POSITIVE WEIGHTS?

What about 0 weight?

- 0-cycles can occur but can be removed.

What about negative weights?

- Ok! Up to a point …



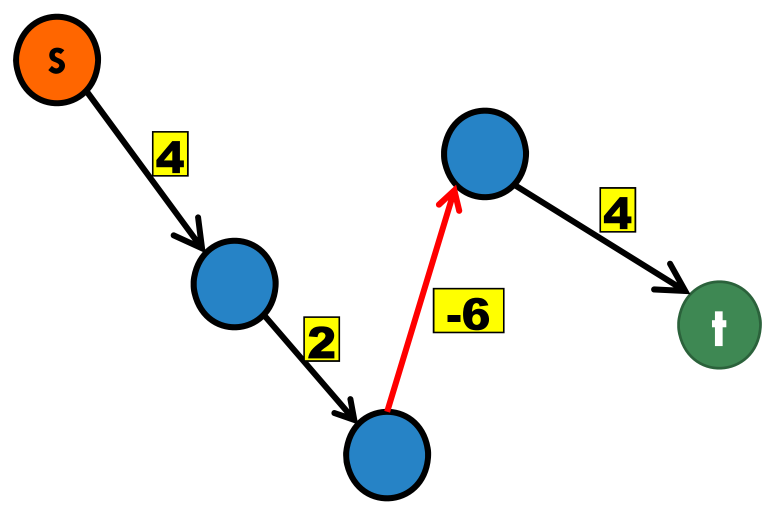
HOW ABOUT NON-POSITIVE WEIGHTS?

What about 0 weight?

- 0-cycles can occur but can be removed.

What about negative weights?

- Ok! Up to a point …



HOW ABOUT NON-POSITIVE WEIGHTS?

What about 0 weight?

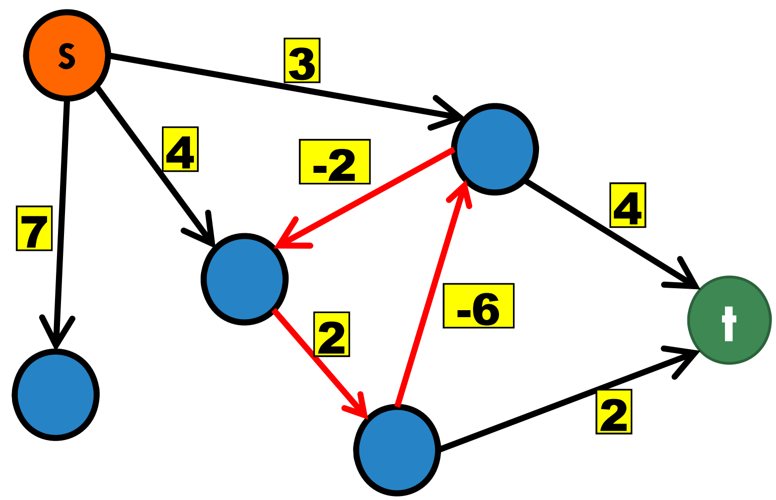
- 0-cycles can occur but can be removed.

What about negative weights?

- Ok!

But **no negative weight cycles**!

- Negative weight cycles make the problem ill-defined



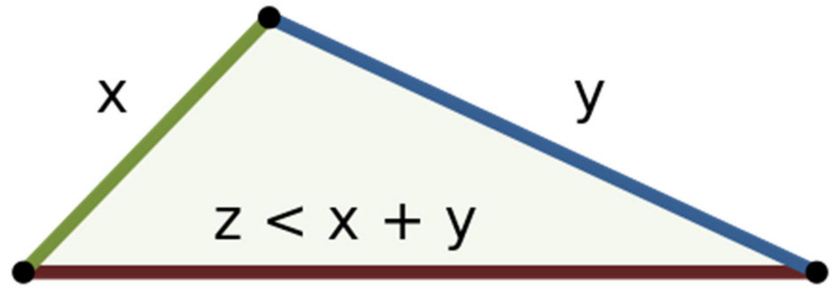
SIMPLE PATHS

Lemma 2: If $G = (V, E)$ contains **no negative weight cycles**, then the shortest path p from source vertex s to a vertex v is a **simple path.**

A **simple path** is defined as path $p = \{v_0, v_1, v_2, \dots, v_k\}$ where $(v_i, v_{i+1}) \in E, \forall\, 0 \le i \le (k-1)$ and there is **<u>no</u>** repeated vertex along this path.

This means that the shortest path can have *at most* $\underline{\quad \boldsymbol{|V|-1} \quad}$ edges

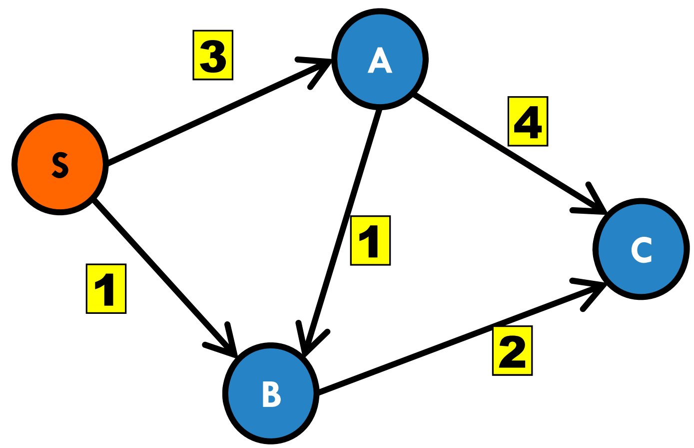
SHORTEST PATHS



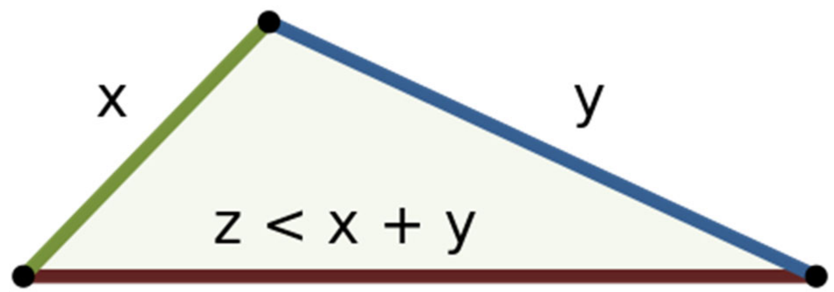
Lemma 3: Triangle Inequality.
For any edge (u, v)

$$\delta(s, v) \leq \delta(s, u) + w(u, v)$$

Proof Sketch (by contradiction):



SHORTEST PATHS



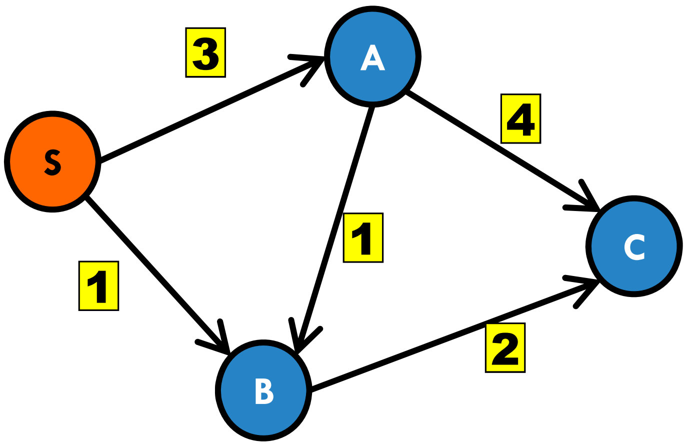
Lemma 3: Triangle Inequality.
For any edge (u, v)

$$\delta(s, v) \leq \delta(s, u) + w(u, v)$$

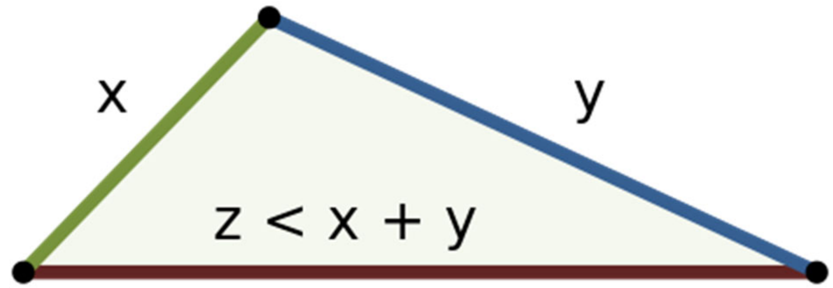
Proof Sketch (by contradiction):

Suppose the shortest path p has

$$\delta(s, v) > \delta(s, u) + w(u, v)$$



SHORTEST PATHS



Lemma 3: Triangle Inequality.
For any edge (u, v)

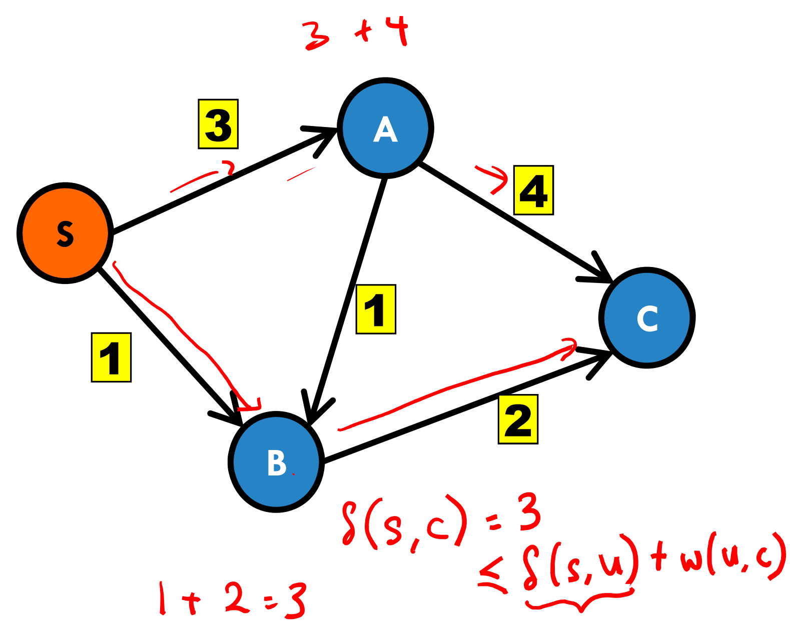
$$\delta(s,v) \le \delta(s,u) + w(u,v)$$

Proof Sketch (by contradiction):

Suppose the shortest path p has

$$\delta(s,v) > \delta(s,u) + w(u,v)$$

But then, we can take the path from $s \rightsquigarrow u \to v$ which has shorter distance so, p could not have been the shortest path. Contradiction! ■

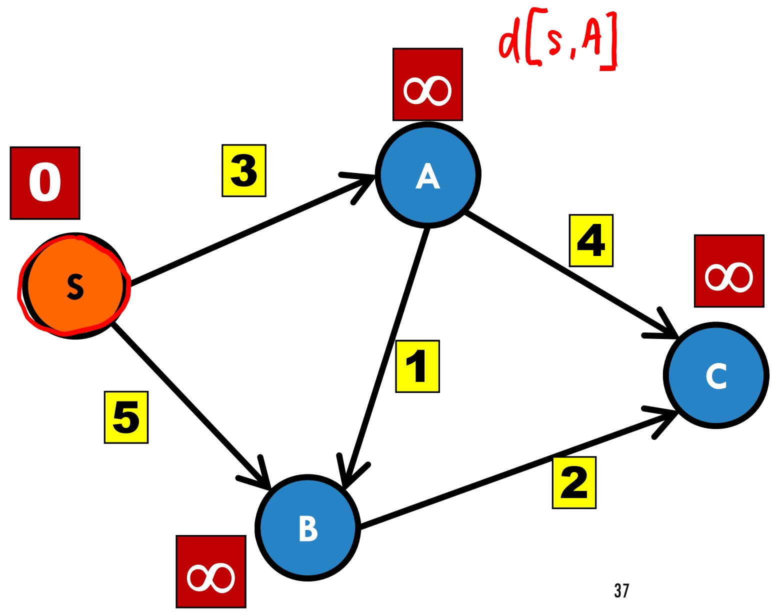


SHORTEST PATHS

```
// in Java
int[] dist = new int[V.length];
Arrays.fill(dist, INFTY);
dist[start] = 0;
```

Maintain estimate for each distance:

- **Reduce** estimate $d[s,u]$
- **Invariant:** estimate $d[s,u] \geq \delta[s,u]$



NEXT, **RELAX**!

SHORTEST PATHS

```
relax(int u, int v){
    if (dist[v] > dist[u] + weight(u,v))
        dist[v] = dist[u] + weight(u,v);
}
```

∞ $=d[S]=0$ $=3$

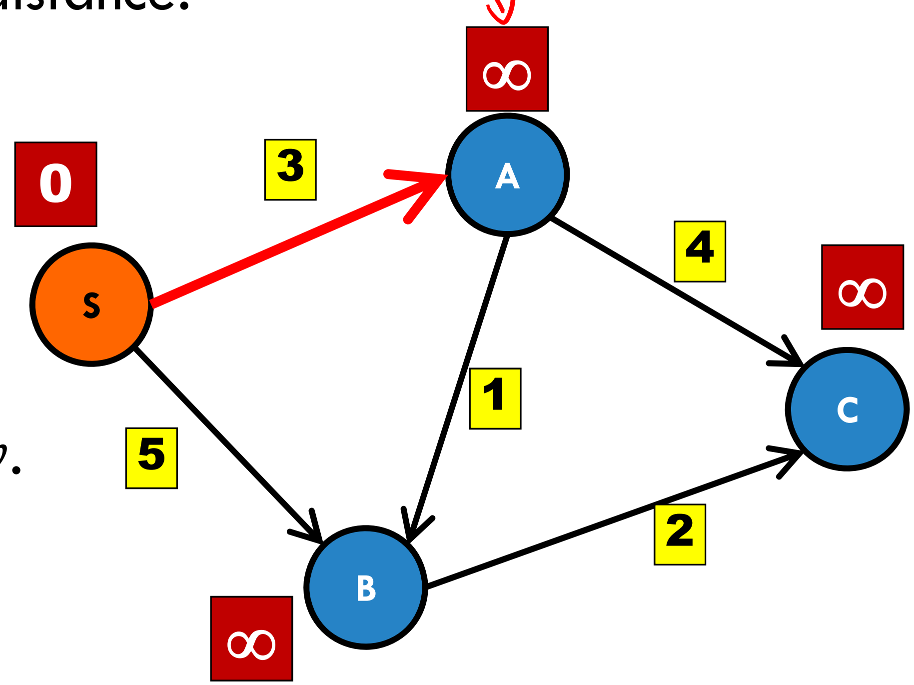
Maintain estimate for each distance:

relax(S, A)

The idea:

relax(w,v):
Test if the best way to get from $s \to v$ is to go from $s \to w$, then $w \to v$.

Update `dist`



SHORTEST PATHS

```
relax(int u, int v){
    if (dist[v] > dist[u] + weight(u,v))
        dist[v] = dist[u] + weight(u,v);
}
```

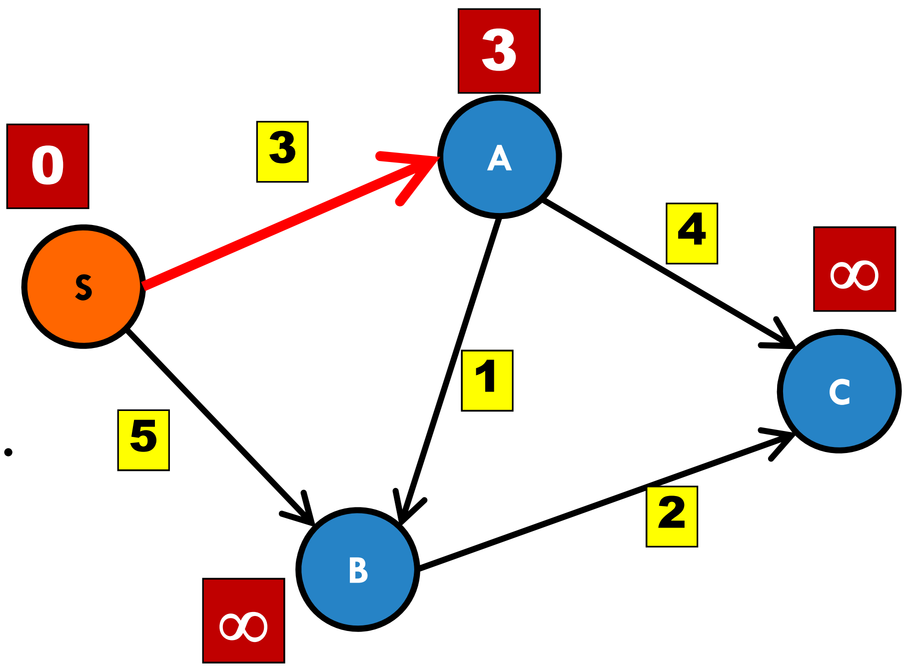
Maintain estimate for each distance:

relax(S, A)

The idea:

relax(w,v):
Test if the best way to get from $s \rightarrow v$ is to go from $s \rightarrow w$, then $w \rightarrow v$.

Update `dist`



SHORTEST PATHS

```
relax(int u, int v){
      if (dist[v] > dist[u] + weight(u,v))
         dist[v] = dist[u] + weight(u,v);
}
```

∞ 3 + 4 = 7

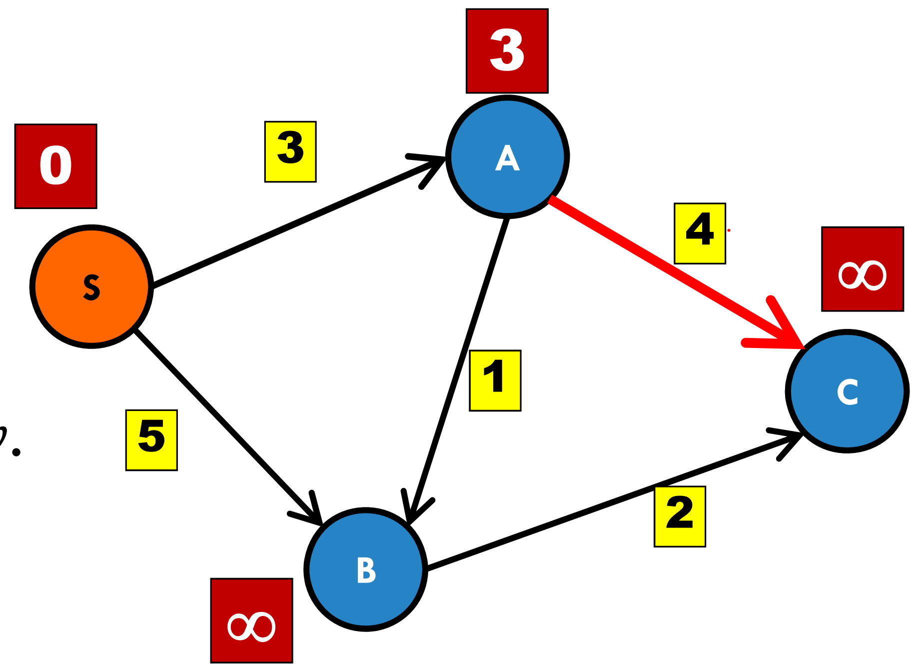
Maintain estimate for each distance:

relax(A, C)

The idea:

relax(w,v):
Test if the best way to get from $s \to v$ is to go from $s \to w$, then $w \to v$.

Update `dist`



SHORTEST PATHS

```
relax(int u, int v){
    if (dist[v] > dist[u] + weight(u,v))
        dist[v] = dist[u] + weight(u,v);
}
```

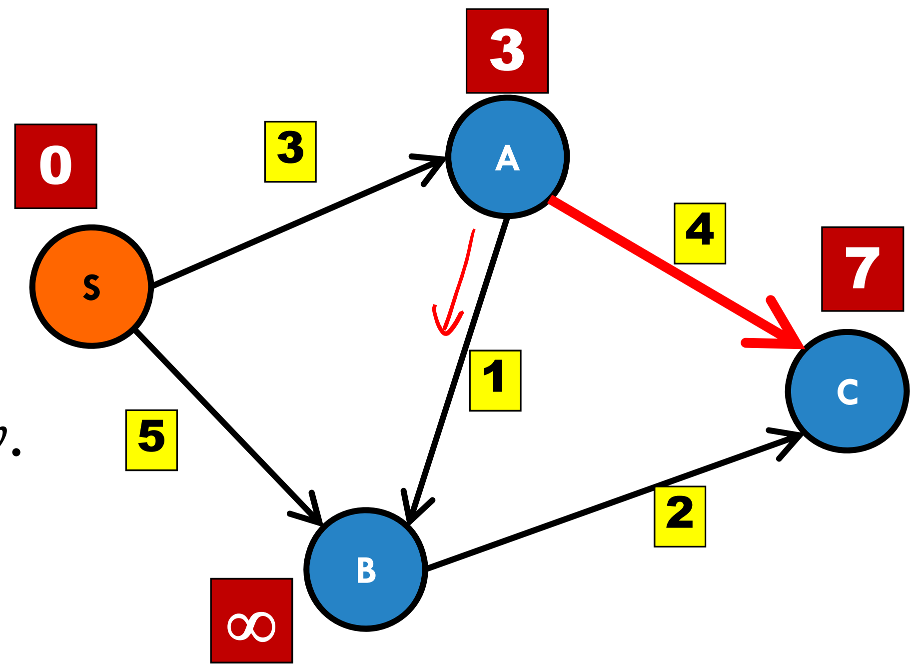
Maintain estimate for each distance:

relax(A, C)

The idea:

relax(w,v):
Test if the best way to get from $s \rightarrow v$ is to go from $s \rightarrow w$, then $w \rightarrow v$.

