# CS2040S
# Data Structures and Algorithms

Shortest Paths!

# Admin

Midterm Scripts Published

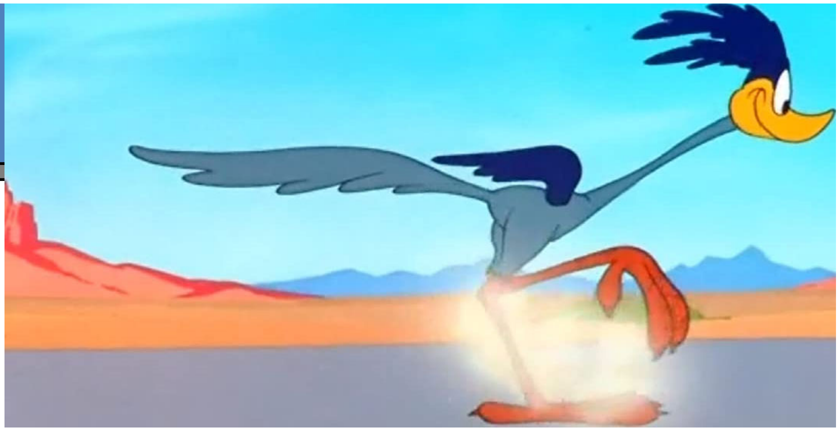
- You should have received your script by e-mail.
- Check Luminus Gradebook! These are the official score.
- If there is a mistake (e.g., an OCR error, or incorrect transfer to Luminus), send Prof. Ben Leong an e-mail by Wed. 23:59.
- After Wednesday, grades are final.

# Admin

## Midterm Scripts Published

- You should have received your script by e-mail.
- Check Luminus Gradebook!  These are the official score.
- If there is a mistake (e.g., an OCR error, or incorrect transfer to Luminus), send Prof. Ben Leong an e-mail by Wed. 23:59.
- After Wednesday, grades are final.

# Speed Demon Competition

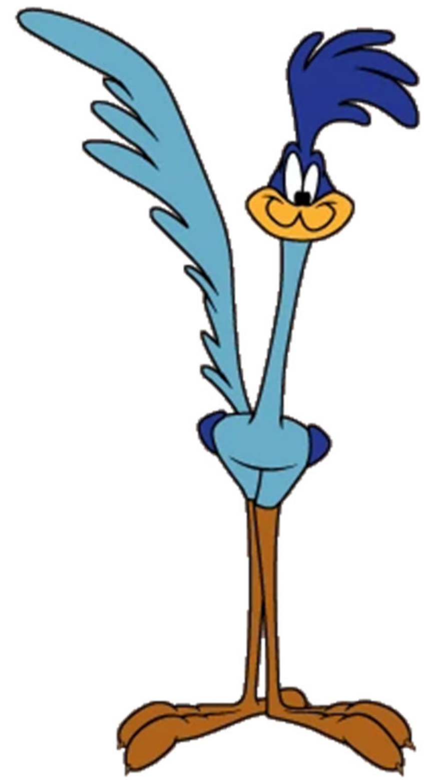


Announcing the winners!

# Speed Demon Competition

## CS2040S Speed Demon Leaderboard

| Name | Runtime |
|---|---|
| Tan Kel Zin | 2.234s |
| Cai Kai'An | 2.24s |
| Tee Weile Wayne | 2.252s |
| Ryan Chung Yi Sheng | 2.291s |
| Tan Weiu Cheng | 2.358s |
| Zhang Shichen | 2.369s |
| Rohit Rajesh Bhat | 2.376s |
| Lee Xiong Jie, Isaac | 2.546s |
| Marcus Tang Xin Kye | 2.789s |
| Lee Zheng Han | 2.792s |