Update `dist`



SHORTEST PATHS

```
relax(int u, int v){
    if (dist[v] > dist[u] + weight(u,v))
        dist[v] = dist[u] + weight(u,v);
}
```

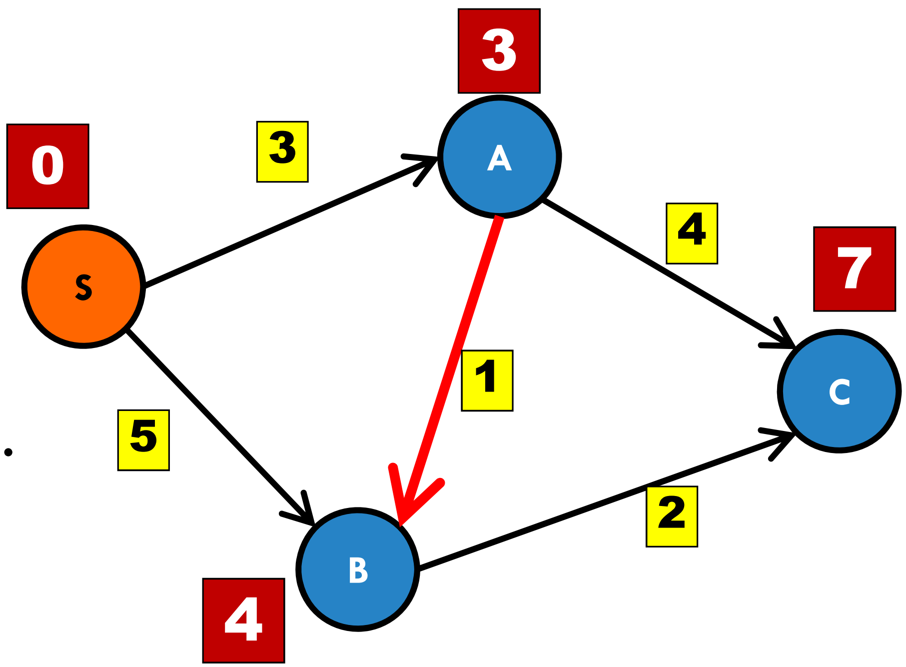
Maintain estimate for each distance:

relax(A, B)

The idea:

relax(w,v):
Test if the best way to get from $s \rightarrow v$ is to go from $s \rightarrow w$, then $w \rightarrow v$.

Update `dist`



SHORTEST PATHS

```
relax(int u, int v){
           4              0     +    5  -5
      if (dist[v] > dist[u] + weight(u,v))
         dist[v] = dist[u] + weight(u,v);
}
```

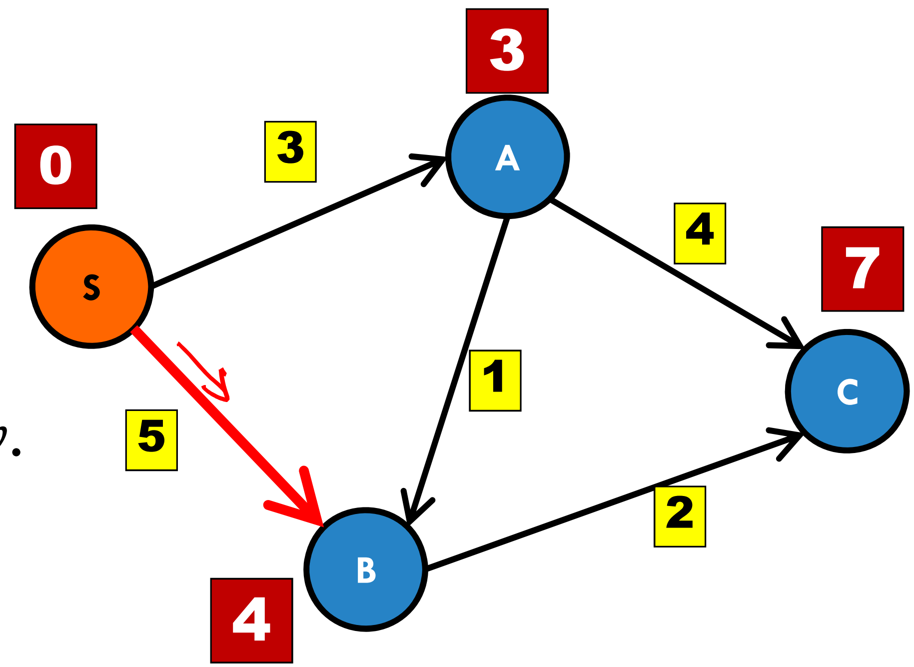
Maintain estimate for each distance:

relax(S, B)

The idea:

relax(w,v):
Test if the best way to get from $s \rightarrow v$ is to go from $s \rightarrow w$, then $w \rightarrow v$.

Update `dist`



SHORTEST PATHS

```
relax(int u, int v){
                 7           4    +    2  =6
      if (dist[v] > dist[u] + weight(u,v))
         dist[v] = dist[u] + weight(u,v);
}
```

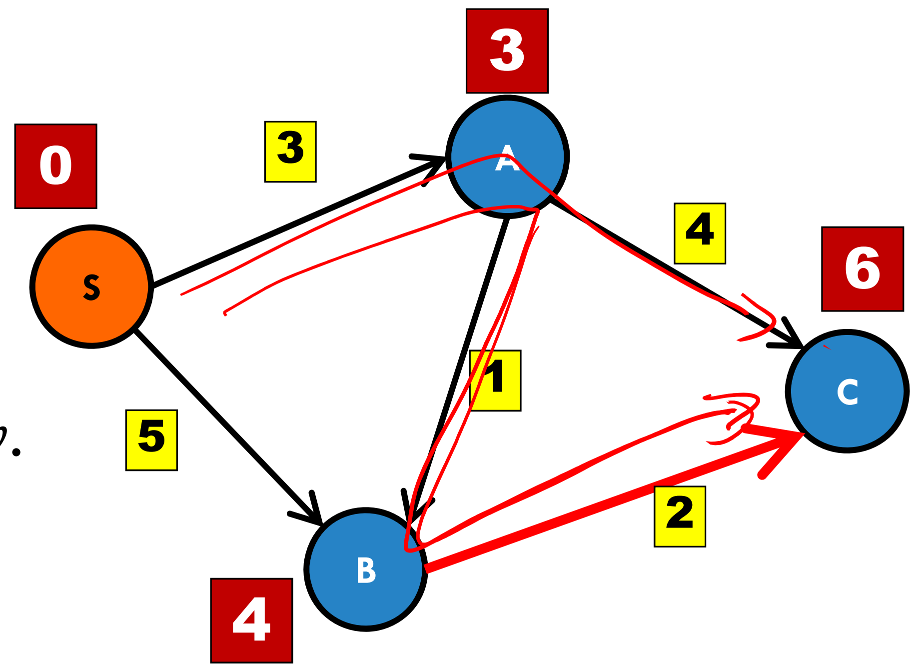
Maintain estimate for each distance:

relax(B, C)

The idea:

relax(w,v):
Test if the best way to get from $s \to v$ is to go from $s \to w$, then $w \to v$.

Update `dist`



```
relax(int u, int v)
    if (dist[v] > dist[u] + weight(u,v))
        dist[v] = dist[u] + weight(u,v);
```

RELAXATION: UPPER BOUND PROPERTY

estimate — true shortest distance

Lemma 4: We always have $d[v] \geq \delta[v]$ for all $v \in V$ and once $d[v] = \delta[v]$, it never changes.

Proof via induction (left as an exercise)

d[s, v]

Notation note: I'm going to drop the dependence of $d[s, v]$ on s to reduce clutter. So, $d[v] = d[s, v]$ for some source node s

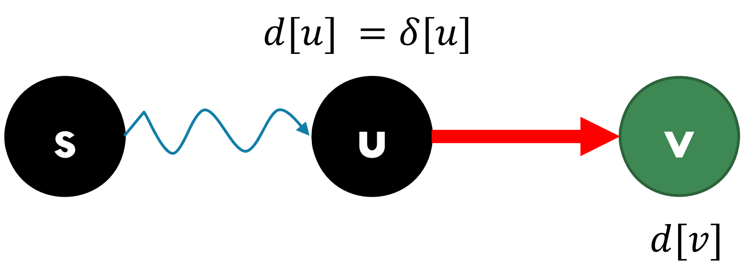
```
relax(int u, int v)
    if (dist[v] > dist[u] + weight(u,v))
        dist[v] = dist[u] + weight(u,v);
```

RELAXATION: CONVERGENCE PROPERTY

Lemma 5: If

- $s \rightsquigarrow u \to v$ is a shortest path from s to v and
- $d[u] = \delta[u]$ before relaxing edge (u, v),

then $d[v] = \delta[v]$ **at all times after relaxing**



```
relax(int u, int v)
    if (dist[v] > dist[u] + weight(u,v))
        dist[v] = dist[u] + weight(u,v);
```

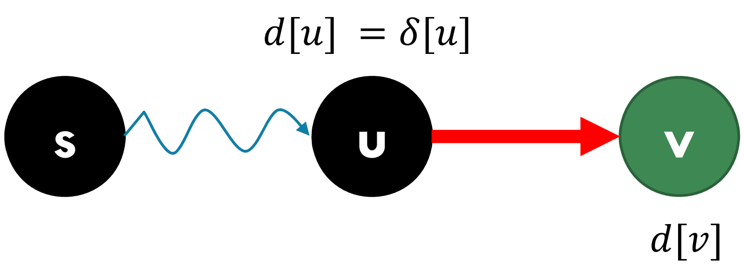
RELAXATION: CONVERGENCE PROPERTY

Lemma 5: If

- $s \rightsquigarrow u \to v$ is a shortest path from s to v and
- $d[u] = \delta[u]$ before relaxing edge (u, v),

then $d[v] = \delta[v]$ **at all times after relaxing**

$$d[v] \le d[u] + w(u, v)$$



```
relax(int u, int v)
    if (dist[v] > dist[u] + weight(u,v))
        dist[v] = dist[u] + weight(u,v);
```

RELAXATION: CONVERGENCE PROPERTY

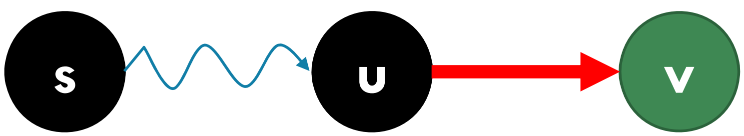
Lemma 5: If

- $s \rightsquigarrow u \to v$ is a shortest path from s to v and
- $d[u] = \delta[u]$ before relaxing edge (u, v),

then $d[v] = \delta[v]$ **at all times after relaxing**

$$
\begin{aligned}
d[v] &\le d[u] + w(u, v) \\
&= \delta[u] + w(u, v)
\end{aligned}
$$

$d[u] = \delta[u]$



$d[v] = \delta[u] + w(u, v)$

```
relax(int u, int v)
    if (dist[v] > dist[u] + weight(u,v))
        dist[v] = dist[u] + weight(u,v);
```

RELAXATION: CONVERGENCE PROPERTY

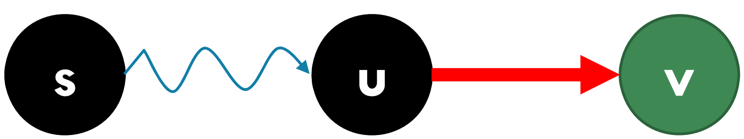
Lemma 5: If

- $s \rightsquigarrow u \rightarrow v$ is a shortest path from s to v and
- $d[u] \ = \delta[u]$ before relaxing edge (u, v),

then $d[v] \ = \delta[v]$ **at all times after relaxing**

$$d[v] \leq d[u] + w(u, v)$$
$$= \delta[u] + w(u, v)$$
$$= \delta[v]$$

$d[u] \ = \delta[u]$



$d[v] \ = \delta[v]$

```
relax(int u, int v)
    if (dist[v] > dist[u] + weight(u,v))
        dist[v] = dist[u] + weight(u,v);
```

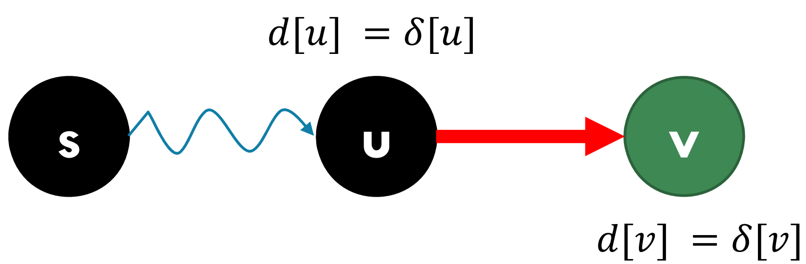
RELAXATION: CONVERGENCE PROPERTY

Lemma 5: If

- $s \rightsquigarrow u \rightarrow v$ is a shortest path from s to v and
- $d[u] = \delta[u]$ before relaxing edge (u, v),

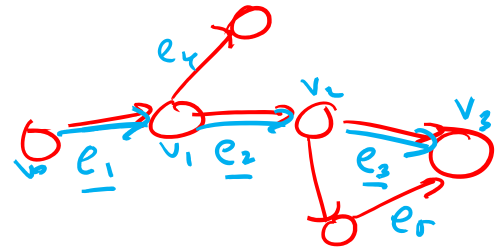
then $d[v] = \delta[v]$ **at all times after relaxing**

$$
\begin{aligned}
d[v] &\leq d[u] + w(u, v) \\
&= \delta[u] + w(u, v) \\
&= \delta[v]
\end{aligned}
$$



By Lemma 3 (upper bound property), $d[v] = \delta[v]$ is maintained and never changes ■

PATH RELAXATION PROPERTY



$d[v_3] = \delta[v_3]$

$e \in E$

e_1, e_5, e_4, e_2, e_3

Lemma 6. If $p = (v_0, v_1, \ldots, v_k)$ is a shortest path from $s = v_0$ to v_k and we relax the edges of p in the order

$e_1\, e_1\, e_1\, e_2\, e_2\, e_4\, e_3$

$$(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$$

Then $d[v_k] = \delta[v_k]$. correct

This property holds ***regardless of any other relaxation steps that occur*** (even **intermixed**)

- E.g., $(v_0, v_1), (v_i, v_j), (v_1, v_2), \ldots, (v_{k-1}, v_k)$ will *still* result in $d[v_k] = \delta[v_k]$.

PROOF (SKETCH) BY INDUCTION

Lemma 5. If $p = (v_0, v_1, \dots, v_k)$ is a shortest path from $s = v_0$ to v_k and we relax the edges of p in the order
$(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$
Then $d[v_k] = \delta[v_k]$.

Consider the shortest path p from source vertex s to v_k

Show: After the k-th edge is relaxed, $d[v_k] = \delta[v_k]$

Proof Strategy: (like recursion)

- *Base case:* Show the statement is true for $k = 0$
- *Inductive hypothesis:* Assume the statement is true for some $k - 1$
- *Inductive step:* Show the statement holds for k

PROOF (SKETCH) BY INDUCTION

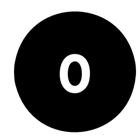
Lemma 5. If $p = (v_0, v_1, \dots, v_k)$ is a shortest path from $s = v_0$ to v_k and we relax the edges of p in the order $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$
Then $d[v_k] = \delta[v_k]$.

Consider the **shortest path $\boldsymbol{p}$** from source vertex v_0 to v_k

Show: After the k-th edge is relaxed, $d[v_k] = \delta[v_k]$

Base Case:

$D[v_0] = \delta[v_0] = 0$



$d[v_0] = \delta[v_0] = 0$

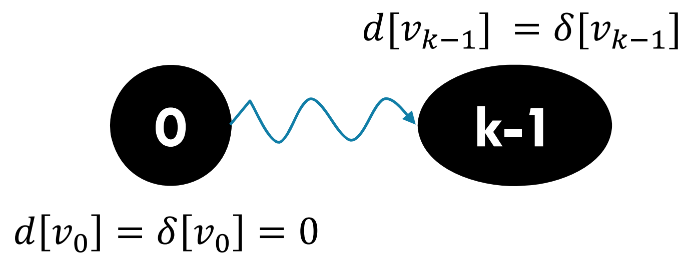
PROOF (SKETCH) BY INDUCTION

Lemma 5. If $p = (v_0, v_1, \dots, v_k)$ is a shortest path from $s = v_0$ to v_k and we relax the edges of p in the order $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$
Then $d[v_k] = \delta[v_k]$.

Inductive hypothesis:

Assume:

$$\mathrm{d}[v_{k-1}] = \delta[v_{k-1}]$$



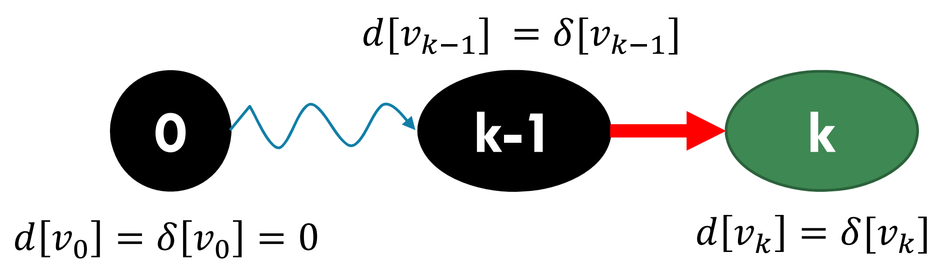
PROOF (SKETCH) BY INDUCTION

Lemma 5. If $p = (v_0, v_1, \dots, v_k)$ is a shortest path from $s = v_0$ to v_k and we relax the edges of p in the order
$(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$
Then $d[v_k] = \delta[v_k]$.

Inductive step: (Show for step k)

v_k, is reachable from v_{k-1} where

$d[v_{k-1}] = \delta[v_{k-1}]$



Lemma 5. If $p = (v_0, v_1, \dots, v_k)$ is a shortest path from $s = v_0$ to v_k and we relax the edges of p in the order $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$
Then $d[v_k] = \delta[v_k]$.

PROOF (SKETCH) BY INDUCTION

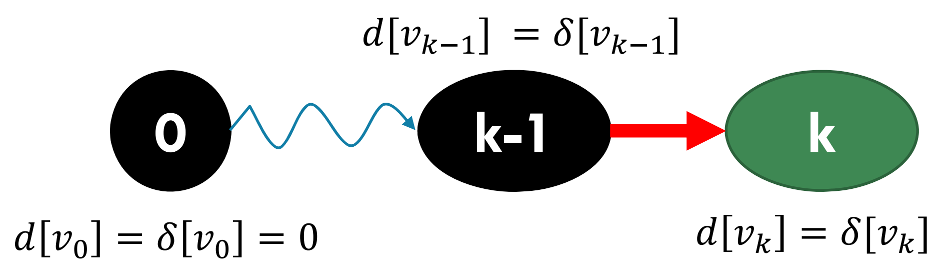
Inductive step: (Show for step k)

v_k, is reachable from v_{k-1} where

$d[v_{k-1}] = \delta[v_{k-1}]$

When we relax $e = (v_{k-1}, v_k)$

$$d[v_k] = \delta[v_{k-1}] + w(e)$$
$$= \delta[v_k]$$



PROOF (SKETCH) BY INDUCTION

Lemma 5. If $p = (v_0, v_1, \dots, v_k)$ is a shortest path from $s = v_0$ to v_k and we relax the edges of p in the order $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$
Then $d[v_k] = \delta[v_k]$.

Inductive step: (Show for step k)

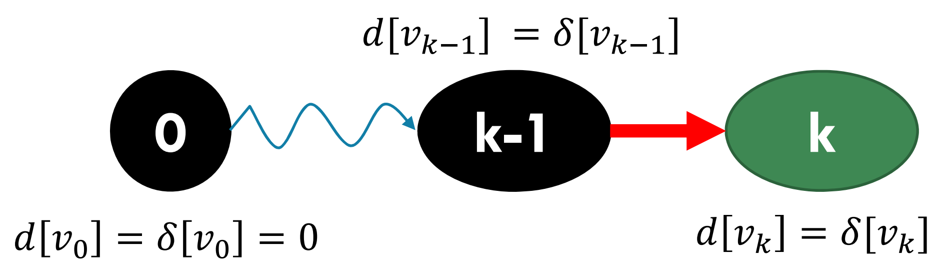
v_k, is reachable from v_{k-1} where

$d[v_{k-1}] = \delta[v_{k-1}]$

When we relax $e = (v_{k-1}, v_k)$

$$d[v_k] = \delta[v_{k-1}] + w(e)$$
$$= \delta[v_k]$$

And by convergence property, after relaxation, the equality is maintained. ■



LET'S SUMMARIZE:

Assuming no negative cycles:

The shortest path must be a simple path

When performing relaxations:

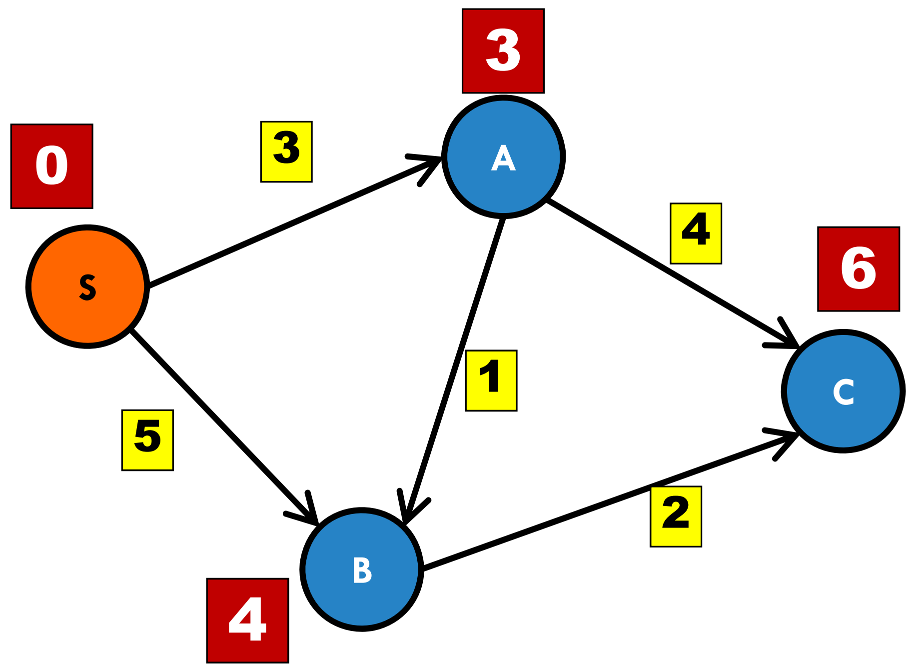
- **Upper-Bound Property:** Once a shortest path estimate $d[v_i]$ is correct, $d[v_i] = \delta[v_i]$, it never changes.
- **Convergence Property:** For a shortest path $v_0 \rightsquigarrow v_{k-1} \rightarrow v_k$, if the estimate $d[v_{k-1}]$ is correct, then after relaxing (v_{k-1}, v_k), the estimate $d[v_k]$ will also be correct (forever).
- **Path Relaxation Property:** If p is a shortest path from v_0 to v_k, then once we relax the edges of p in order, then $d[v_k] = \delta[v_k]$

SHORTEST PATHS

```
relax(int u, int v){
    if (dist[v] > dist[u] + weight(u,v))
        dist[v] = dist[u] + weight(u,v);
}
```

Maintain estimate for each distance:

```
for Edge e in graph
    relax(e)
```



SHORTEST PATHS

Maintain estimate for each distance:

```
for Edge e in graph
     relax(e)
```

Does this algorithm always work?

A. Yes!
B. No!
C. Maybe yes, maybe no…
D. Hmm.. I would ask Naruto but he hasn't appeared for a while…

SHORTEST PATHS

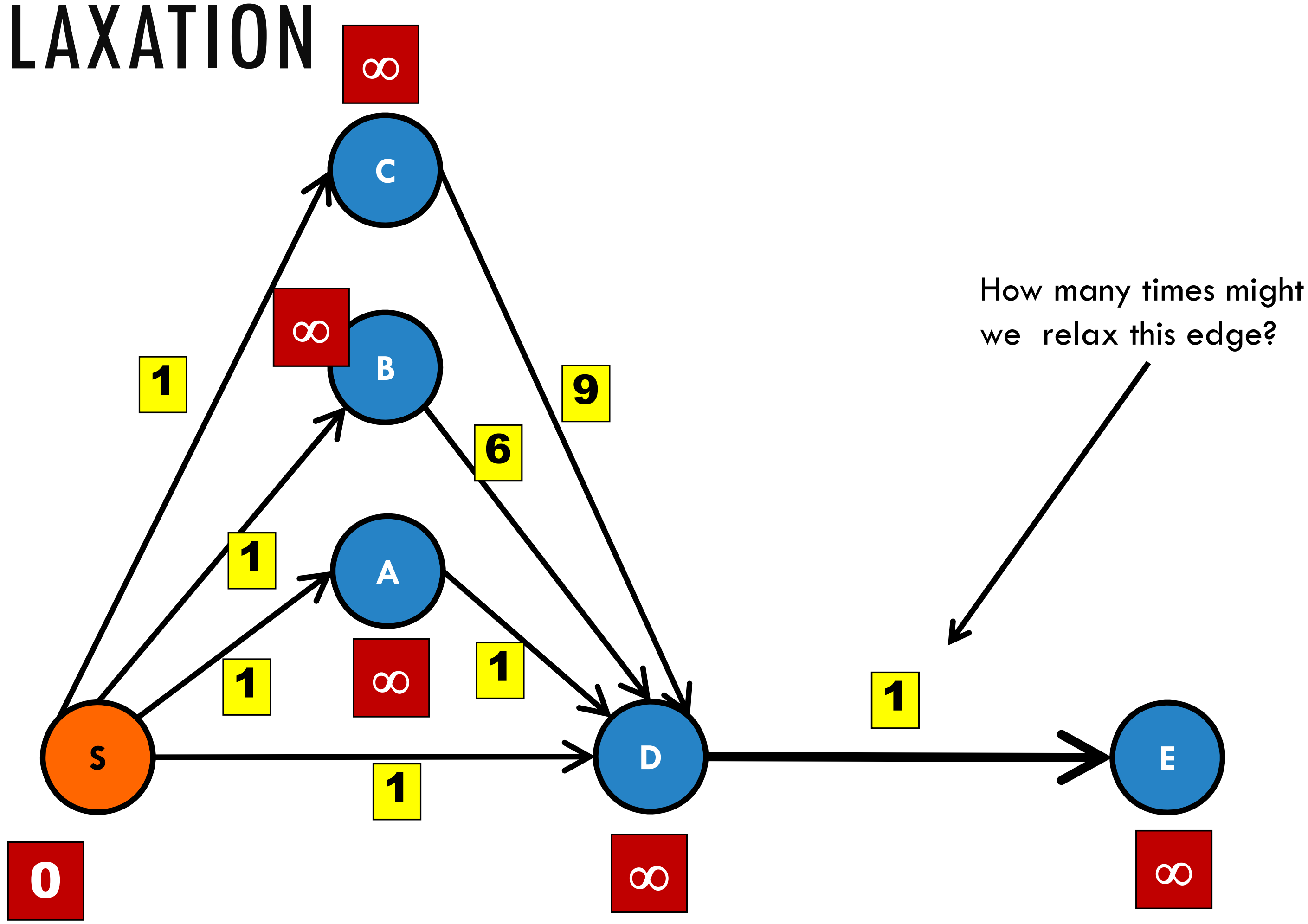
Maintain estimate for each distance:

```
for Edge e in graph
     relax(e)
```

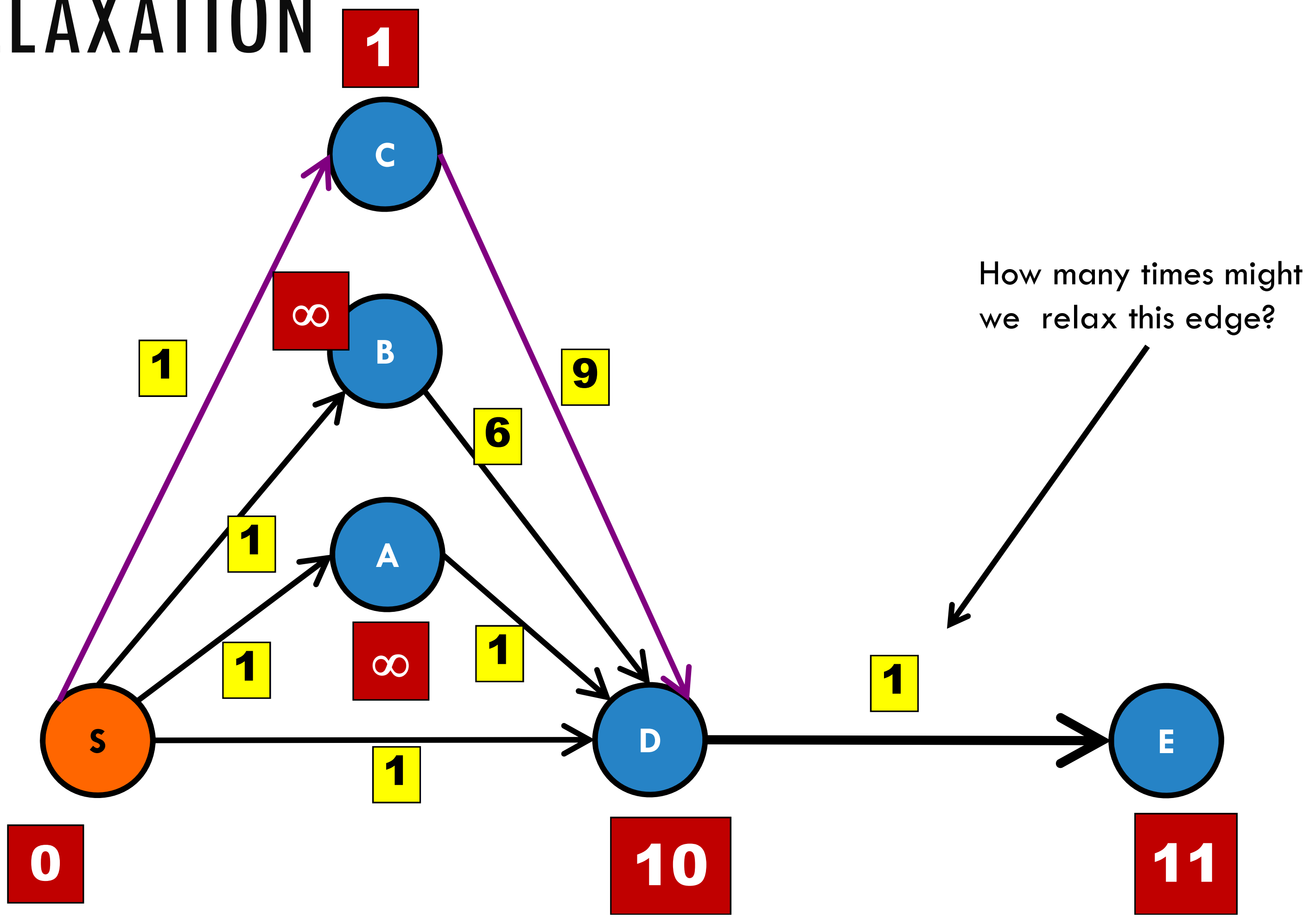
Does this algorithm always work?