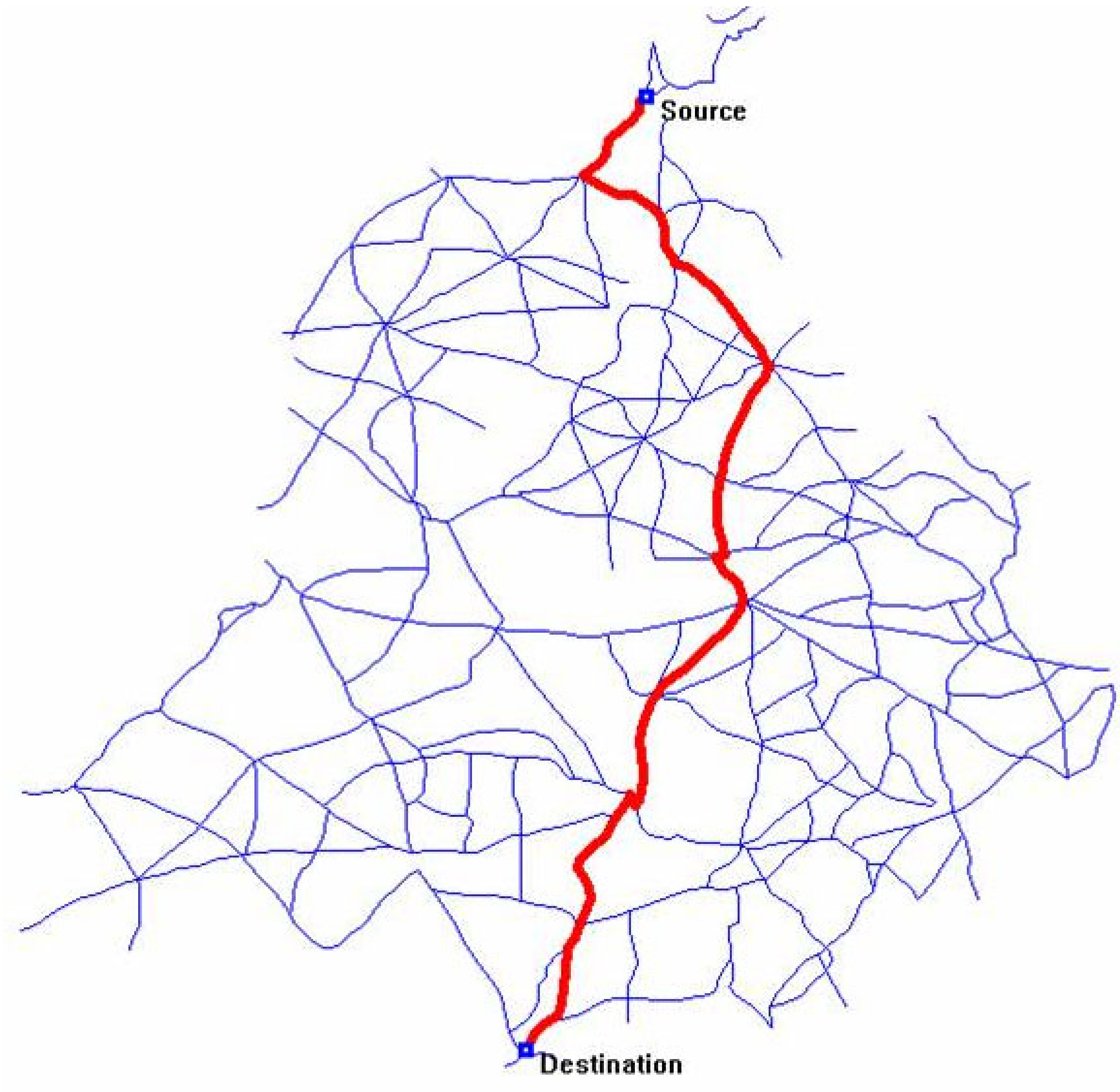
# Last Week

Introduction to Graphs

- What is a graph?  What is a directed graph?
- Modelling problems as graphs.
- Graph representations.
- Graph searching: BFS/DFS

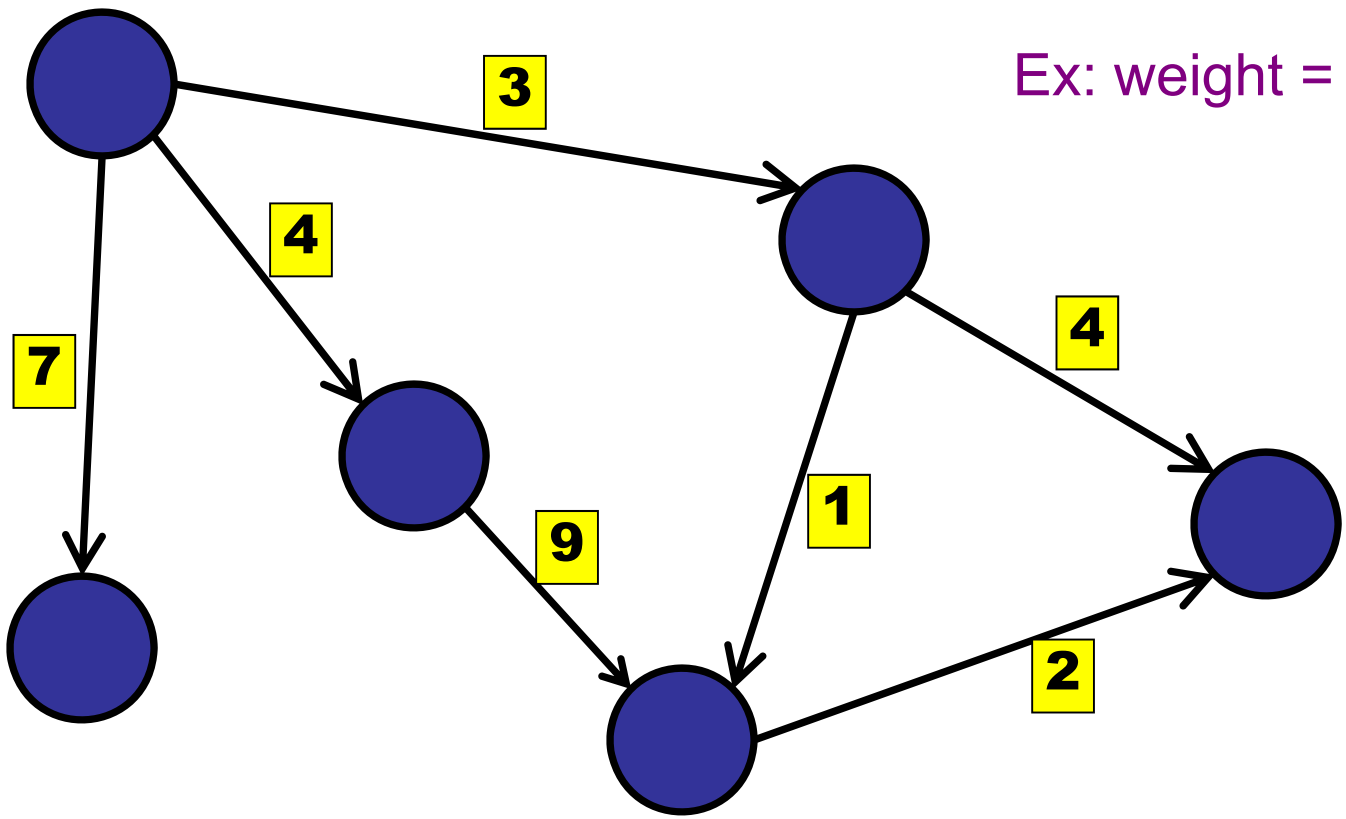
Postponed: topologic order, topological sort

# SHORTEST PATHS



# Weighted Graphs

**Edge weights:** w(e) : E ➔ R

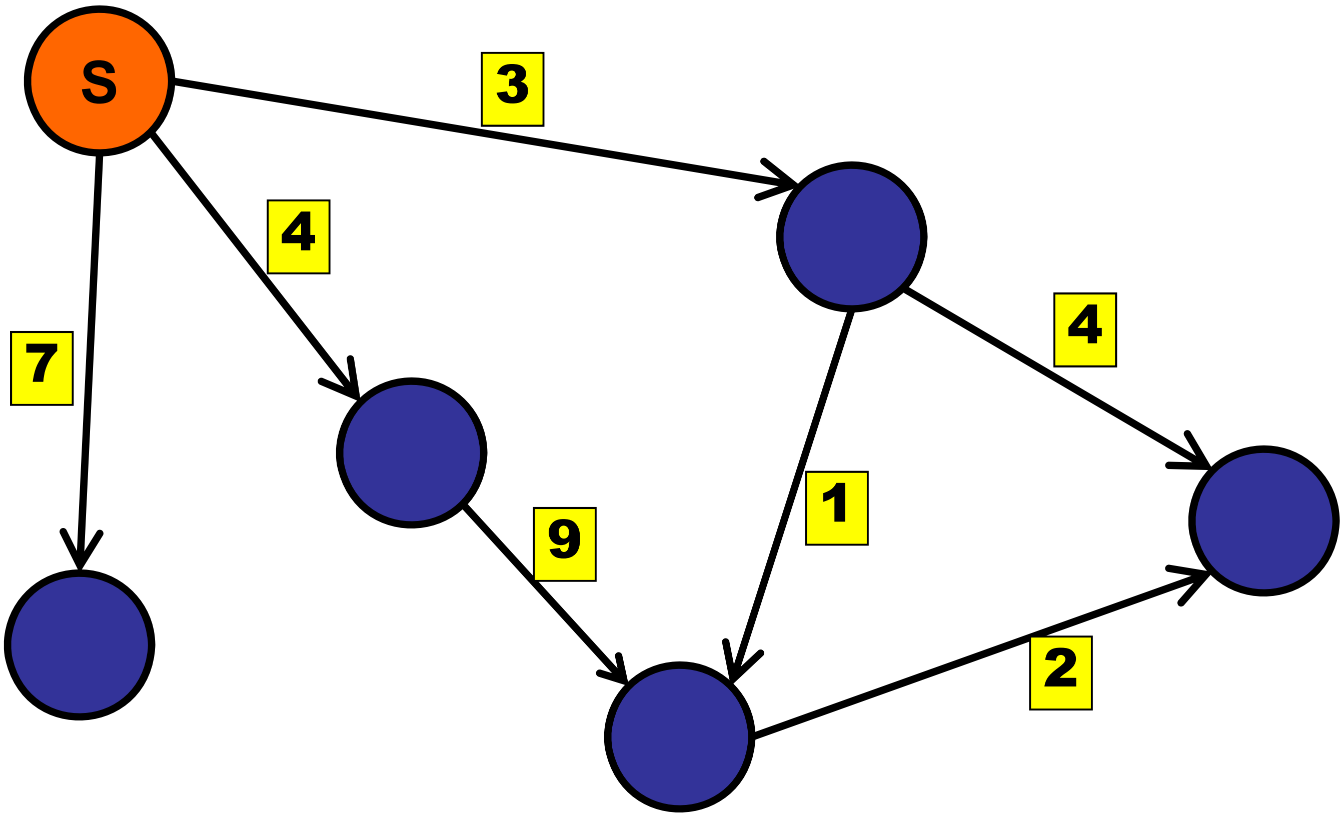



Ex: weight = distance

Adjacency list: stores weights with edge in NbrList

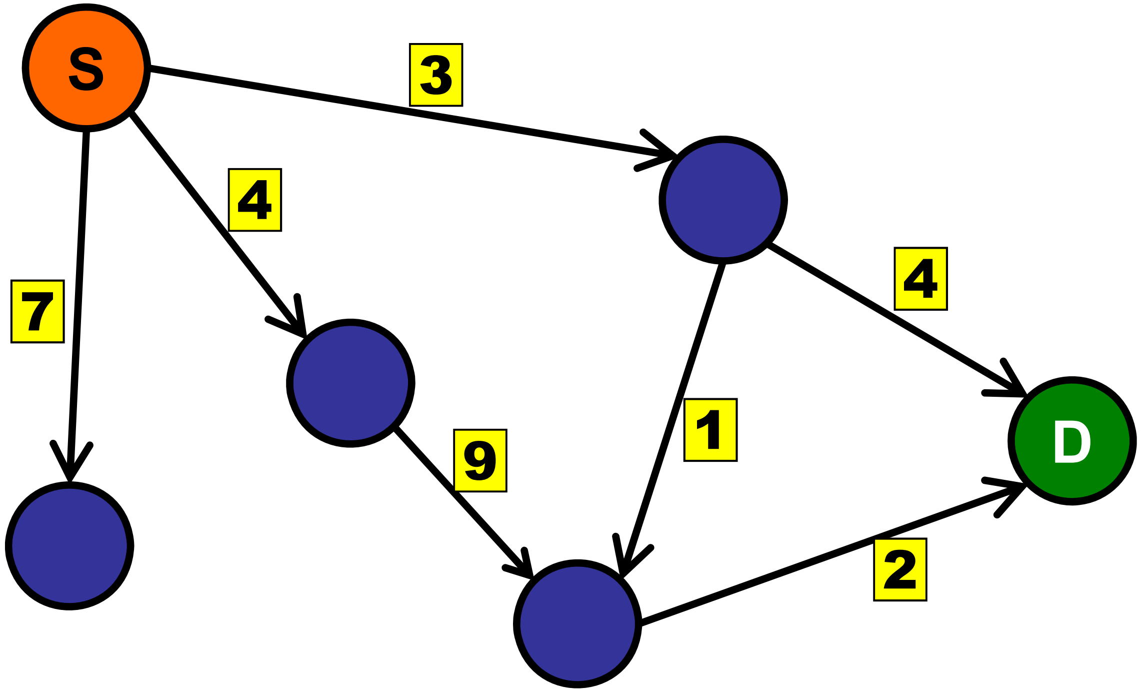
# Shortest Paths

Distance from source?



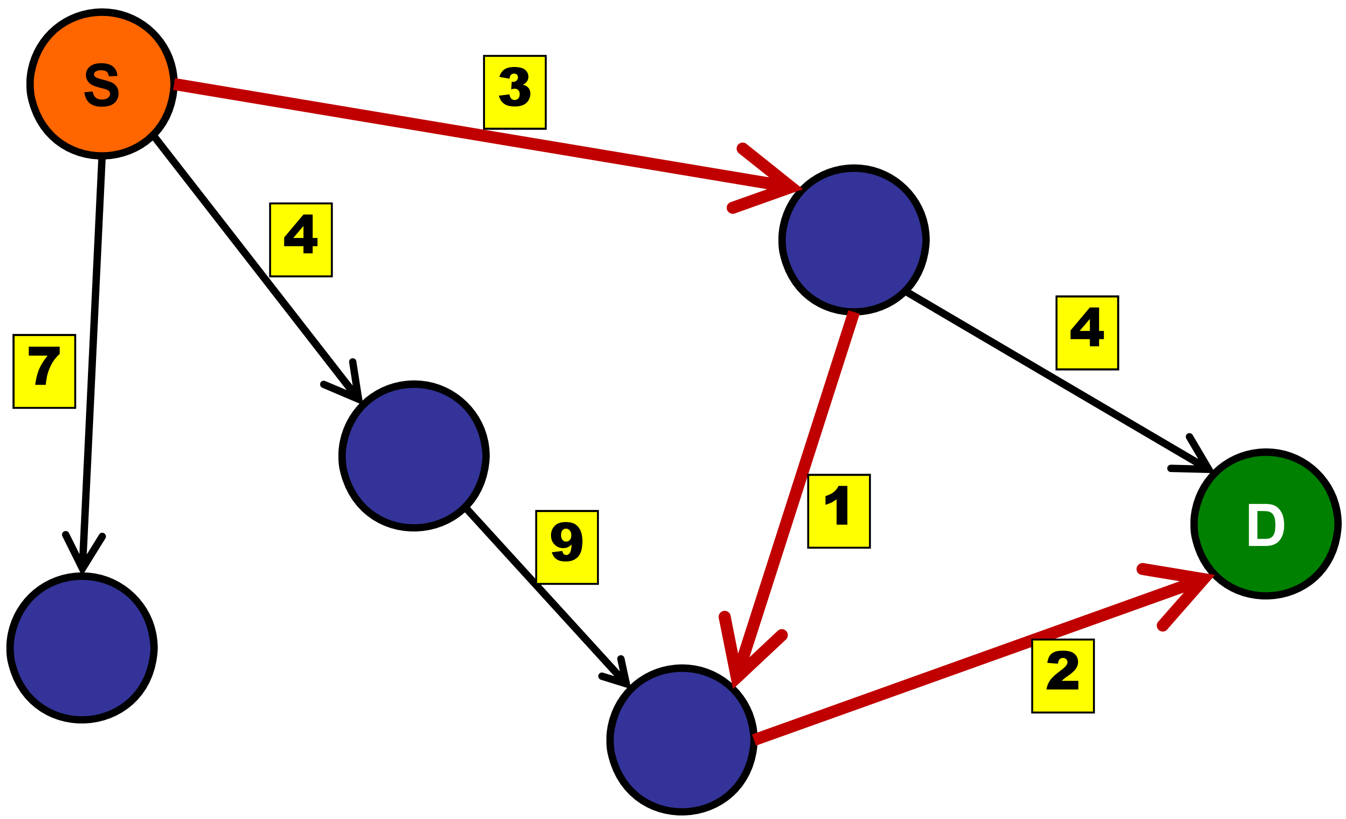
## What is the distance from S to D?