A. Yes!
B. No!
C. Maybe yes, maybe no…
D. Hmm.. I would ask Naruto but he hasn't appeared for a while…

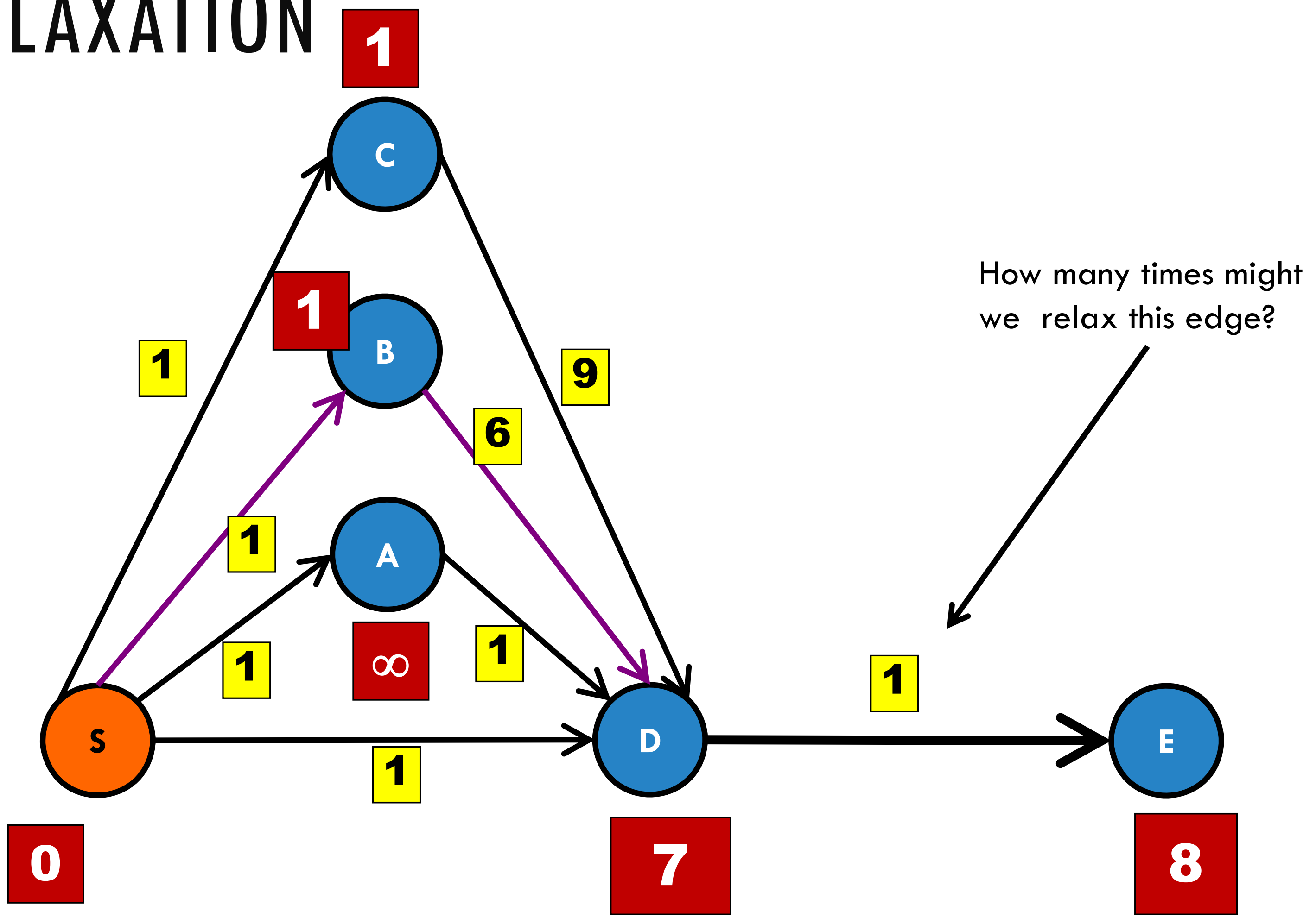
RELAXATION



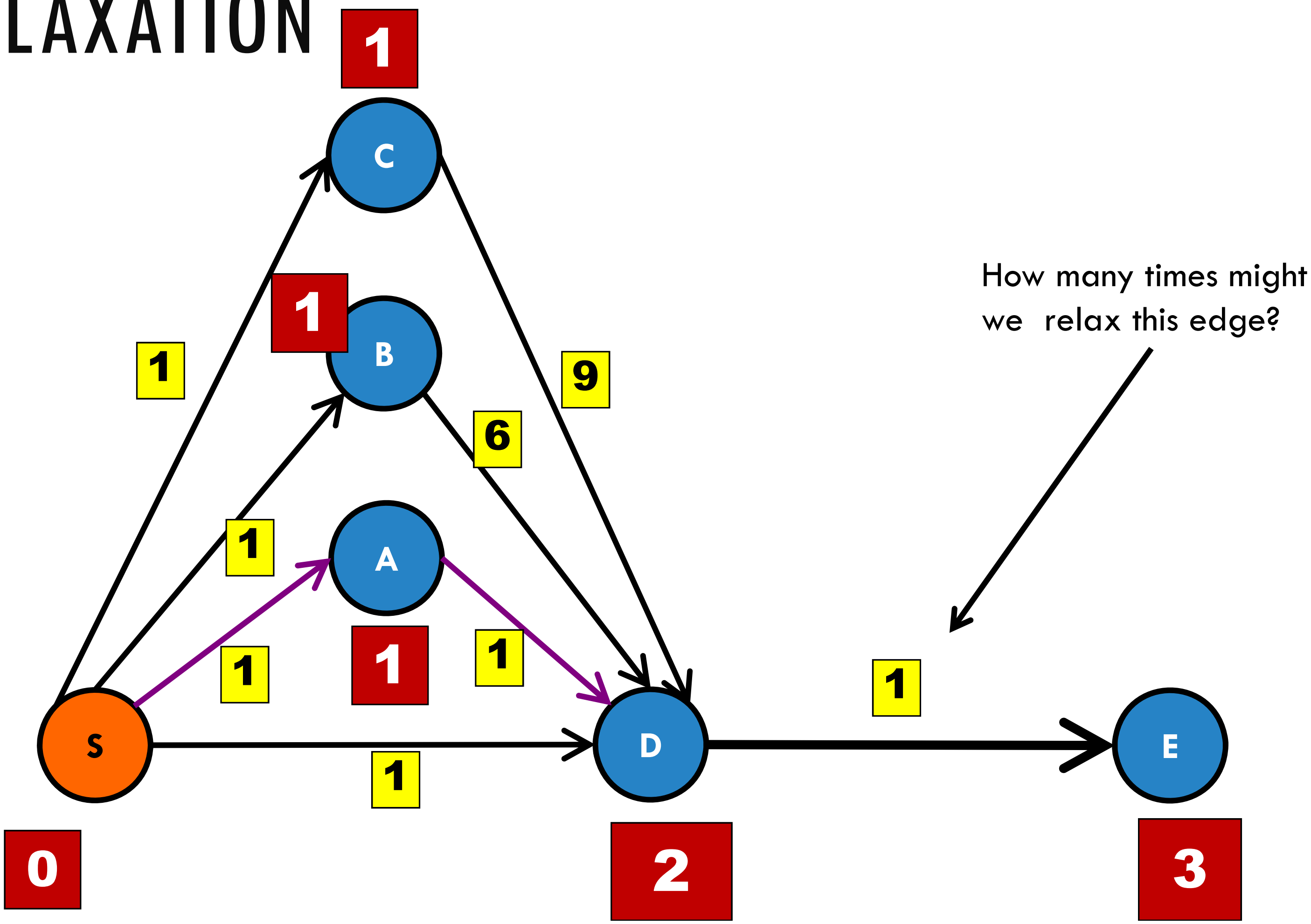
RELAXATION


1
C
∞
B
1
9
6
1
A
1
1
∞
1
S
D
1
E
1
0
10
11
How many times might we relax this edge?

RELAXATION

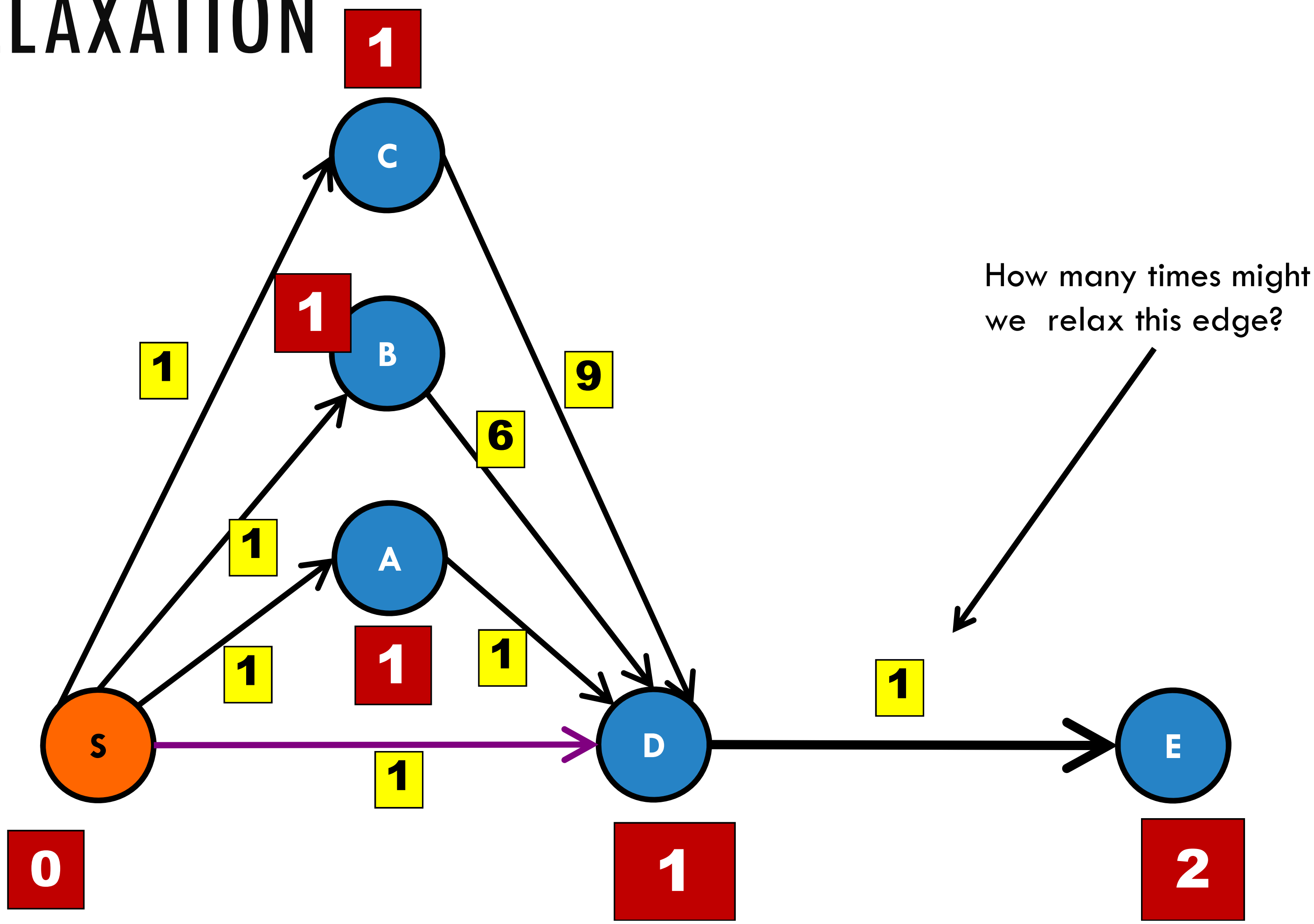

1
C
1
B
1
9
6
1
A
1
∞
1
1
S
1
D
E
0
7
8
How many times might
we relax this edge?

RELAXATION



1
C
1
B
9
6
1
A
1
1
1
1
S
D
E
1
0
2
3
How many times might
we relax this edge?

RELAXATION



1
C
1
B
1
9
6
1
A
1
1
1
S
D
E
1
1
0
1
2
How many times might we relax this edge?

HOW MANY TIMES MUST I RELAX?

Assuming no negative cycles:

The shortest path must be a simple path

When performing relaxations:

- **Upper-Bound Property:** Once a shortest path estimate $d[v_i]$ is correct, $d[v_i] = \delta[v_i]$, it never changes.
- **Convergence Property:** For a shortest path $v_0 \rightsquigarrow v_{k-1} \rightarrow v_k$, if the estimate $d[v_{k-1}]$ is correct, then after relaxing (v_{k-1}, v_k), the estimate $d[v_k]$ will also be correct (forever).
- **Path Relaxation Property:** If p is a shortest path from v_0 to v_k, then once we relax the edges of p in order, then $d[v_k] = \delta[v_k]$

SIMPLE PATHS

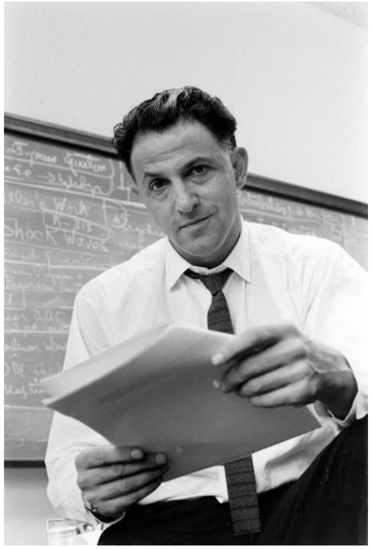
Lemma 2: If $G = (V, E)$ contains **no negative weight cycles,** then the shortest path p from source vertex s to a vertex v is a **simple path.**

A **simple path** is defined as path $p = \{v_0, v_1, v_2, \ldots, v_k\}$ where $(v_i, v_{i+1}) \in E, \forall\, 0 \le i \le (k-1)$ and there is **<u>no</u>** repeated vertex along this path.

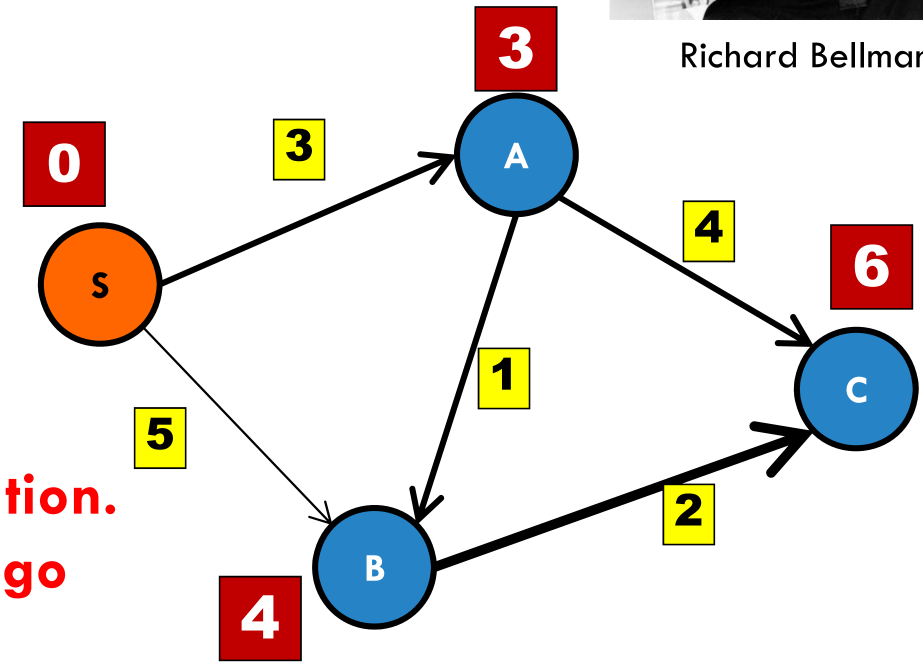
This means that the shortest path can have *at most* $\underline{\;\mathbf{|V|-1}\;}$ edges

BELLMAN-FORD ALGORITHM

```
n = V.length
for i = 1 to n-1
   for Edge e in Graph
       relax(e)
```



Richard Bellman



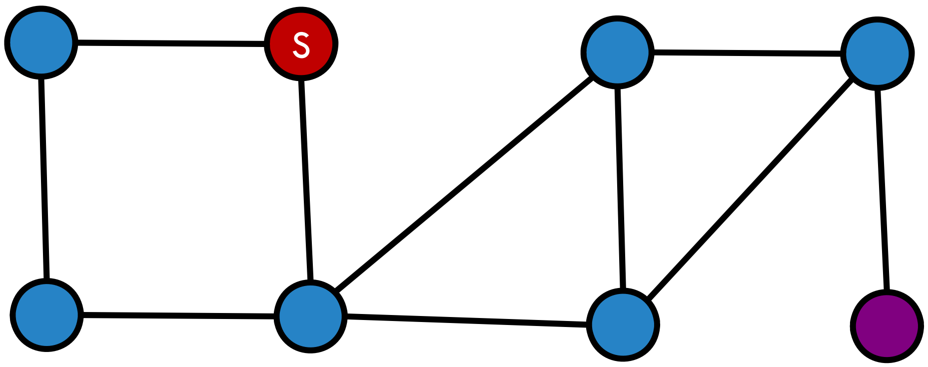
Does Bellman-Ford always work?

Yes! Because of Path Relaxation. Proof by Induction in Visualgo

WHY DOES BELLMAN FORD WORK?

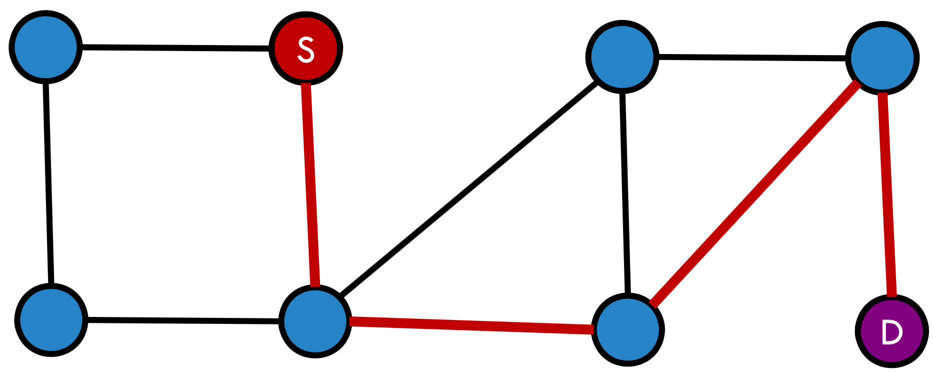
Theorem 1: If $G = (V, E)$ contains **no negative weight cycle,** then after Bellman Ford's algorithm terminates, we will have $\boldsymbol{D[u] = \delta(s, u)}, \forall\, \boldsymbol{u} \in \boldsymbol{V}$.

WHY DOES BELLMAN-FORD WORK?



WHY DOES BELLMAN-FORD WORK?

```
BellmanFord(V,E)
    n = V.length
    for i = 1 to n-1
      for Edge e in E
        relax(e)
```

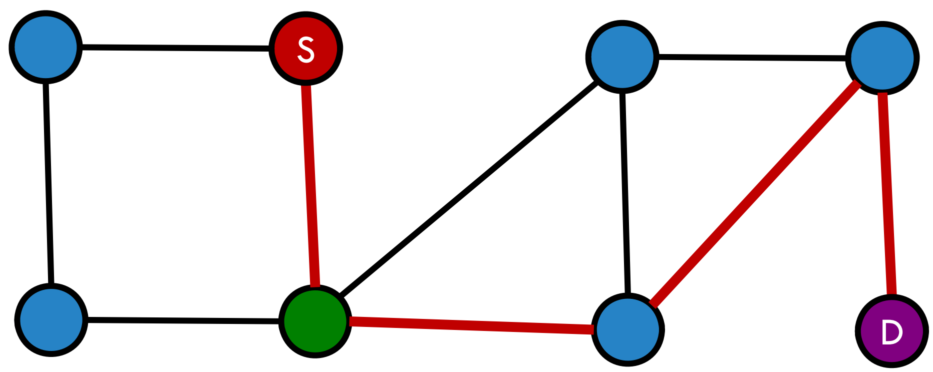


Look at minimum weight path from S to D.

(Path is simple: no loops.)

WHY DOES BELLMAN-FORD WORK?

```
BellmanFord(V,E)
    n = V.length
    for i = 1 to n-1
      for Edge e in E
         relax(e)
```

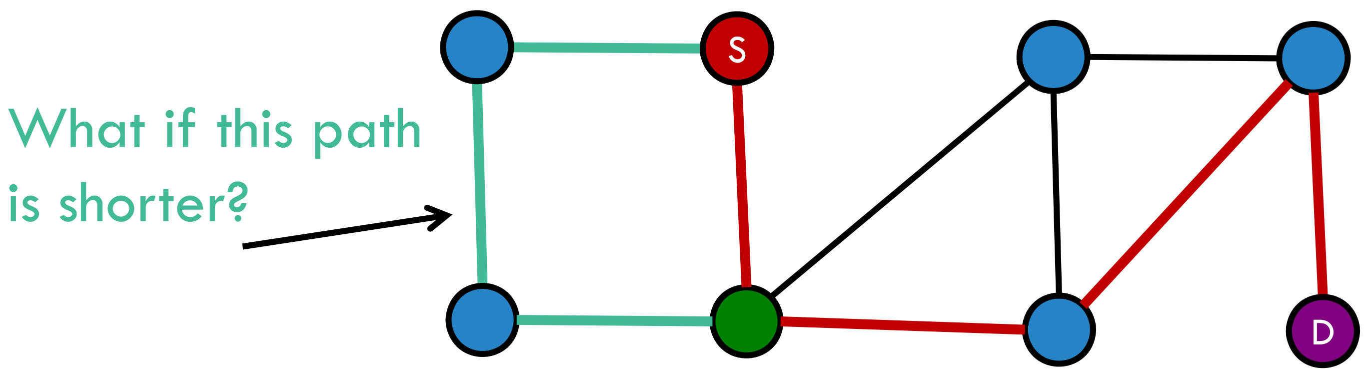


After 1 iteration, 1 hop estimate is correct. **(Path Relaxation)**

meaning: All shortest paths that are 1 hop long are now correct

WHY DOES BELLMAN-FORD WORK?

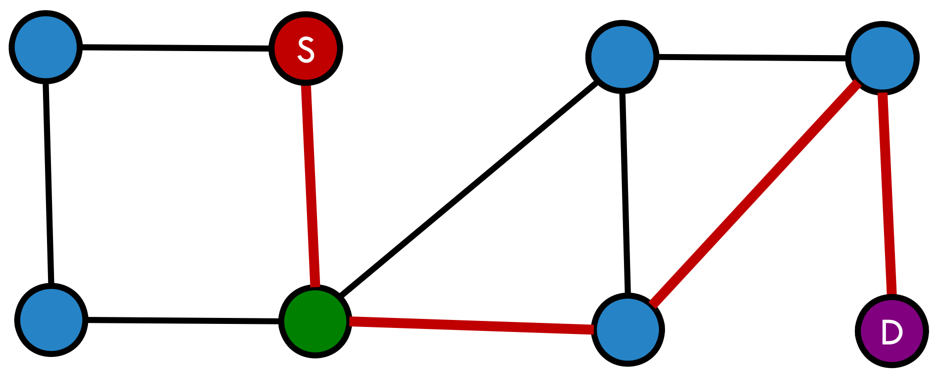
```
BellmanFord(V,E)
    n = V.length
    for i = 1 to n-1
      for Edge e in E
        relax(e)
```



After 1 iteration, 1 hop estimate is correct. **(Path Relaxation)**

WHY DOES BELLMAN-FORD WORK?

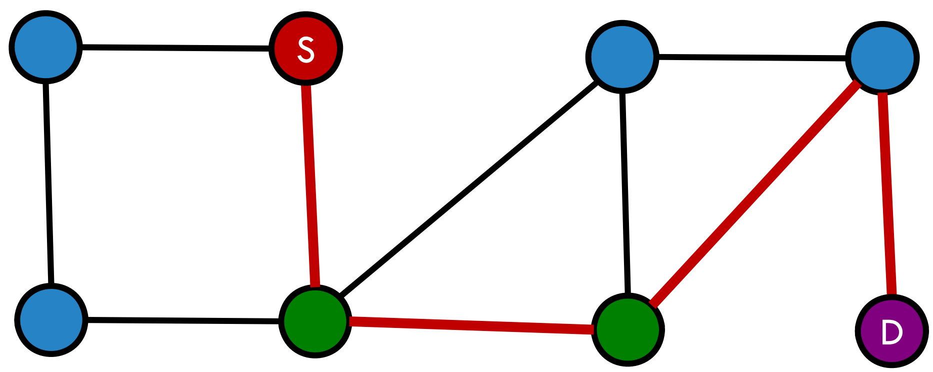
```
BellmanFord(V,E)
    n = V.length
    for i = 1 to n-1
      for Edge e in E
         relax(e)
```



After 1 iteration, 1 hop estimate is correct. **(Path Relaxation)**

WHY DOES BELLMAN-FORD WORK?

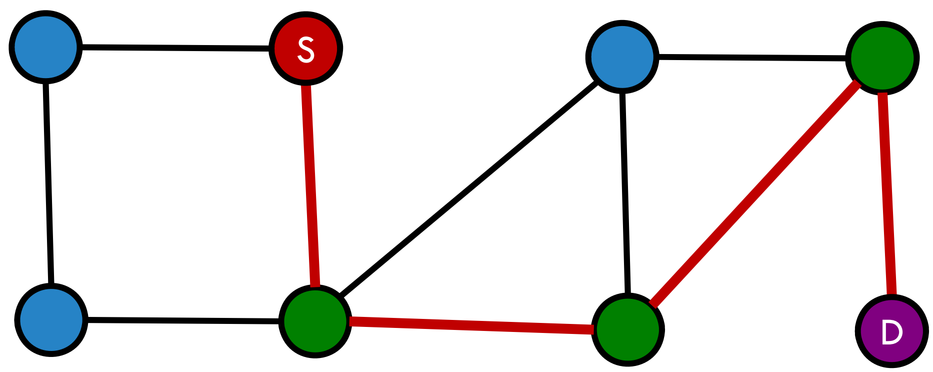
```
BellmanFord(V,E)
    n = V.length
    for i = 1 to n-1
      for Edge e in E
         relax(e)
```



After 2 iterations, 2 hop estimate is correct. **(Path Relaxation)**

WHY DOES BELLMAN-FORD WORK?

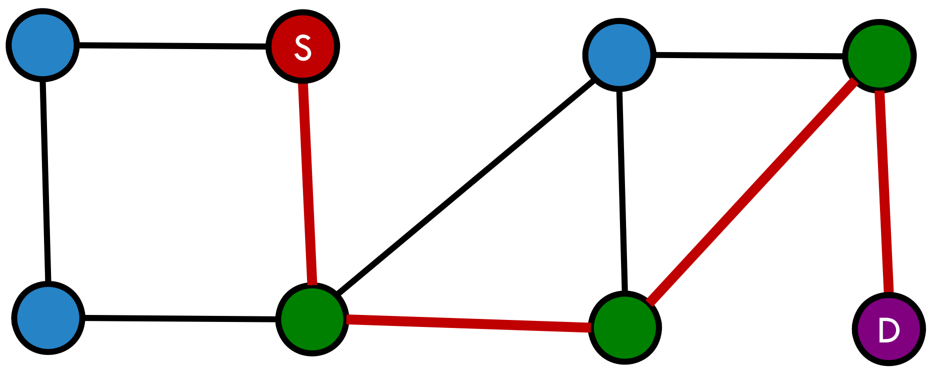
```
BellmanFord(V,E)
    n = V.length
    for i = 1 to n-1
      for Edge e in E
        relax(e)
```



After 3 iterations, 3 hop estimate is correct. **(Path Relaxation)**

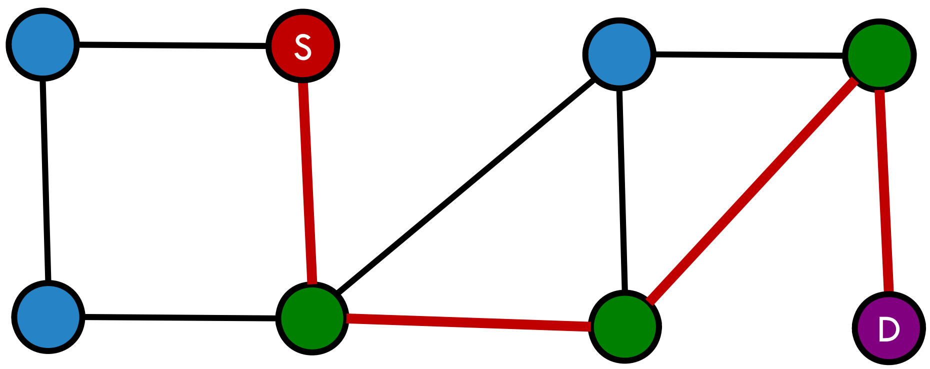
WHY DOES BELLMAN-FORD WORK?

```
BellmanFord(V,E)
    n = V.length
    for i = 1 to n-1
      for Edge e in E
         relax(e)
```



After 4 iterations, D estimate is correct. **(Path Relaxation)**

WHY DOES BELLMAN-FORD WORK?



Keep running till V-1 and Bellman-Ford finds shortest paths from s to all other nodes!

```
BellmanFord(V,E)
    n = V.length
    for i = 1 to n-1
      for Edge e in E
        relax(e)
```

BELLMAN-FORD WORKS.