1. 2
2. 4
3. 6
4. 7
5. 9
6. Infinite



# Shortest Paths

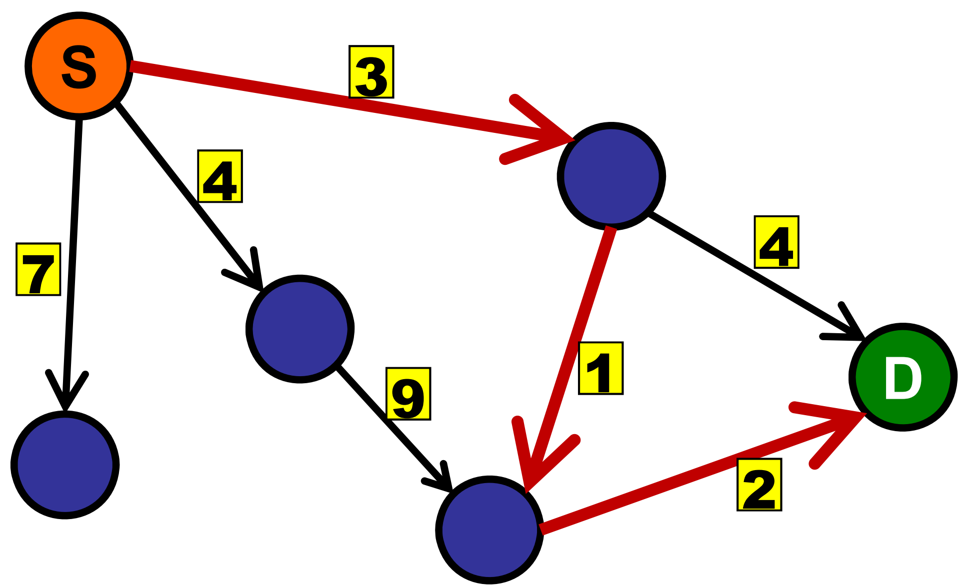
Distance from source?



# Shortest Paths

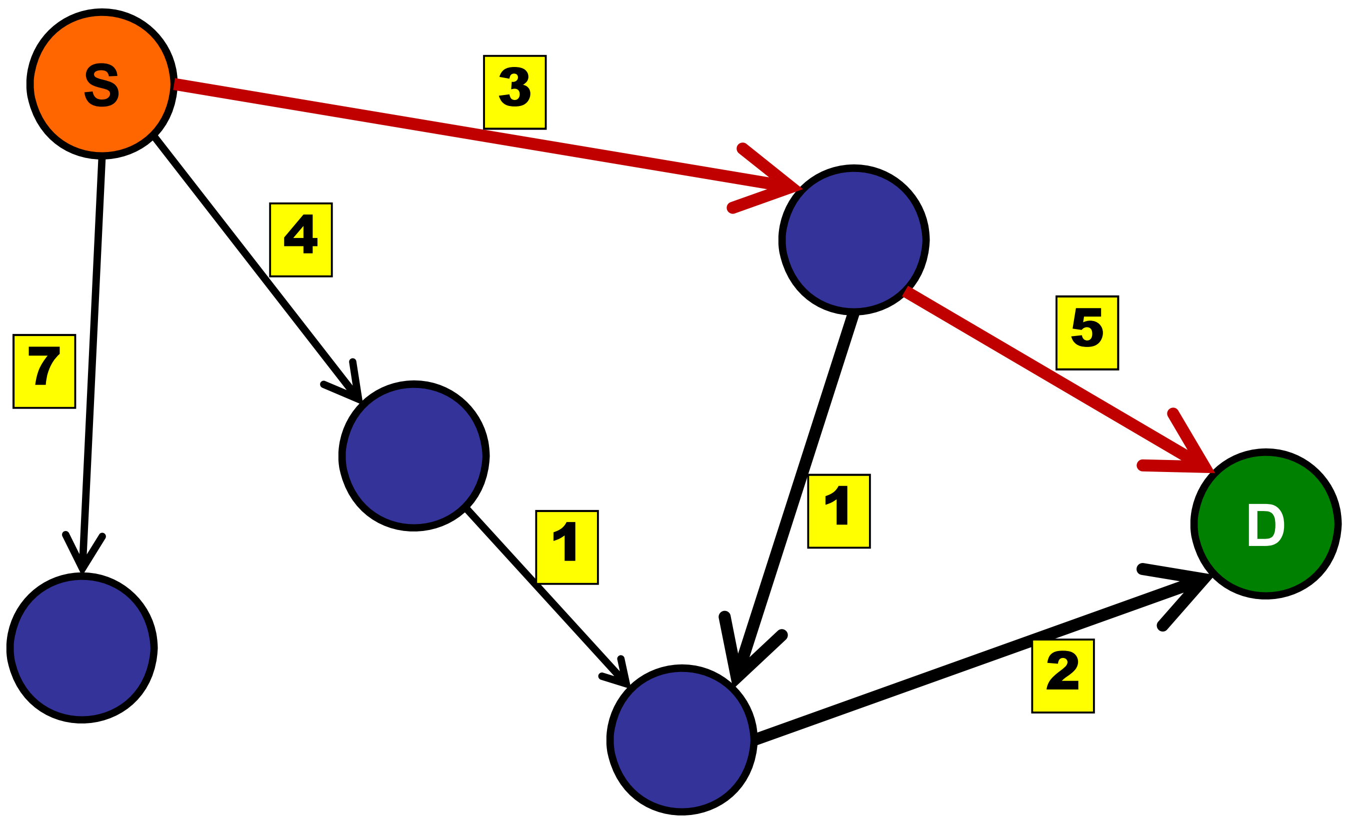
Questions:

- 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.



# Shortest Paths

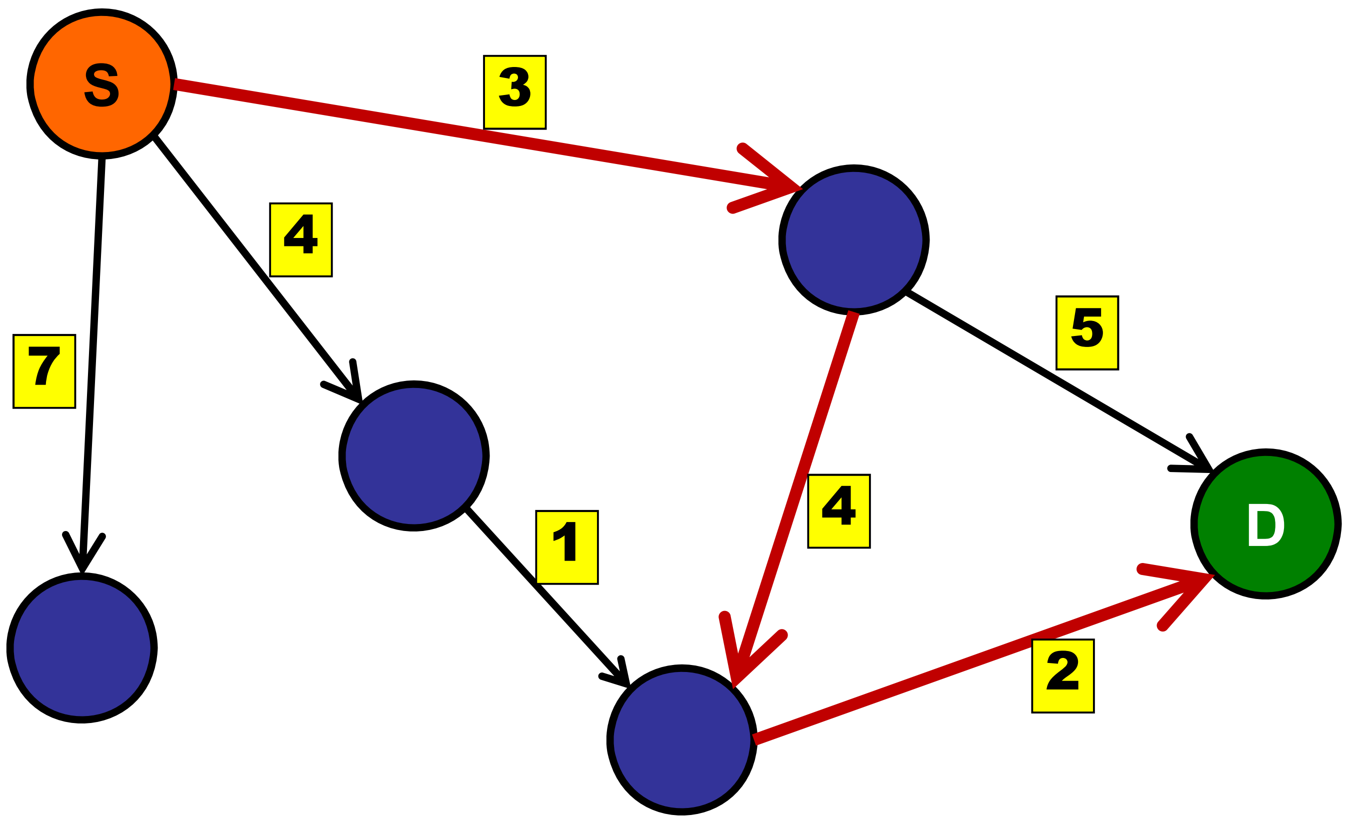
Common mistake: "Why can't I use BFS?"




Remember: BFS does not explore every path in the graph!

# Shortest Paths

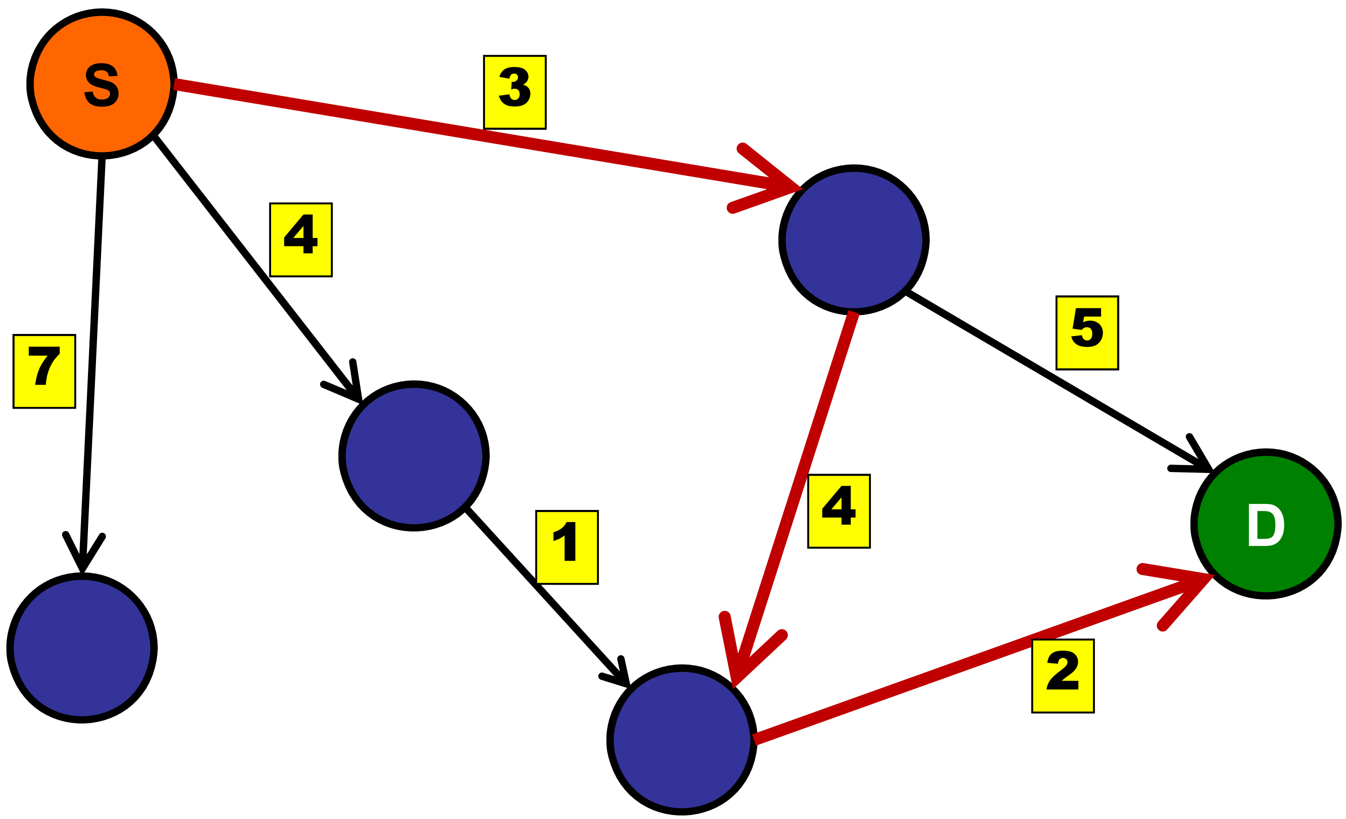
Common mistake: “Why can’t I use BFS?”