Theorem 1: If $G = (V, E)$ contains **no negative weight cycles,** then after Bellman Ford's algorithm terminates, we will have $\boldsymbol{D[u] = \delta(s, u), \forall\, u \in V}$.

Proof Sketch (Direct):

Given source s and *any* destination t

Let $p = (v_0, v_1, \dots, v_k)$ be the shortest path from s to t

$s \qquad\qquad t$

```
BellmanFord(V,E)
    n = V.length
    for i = 1 to n-1
      for Edge e in E
        relax(e)
```

> **Theorem 1:** If $G = (V, E)$ contains **no negative weight cycles**, then after Bellman Ford's algorithm terminates, we will have $\boldsymbol{D}[\boldsymbol{v}] = \boldsymbol{\delta}(\boldsymbol{s}, \boldsymbol{u}), \forall\, \boldsymbol{u} \in \boldsymbol{V}$.

Proof Sketch (Direct):

How long can p be? $k \leq |V| - 1$

What happens at each iteration?

In each iteration, we relax all $|E|$ edges.
Within the $i = 1,2, \ldots, k$ iteration, we relax (v_{i-1}, v_i)
By the path relaxation property, after $|V| - 1$ iterations,
$d[t = v_k] = \delta[t = v_k]$ ■

WHAT IS THE RUNNING TIME OF BELLMAN-FORD?

```
n = V.length
for i = 1 to n-1
  for Edge e in Graph
     relax(e)
```

$|V|$

$|E|$

$O(|V||E|)$

What is the running time of Bellman-Ford?

A. $O(V)$
B. $O(E)$
C. $O(V + E)$
D. $O(VE)$
E. $O(E \log V)$
F. I have no idea.

WHAT IS THE RUNNING TIME OF BELLMAN-FORD?

```
n = V.length
for i = 1 to n-1
  for Edge e in Graph
      relax(e)
```

What is the running time of Bellman-Ford?

A. $O(V)$
B. $O(E)$
C. $O(V+E)$
D. $\boldsymbol{O(VE)}$
E. $O(E \log V)$
F. I have no idea.

EARLY TERMINATION?

```
n = V.length
for i = 1 to n-1
  for Edge e in Graph
     relax(e)
```

When can we terminate early?

A. When a relax operation has no effect.
B. When two consecutive relax operations have no effect.
C. When an entire sequence of |E| relax operations have no effect.
D. Never. Only after |V| complete iterations.

EARLY TERMINATION?

```
n = V.length
for i = 1 to n-1
  for Edge e in Graph
     relax(e)
```

When can we terminate early?

A. When a relax operation has no effect.
B. When two consecutive relax operations have no effect.
C. When an entire sequence of |E| relax operations have no effect.
D. Never. Only after |V| complete iterations.

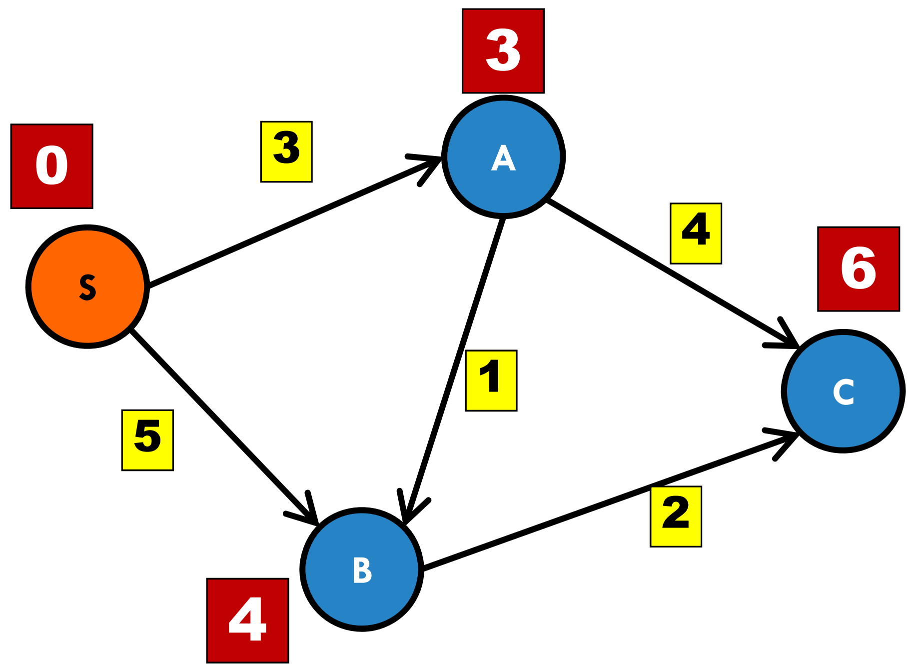
SHORTEST PATHS

```
relax(int u, int v){
      if (dist[v] > dist[u] + weight(u,v))
         dist[v] = dist[u] + weight(u,v);
}
```

Maintain estimate for each distance:

```
for Edge e in graph
     relax(e)
```

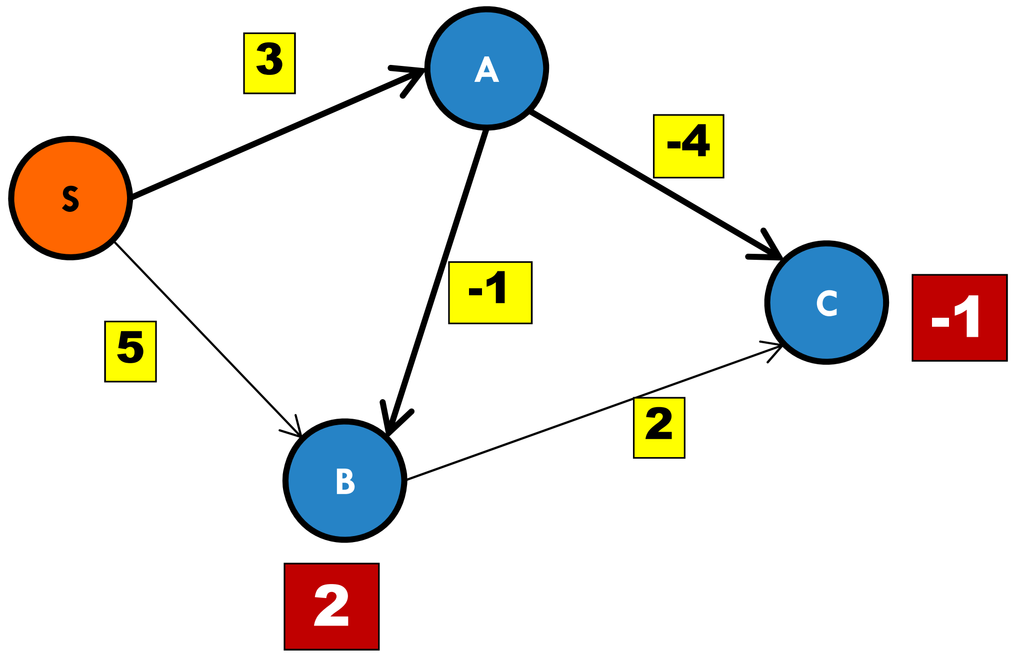
If we relax all the edges and there is no faster way to get to any node, we have the shortest paths!



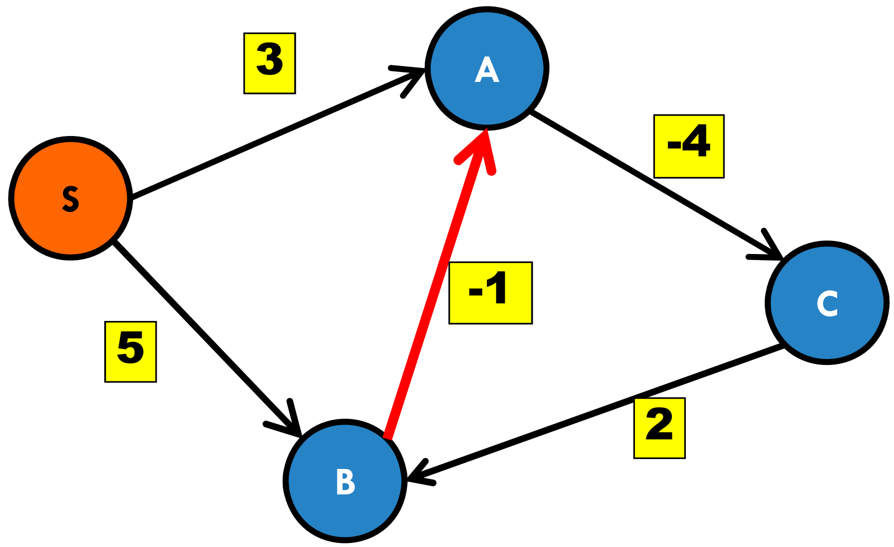
NEGATIVE EDGE WEIGHTS?

Bellman-Ford has no problems with negative edge weights!

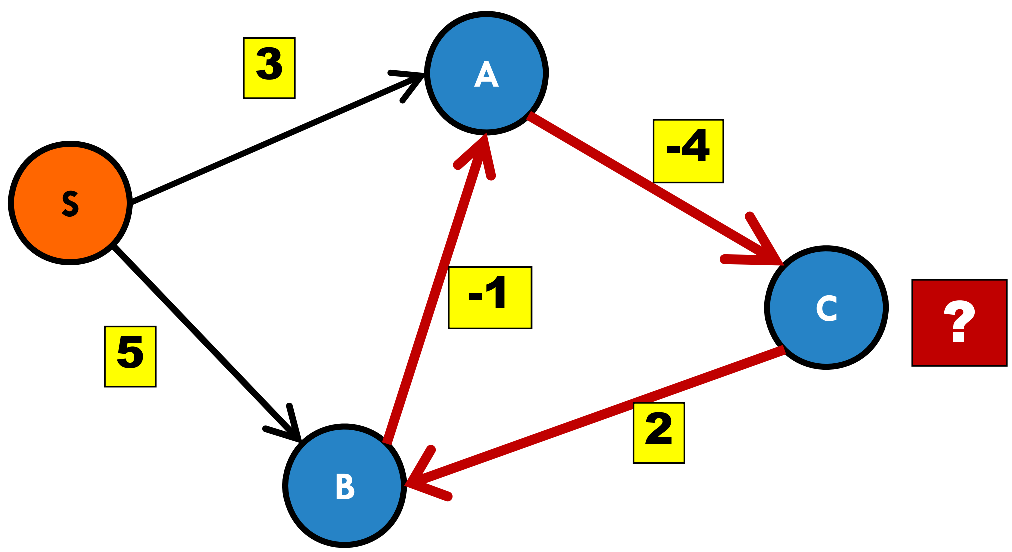
Almost…



WHAT IF THE GRAPH LOOKS LIKE THIS:



NEGATIVE WEIGHT **CYCLE**

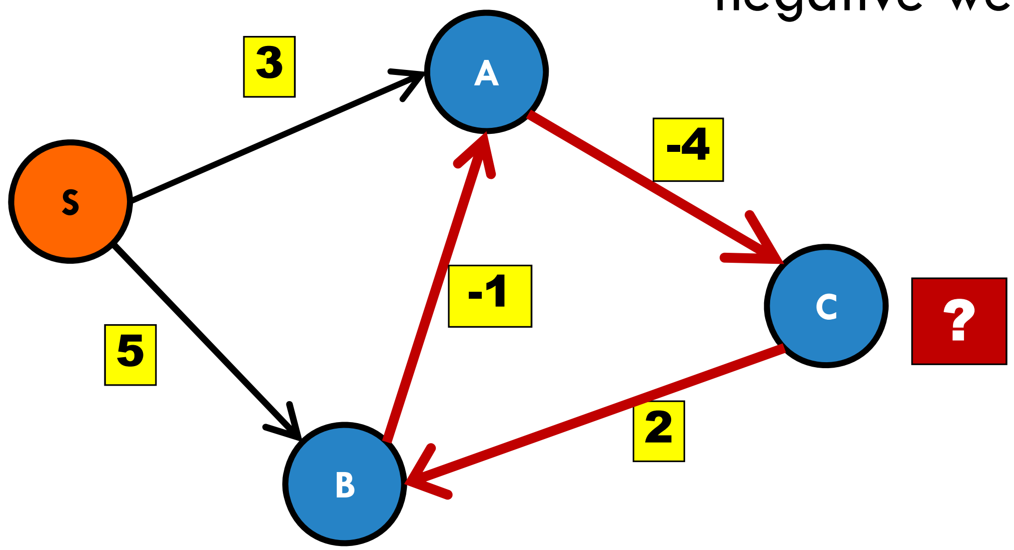


d(S,C) is infinitely negative!

NEGATIVE WEIGHT **CYCLE**

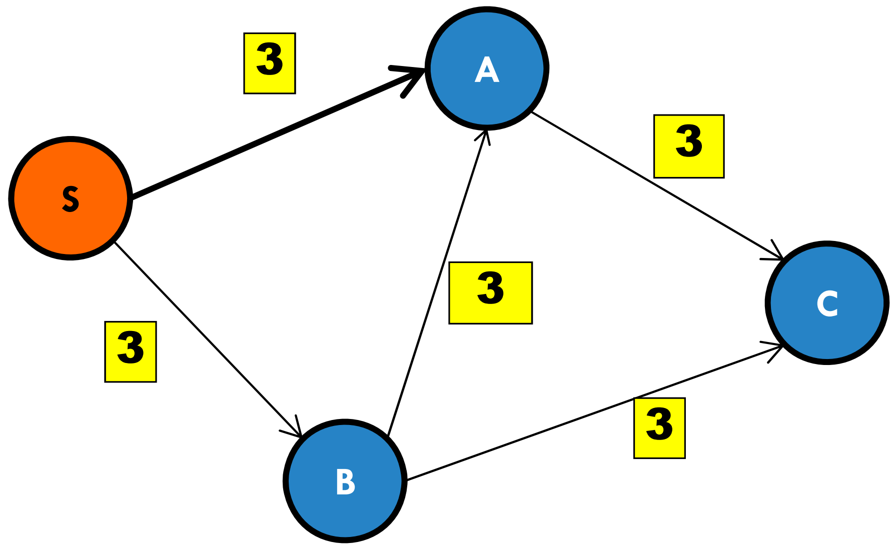
Run Bellman-Ford for |V| iterations.

If an **estimate changes in the last iteration** then negative weight cycle.



How to **detect** negative weight cycles?

SPECIAL CASE:



all edges have the **same weight**: **What can we use?** **BFS!!!**

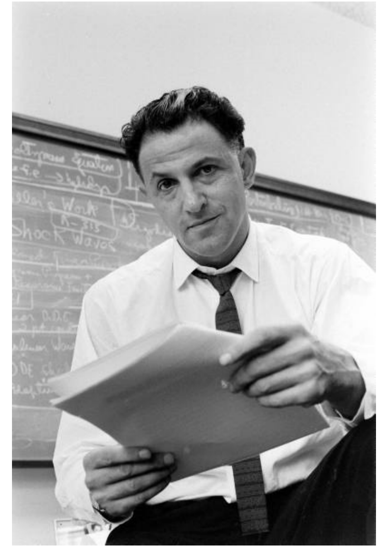
SPECIAL CASES

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	
On Tree	BFS / DFS	
On DAG	Topological Sort	

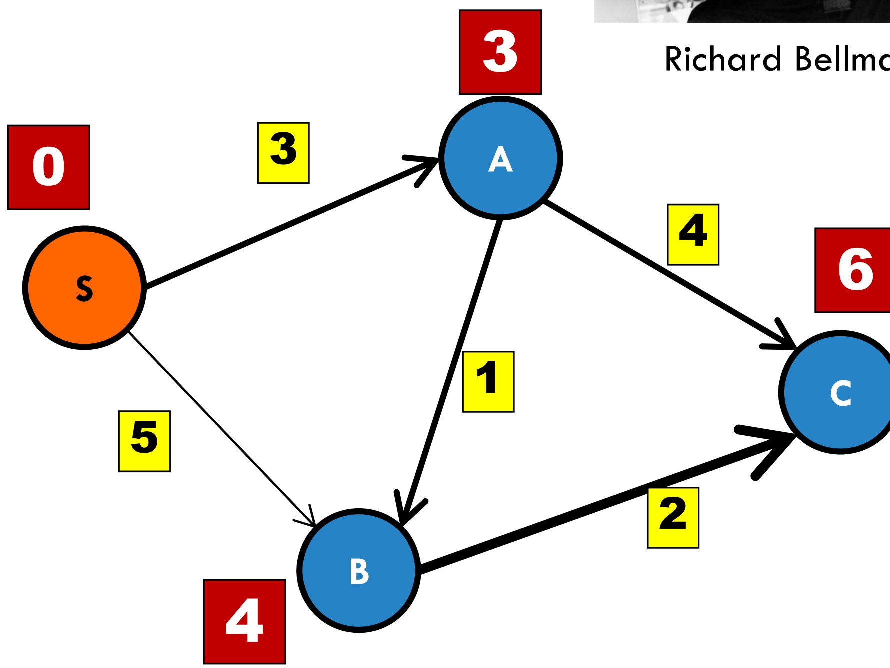
BELLMAN-FORD ALGORITHM

```
n = V.length
for i = 1 to n-1
  for Edge e in Graph
      relax(e)
```

In what order should we relax edges?



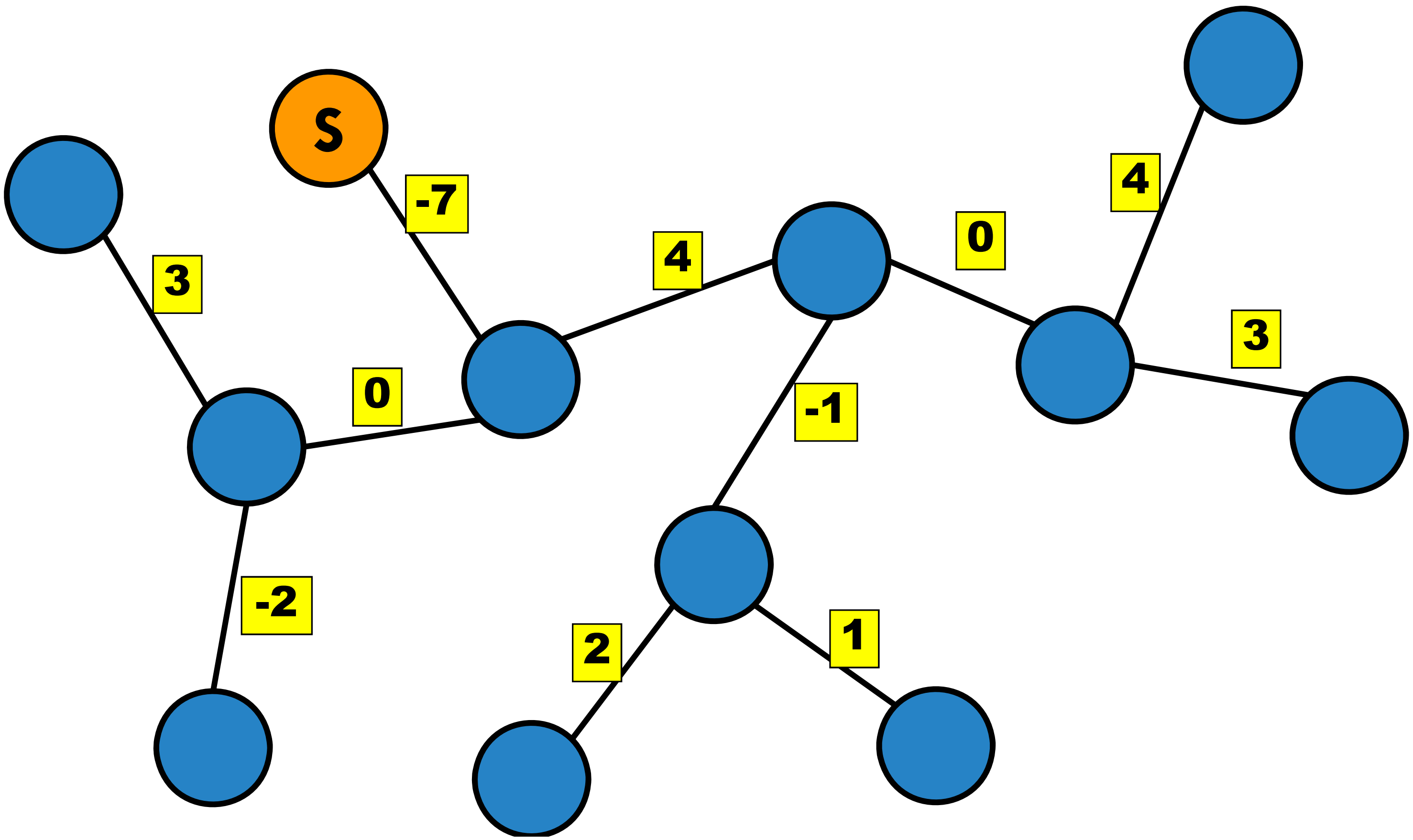
Richard Bellman



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	
On Tree	BFS / DFS	
On DAG	Topological Sort	

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?