Remember: BFS does not explore every path in the graph!

# Shortest Paths

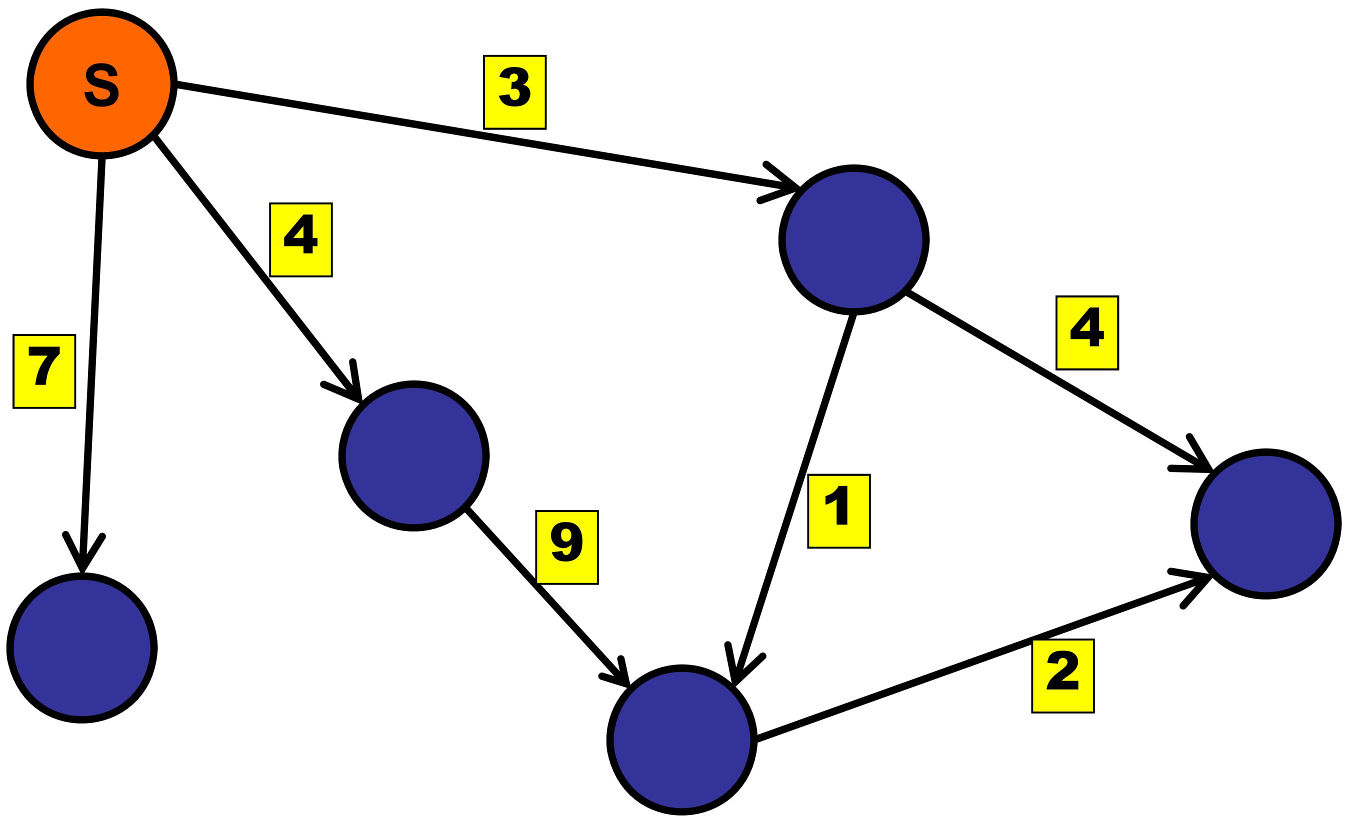
Common mistake: “Why can’t I use BFS?”




BFS finds minimum number of HOPS not minimum DISTANCE.

# Shortest Paths

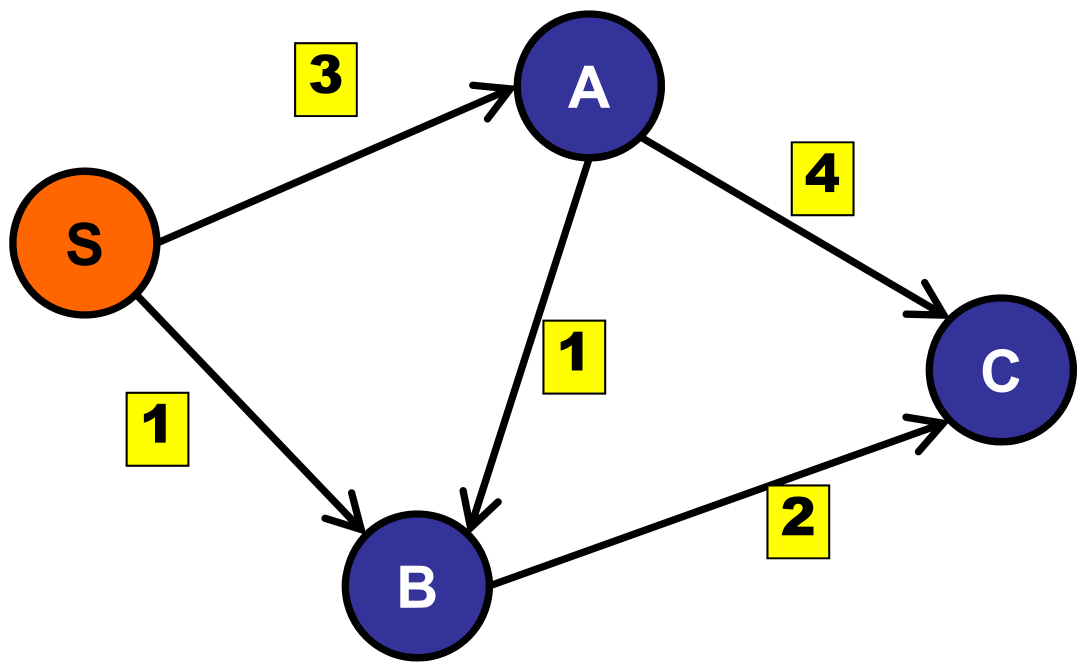
Notation: $\delta(u,v)$ = distance from u to v