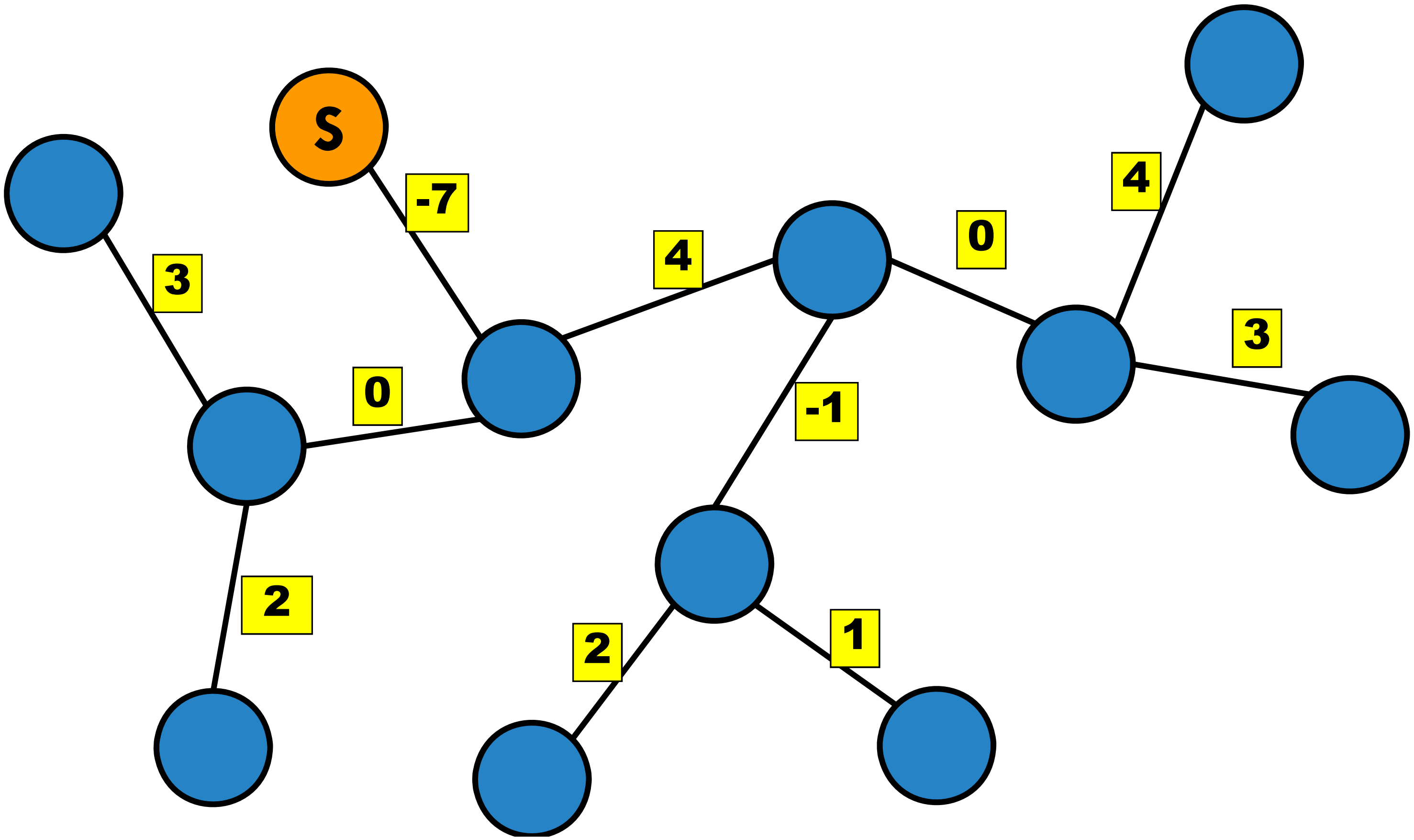
- A tree with a special designated root note.

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

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

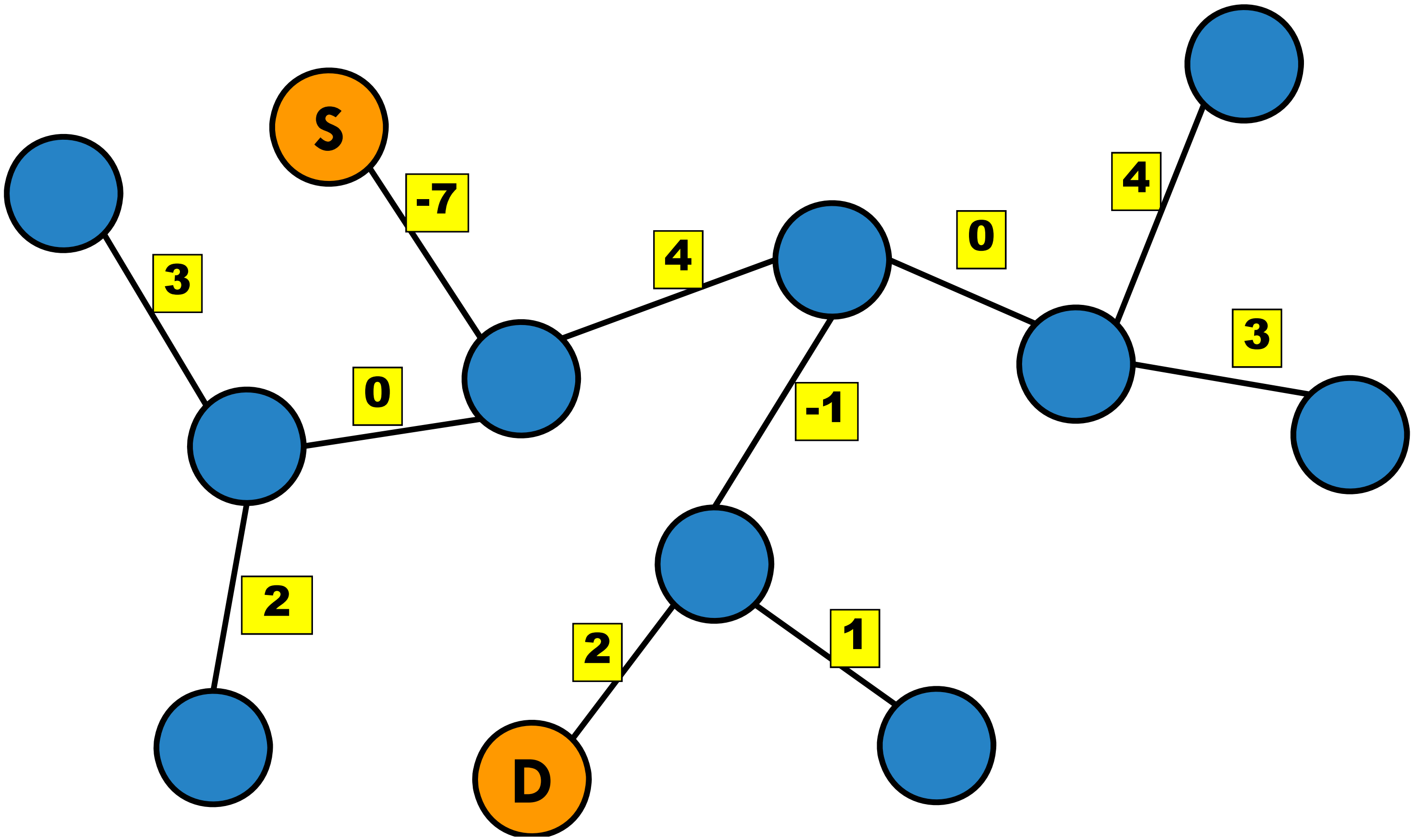
UNDIRECTED WEIGHTED TREE

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



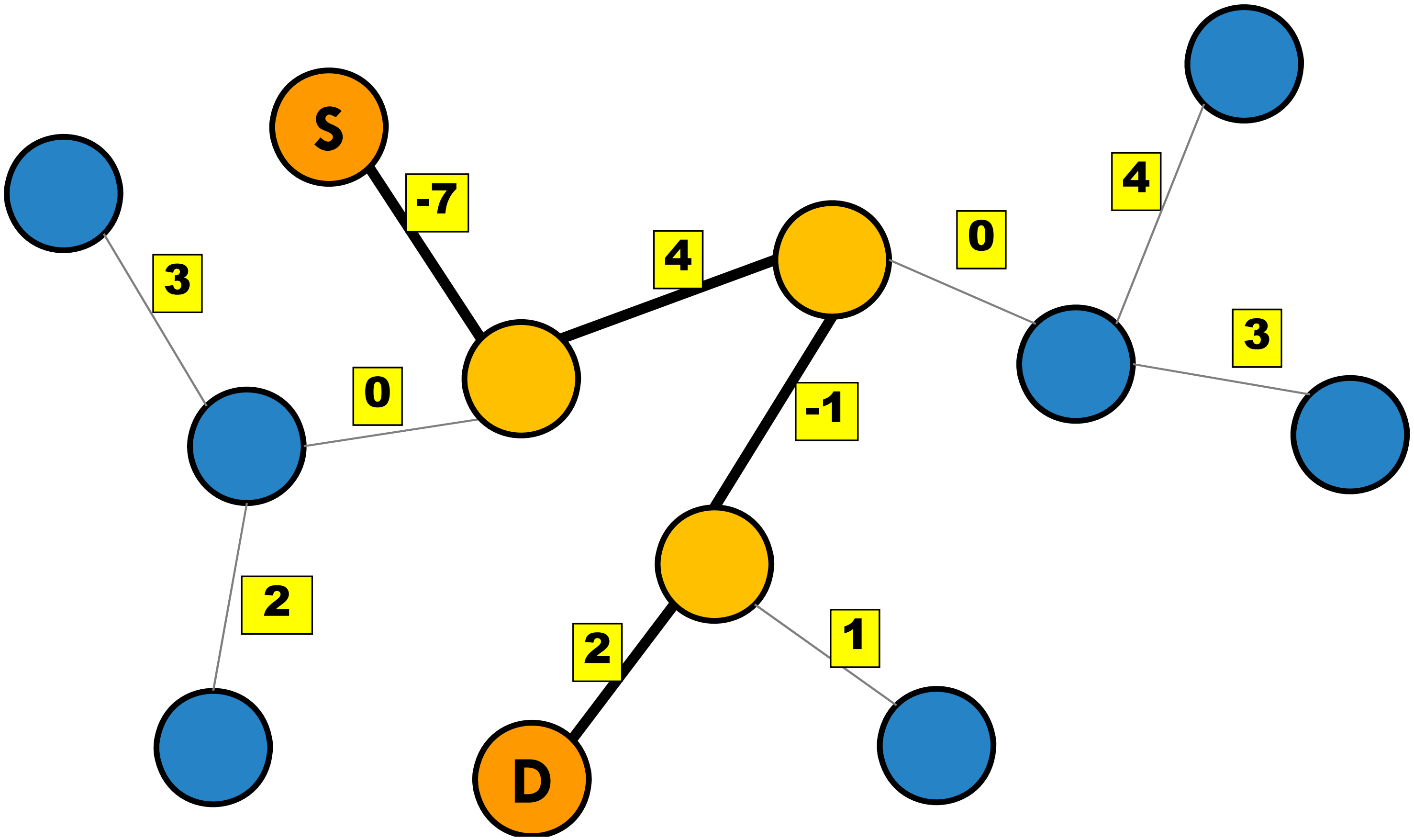
UNDIRECTED WEIGHTED TREE

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



UNDIRECTED WEIGHTED TREE

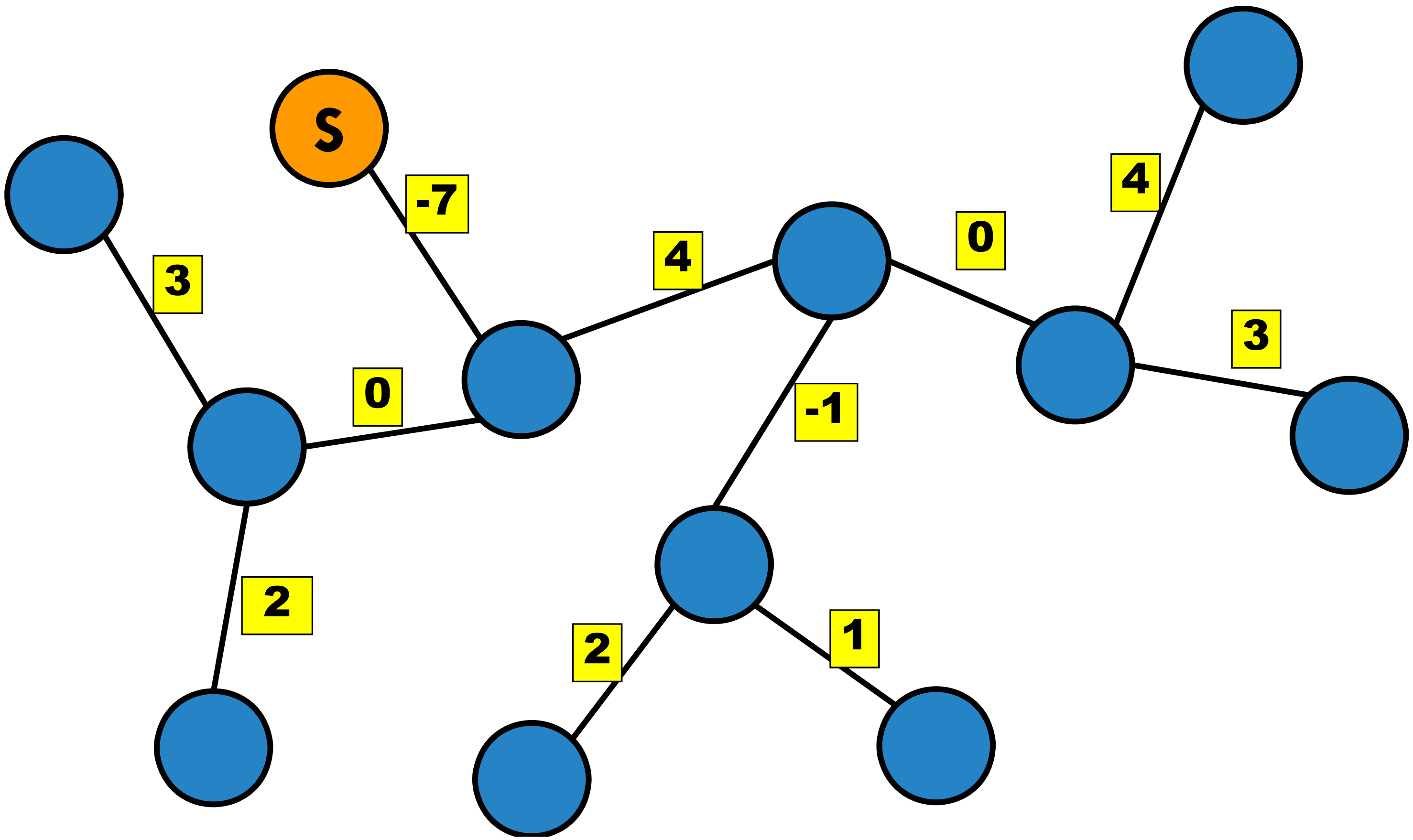
Just 1 way! It's a tree!



TREE: SOURCE-TO-ALL

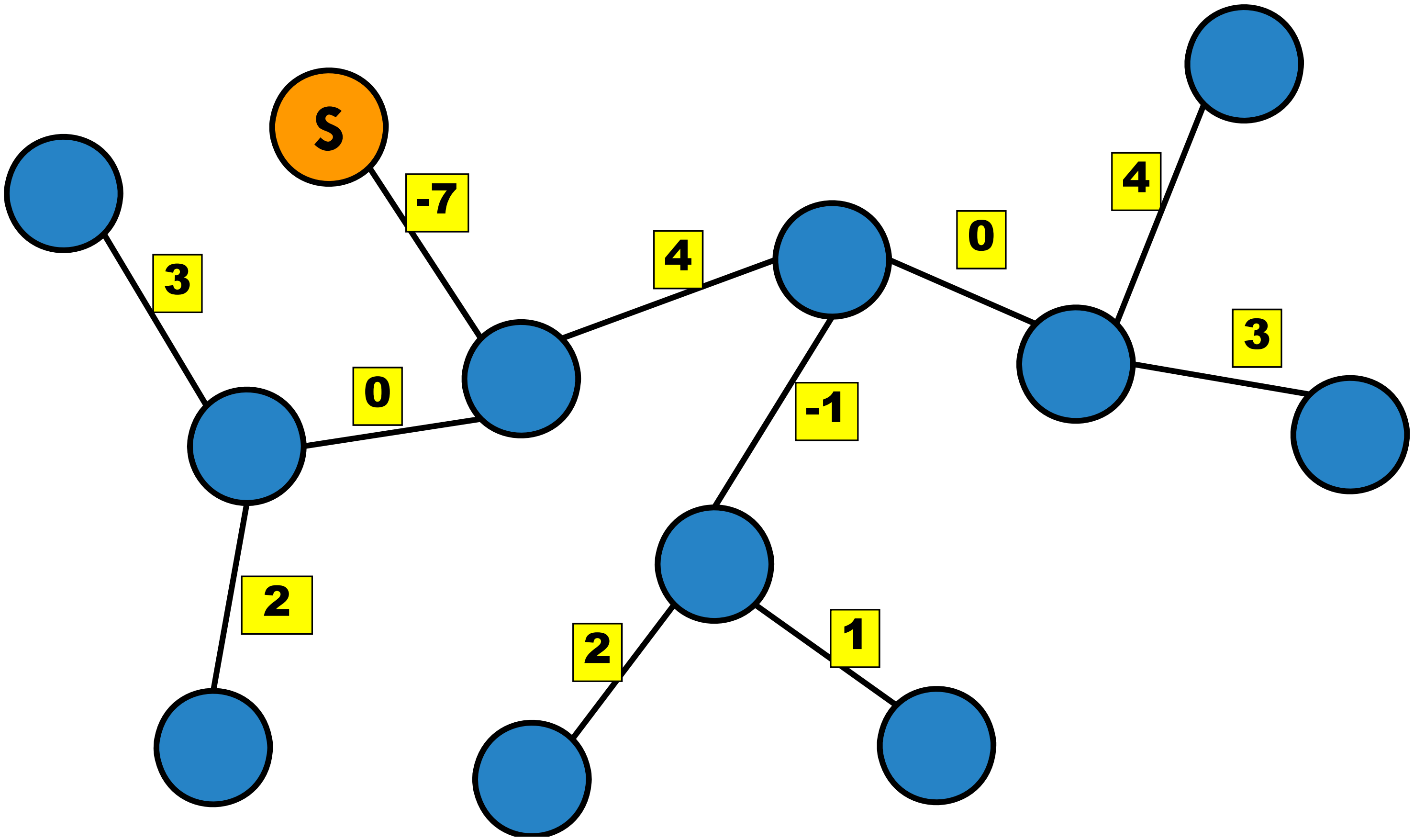
In what order should we relax the nodes?

Use DFS or BFS



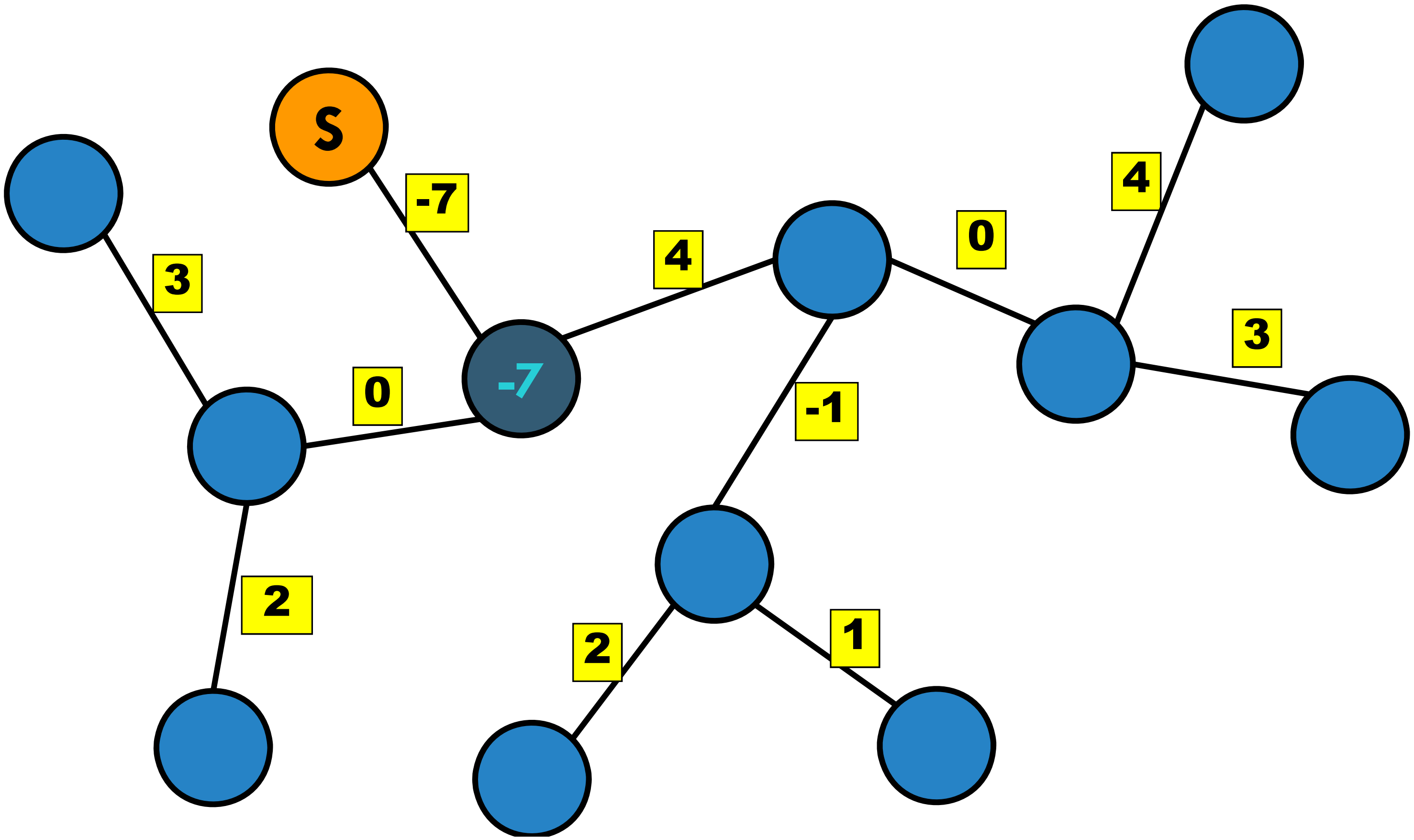
TREE: SOURCE-TO-ALL

Relax in DFS Order



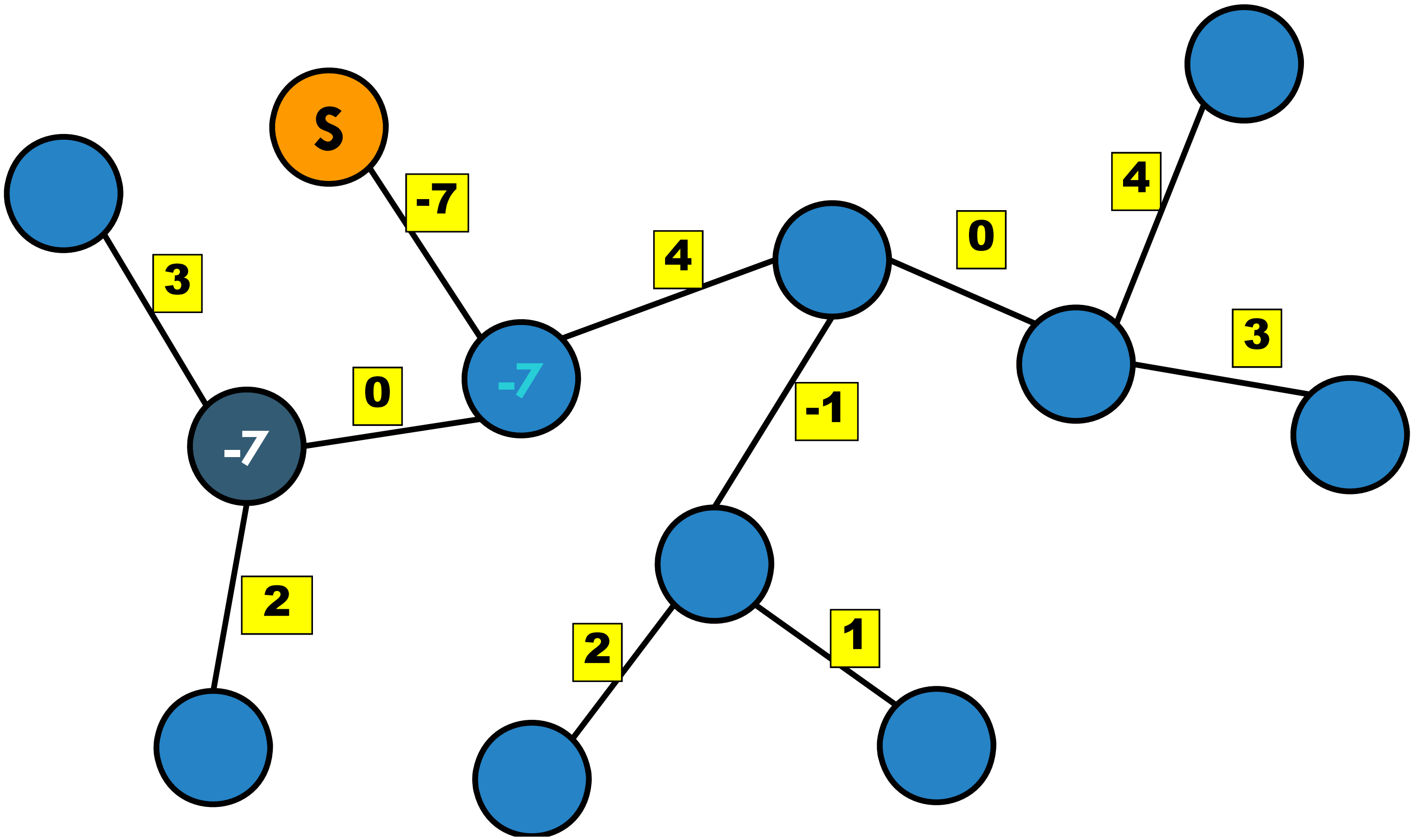
TREE: SOURCE-TO-ALL

Relax in DFS Order



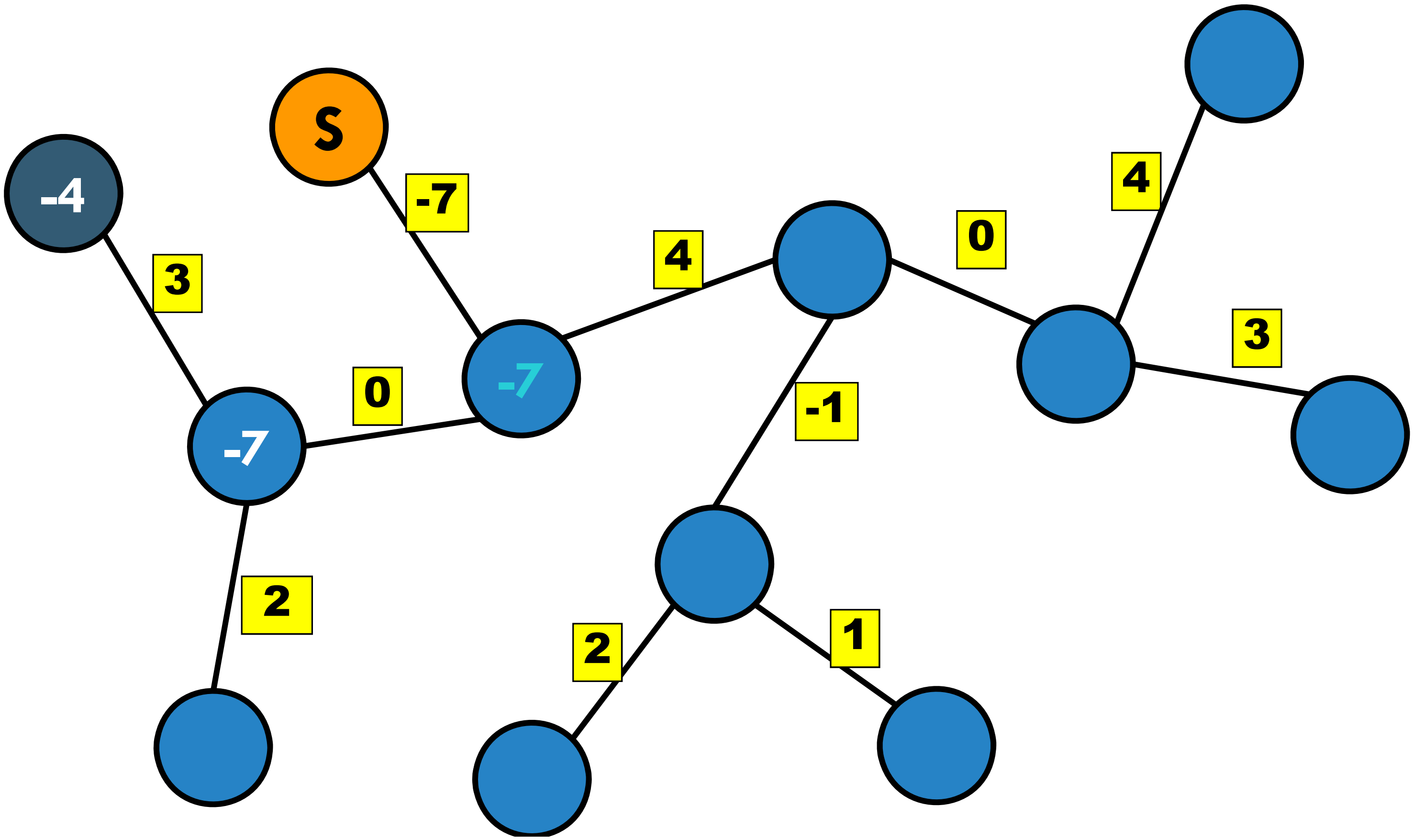
TREE: SOURCE-TO-ALL

Relax in DFS Order



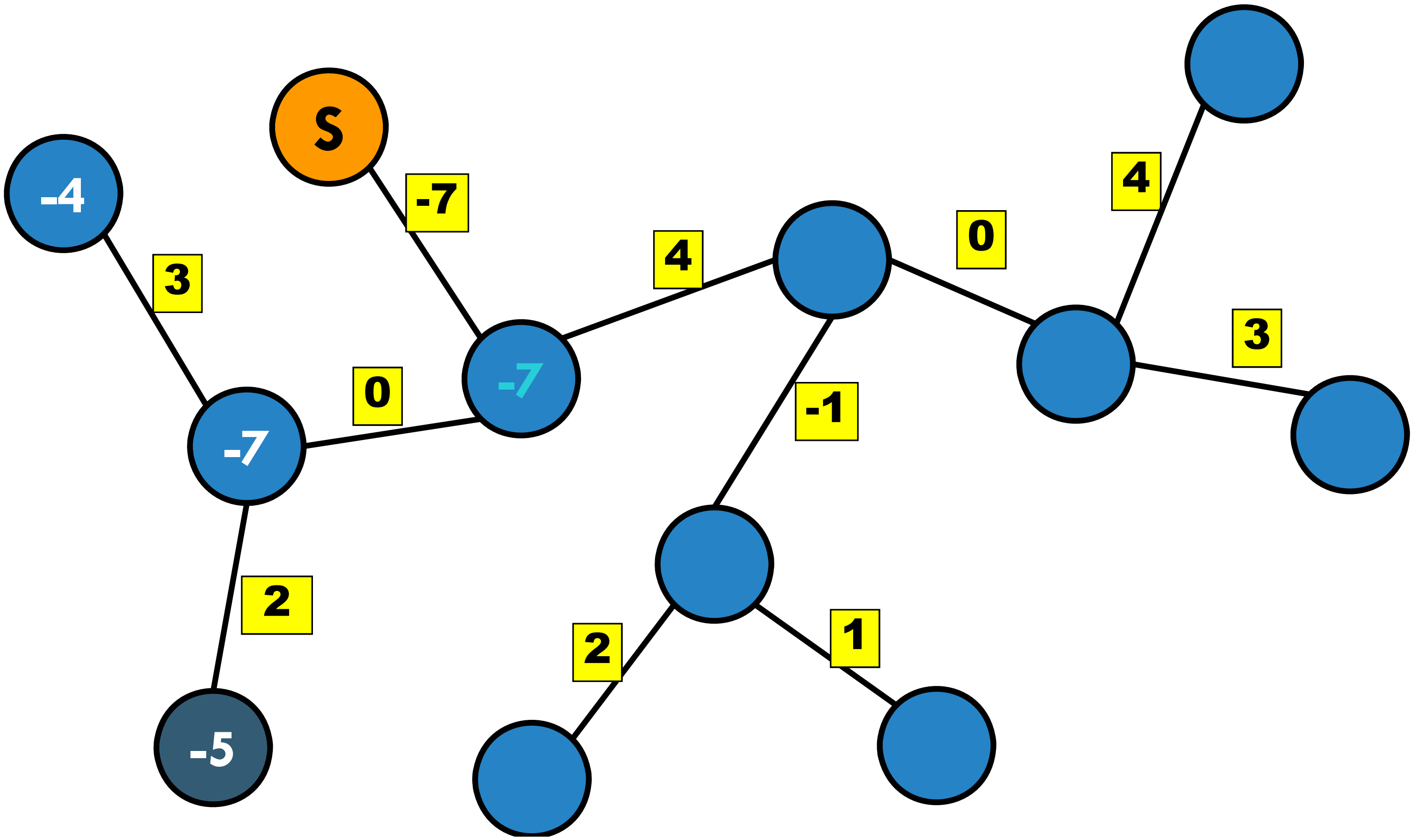
TREE: SOURCE-TO-ALL

Relax in DFS Order



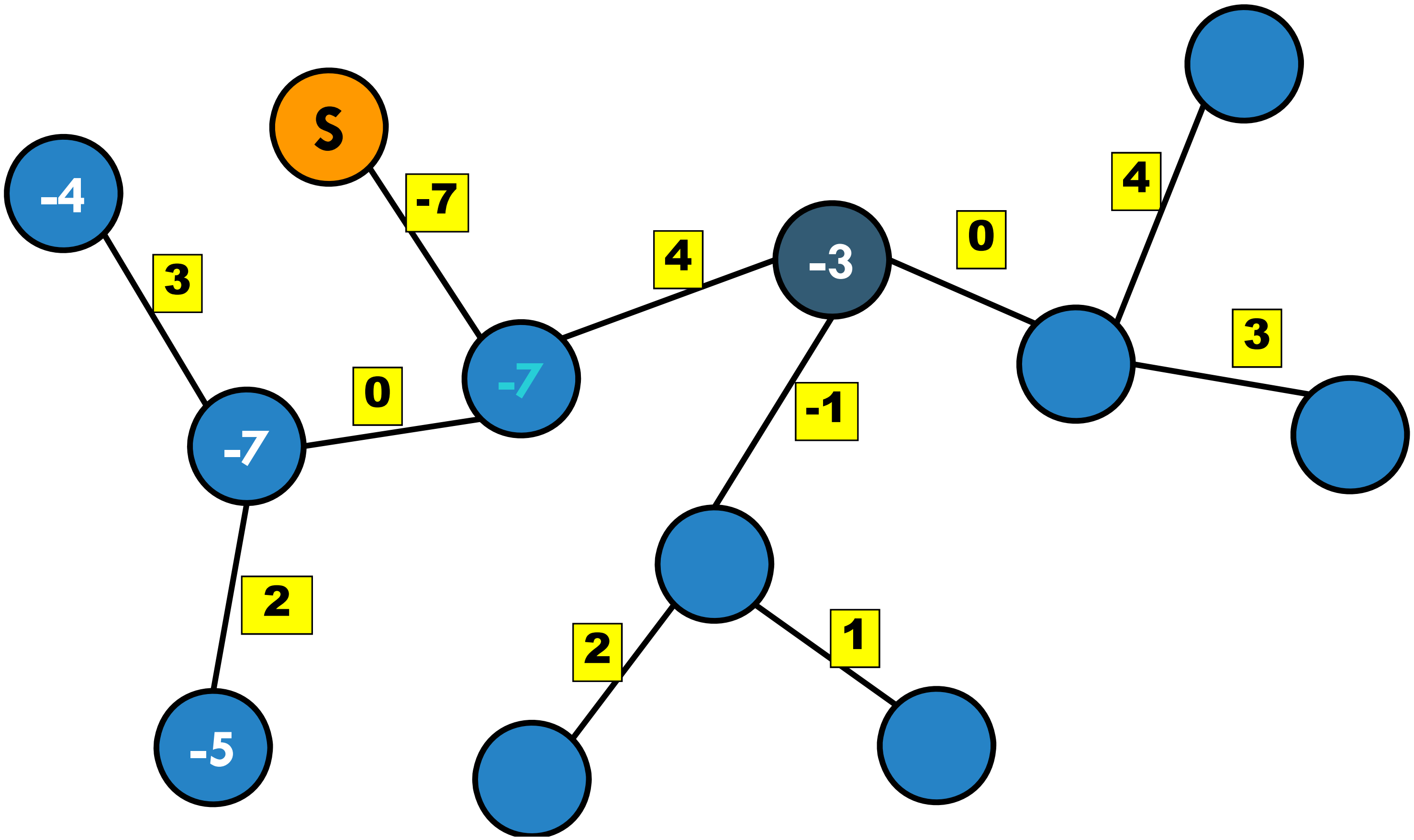
TREE: SOURCE-TO-ALL

Relax in DFS Order



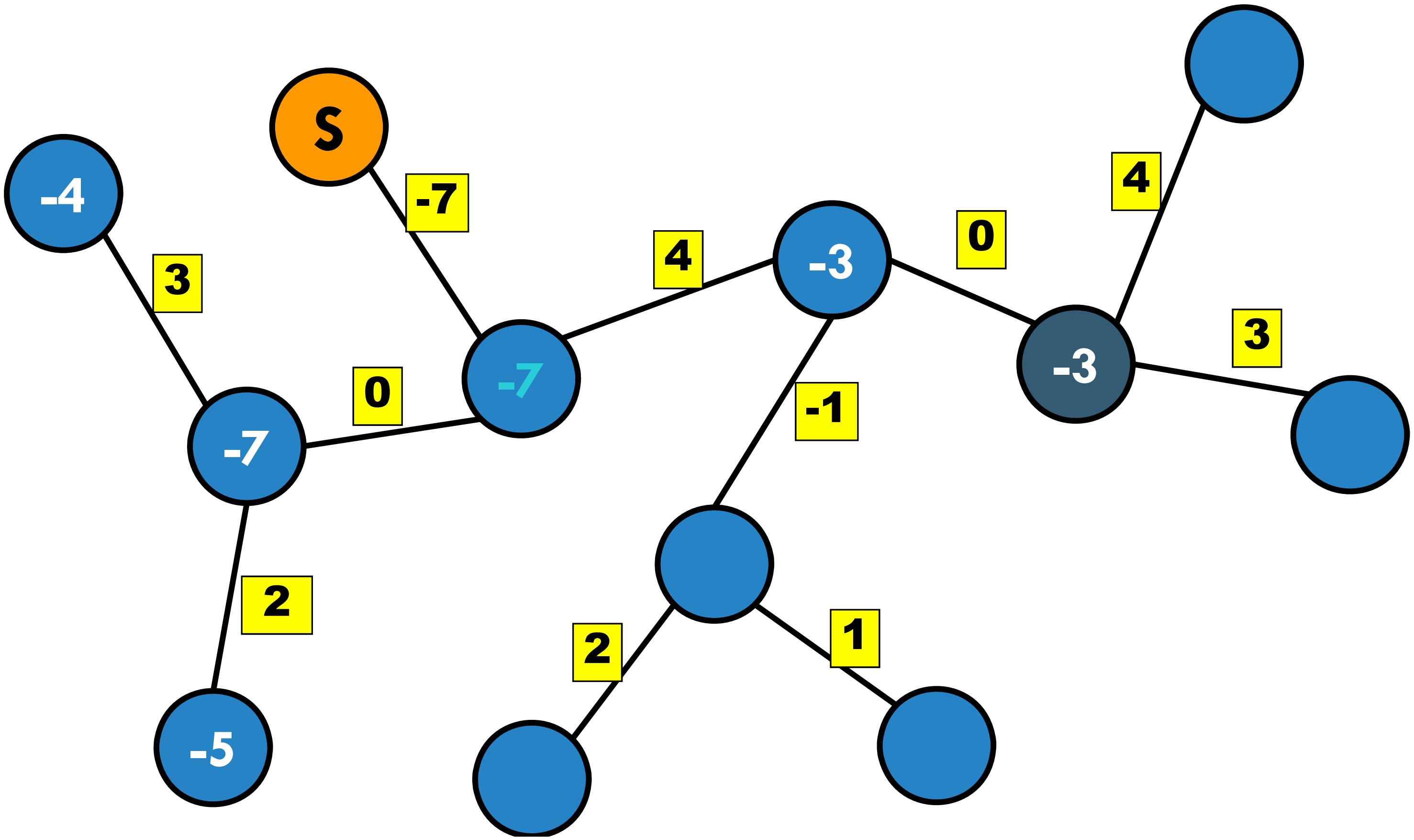
TREE: SOURCE-TO-ALL

Relax in DFS Order



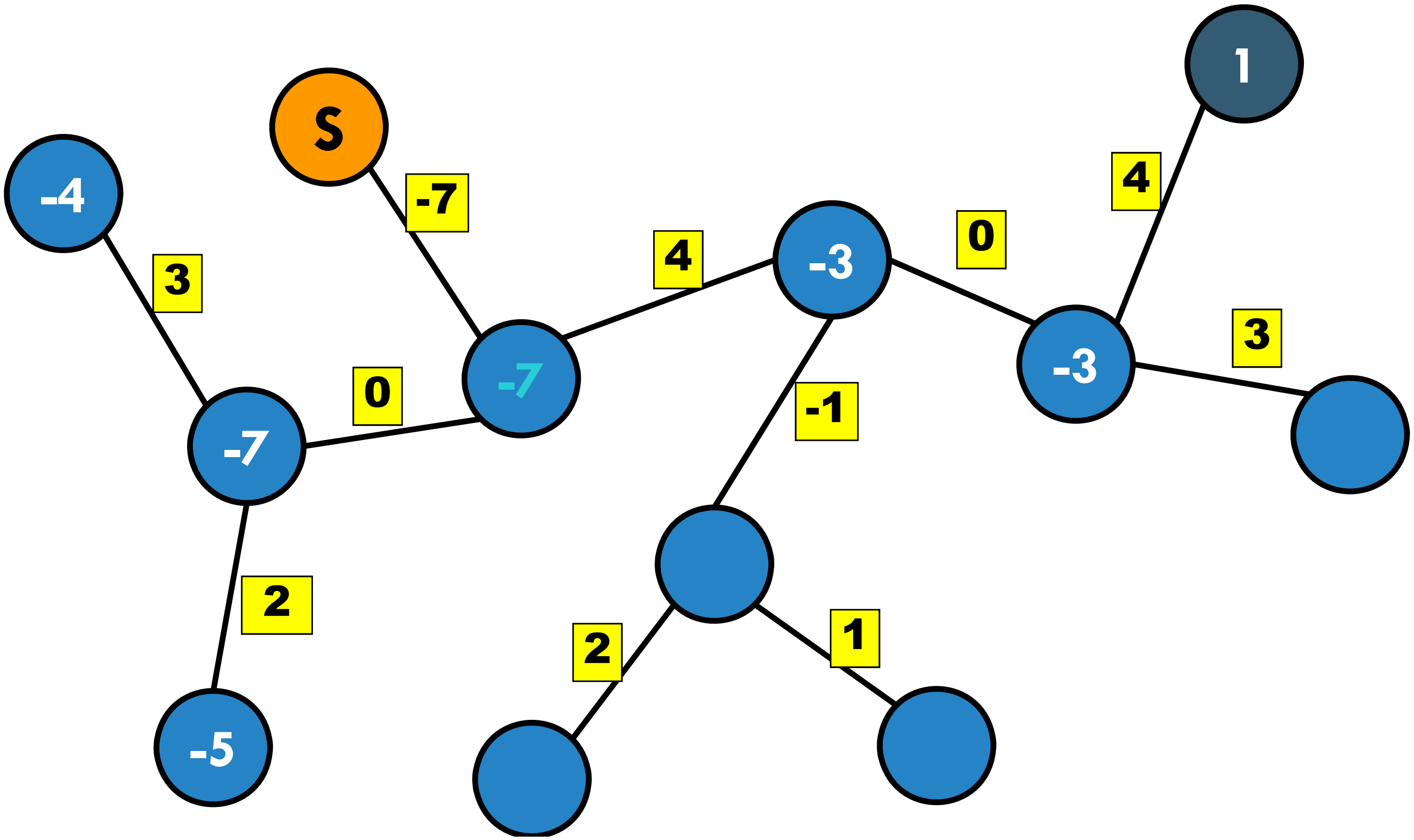
TREE: SOURCE-TO-ALL

Relax in DFS Order



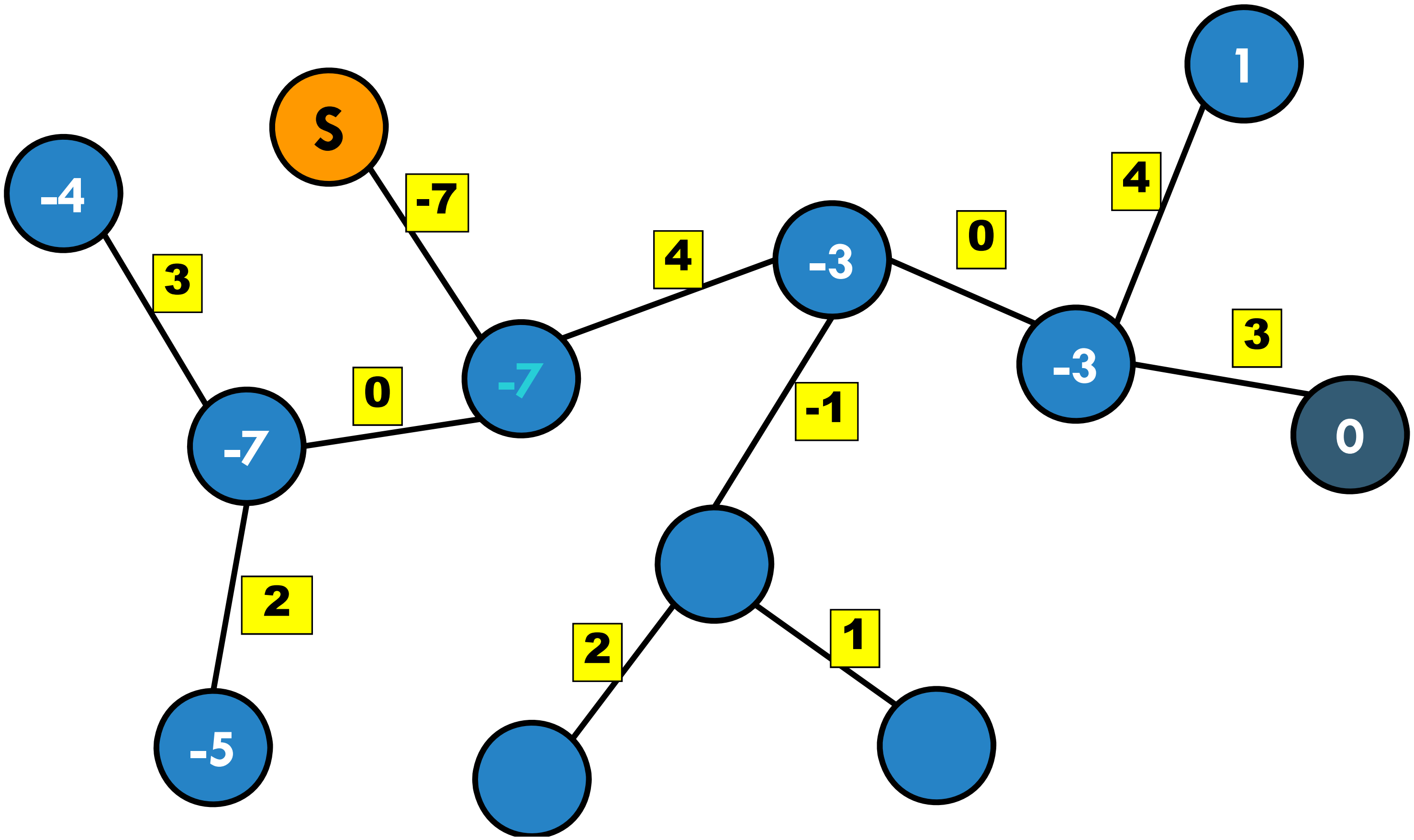
TREE: SOURCE-TO-ALL

Relax in DFS Order



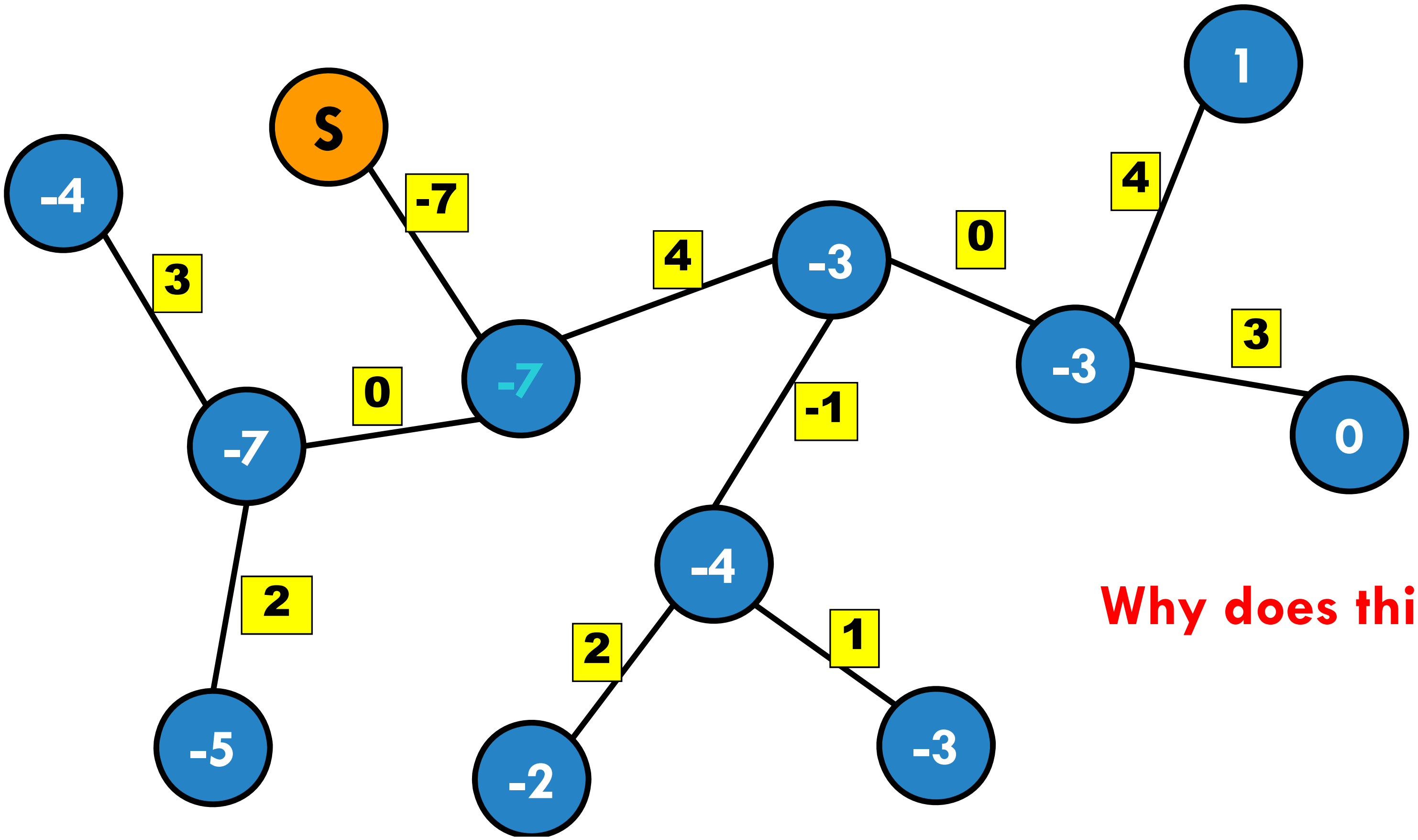
TREE: SOURCE-TO-ALL

Relax in DFS Order



TREE: SOURCE-TO-ALL

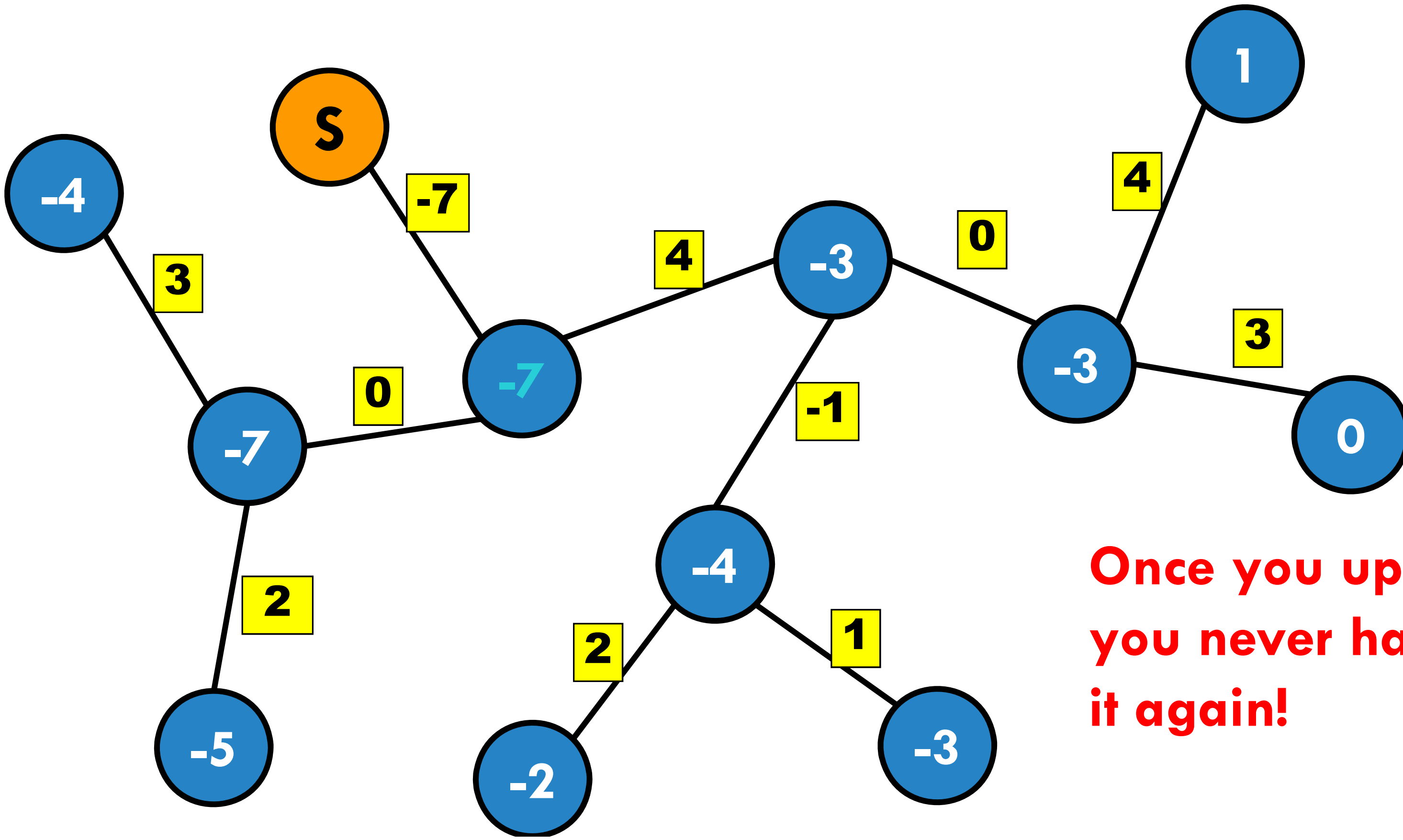
Relax in DFS Order



Why does this work?

TREE: SOURCE-TO-ALL

Relax in DFS Order



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

TREE: SOURCE-TO-ALL

Special case:

- Positive or negative weights