# Shortest Paths

Key idea: triangle inequality

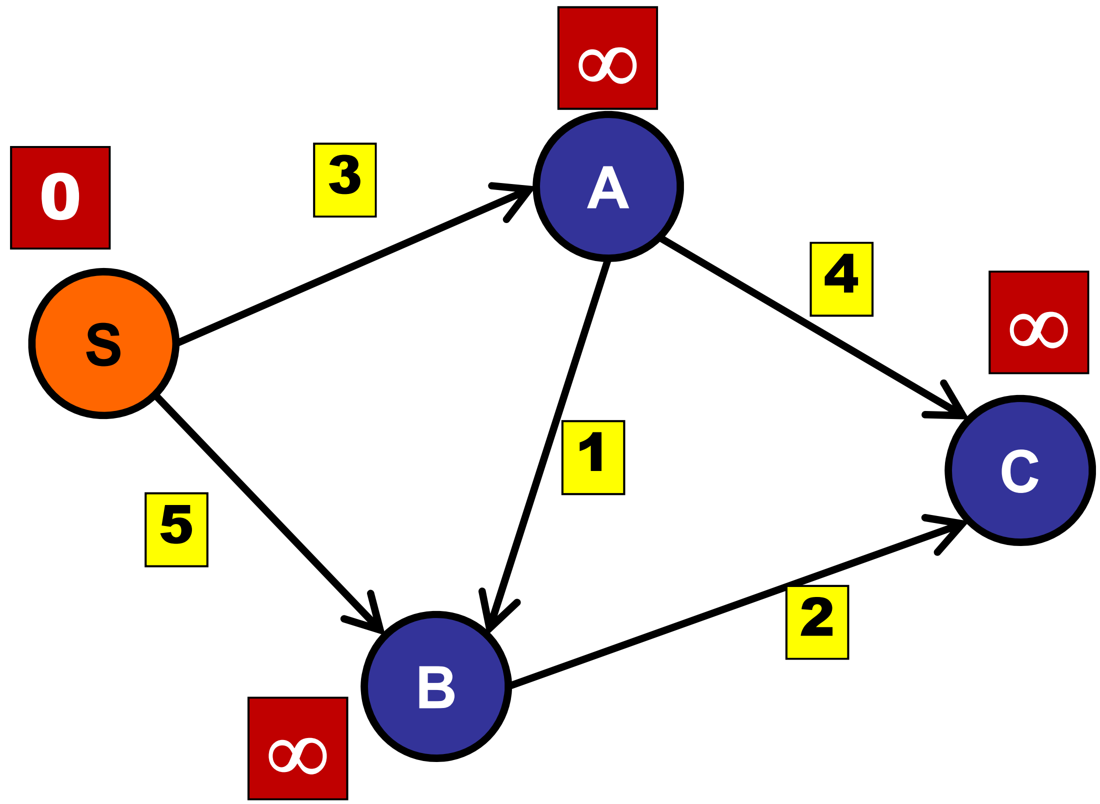
$\delta(S, C) \leq \delta(S, A) + \delta(A, C)$



# Shortest Paths

Maintain estimate for each distance:

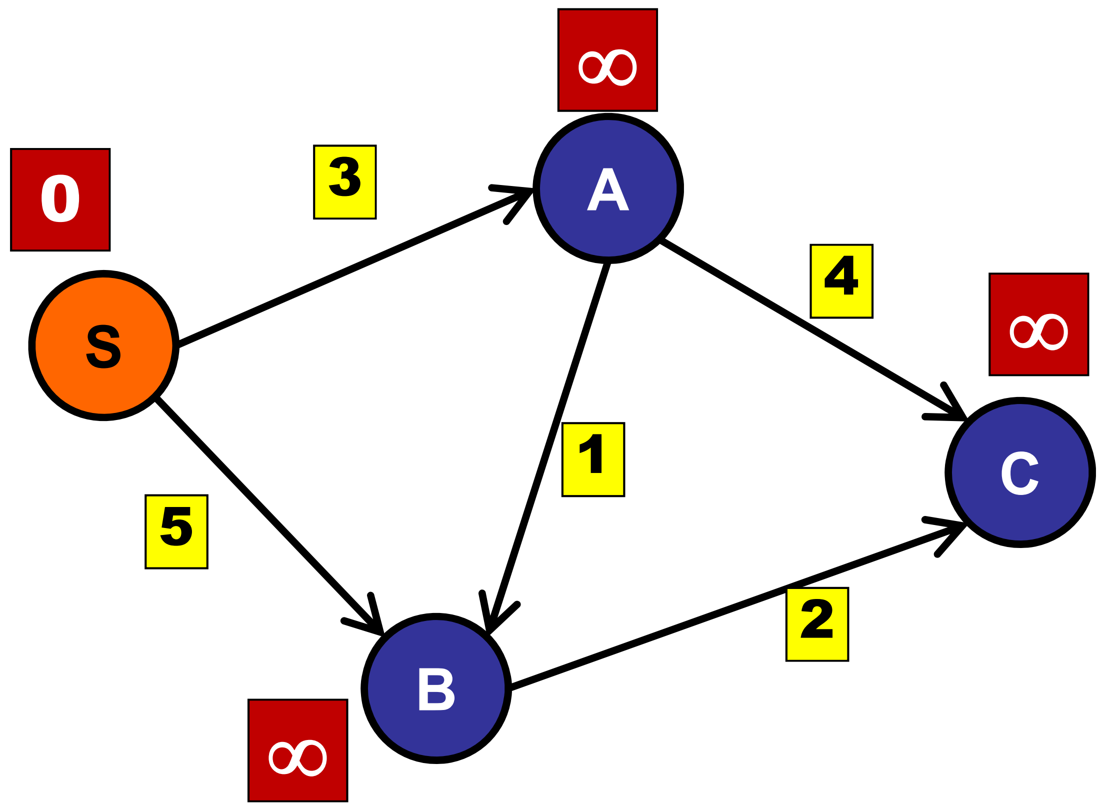
```
int[] dist = new int[V.length];
Arrays.fill(dist, INFTY);
dist[start] = 0;
```