- Undirected tree (no backpedaling)

Basic idea:

- Perform DFS or BFS
- Relax each edge the first time you see it.

What is the running time?

A. $O(V)$
B. $O(E)$
C. $O(V+E)$
D. $O(VE)$
E. Naturo says it is **not** C.

TREE: SOURCE-TO-ALL

Special case:

- Positive or negative weights
- Undirected tree (no backpedaling)

Basic idea:

- Perform DFS or BFS
- Relax each edge the first time you see it.

how many edges in a tree?

What is the running time?

A. $O(V)$

B. $O(E)$

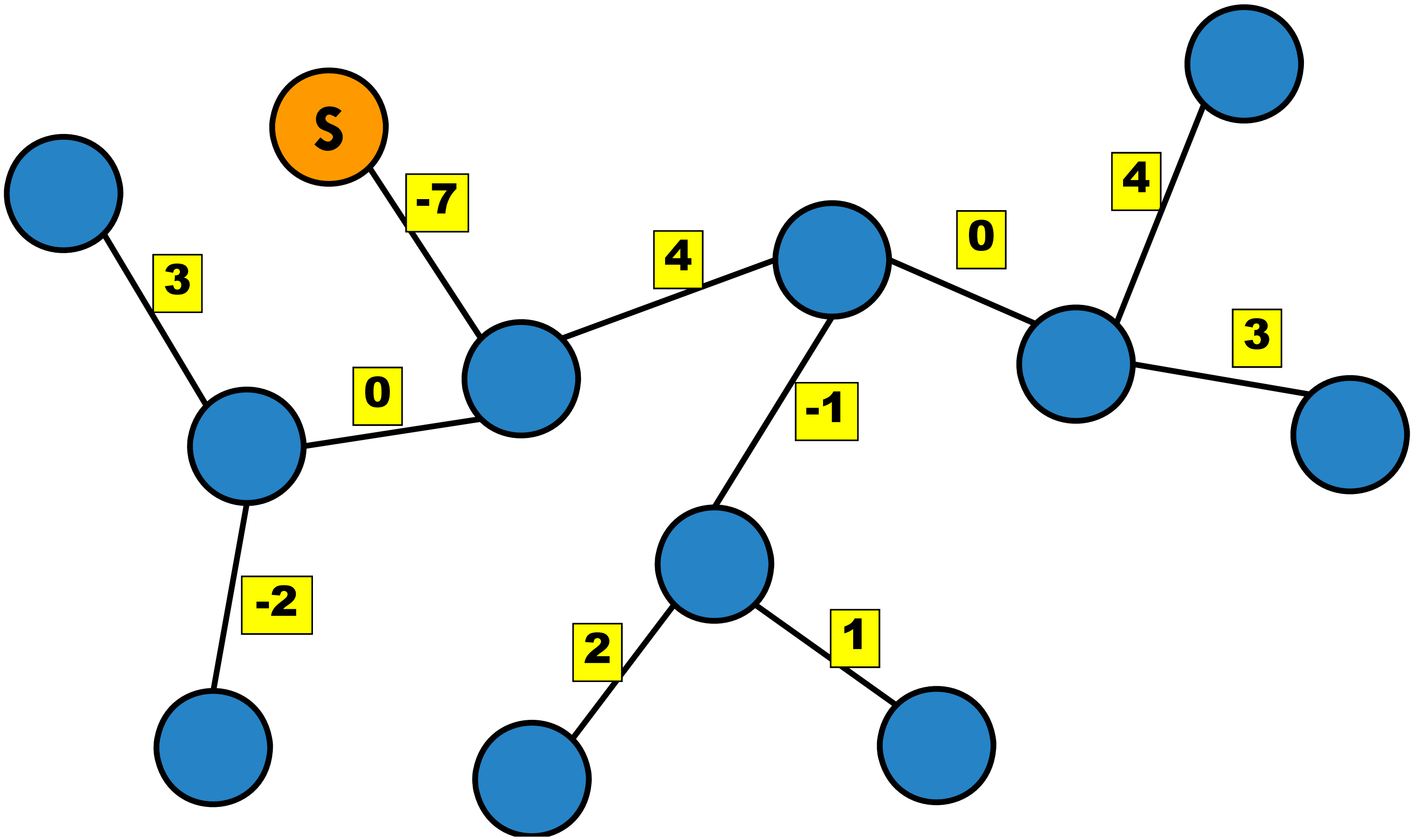
C. $O(V+E)$

D. $O(VE)$

E. Naturo says it is **not** C.

UNDIRECTED WEIGHTED TREE

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



SUMMARY

By the end of the session, students should be able to:

- describe the **shortest path algorithm** for **unweighted graphs**
- explain the **Bellman-Ford algorithm**
- describe the time complexity of the **Bellman-Ford algorithm**
- Understand when **Bellman-Ford will fail**

NEXT WEEK: 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	
On Tree	BFS / DFS	$O(V)$
On DAG	Topological Sort	