# Shortest Paths

Maintain estimate for each distance:

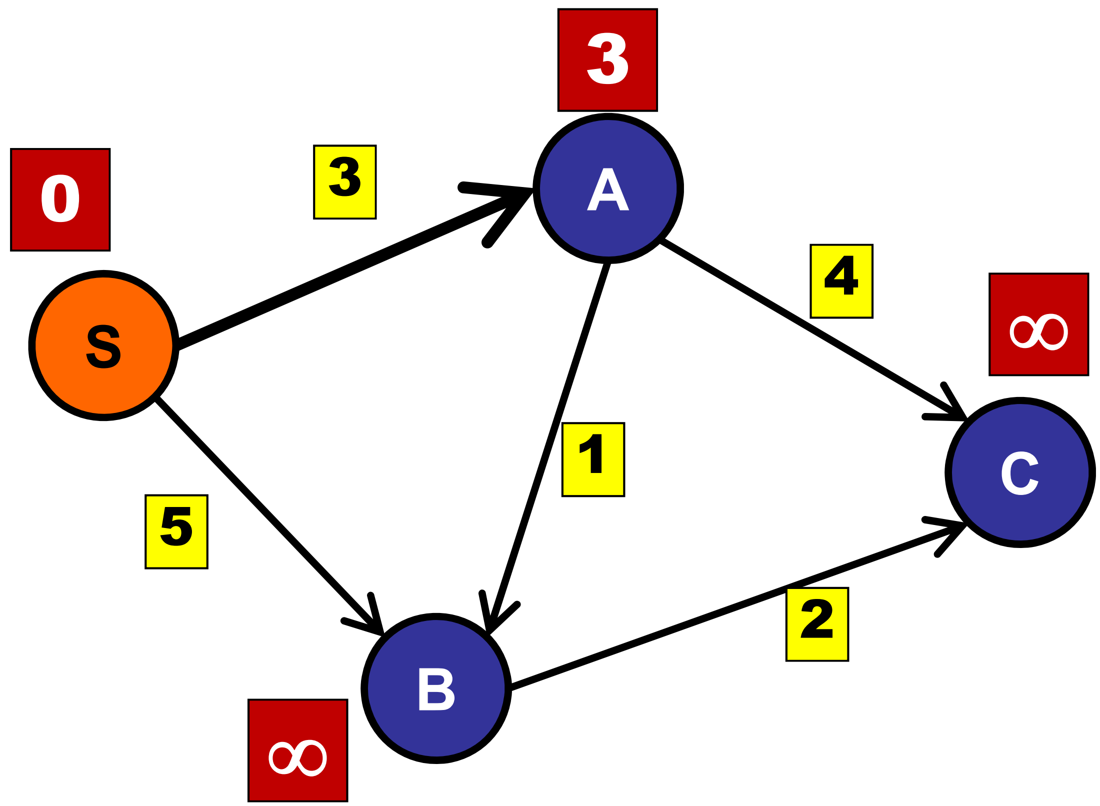
- Reduce estimate
- Invariant: estimate ≥ distance



# Shortest Paths

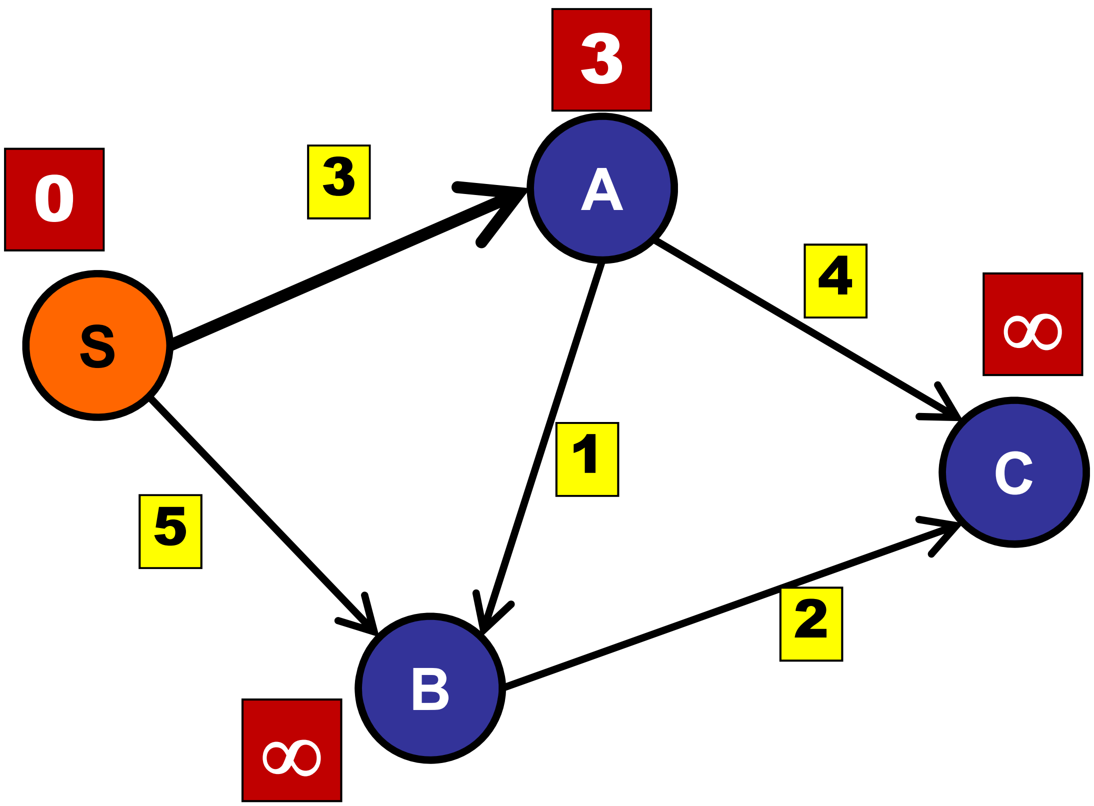
Maintain estimate for each distance:

relax(S, A)



# Shortest Paths

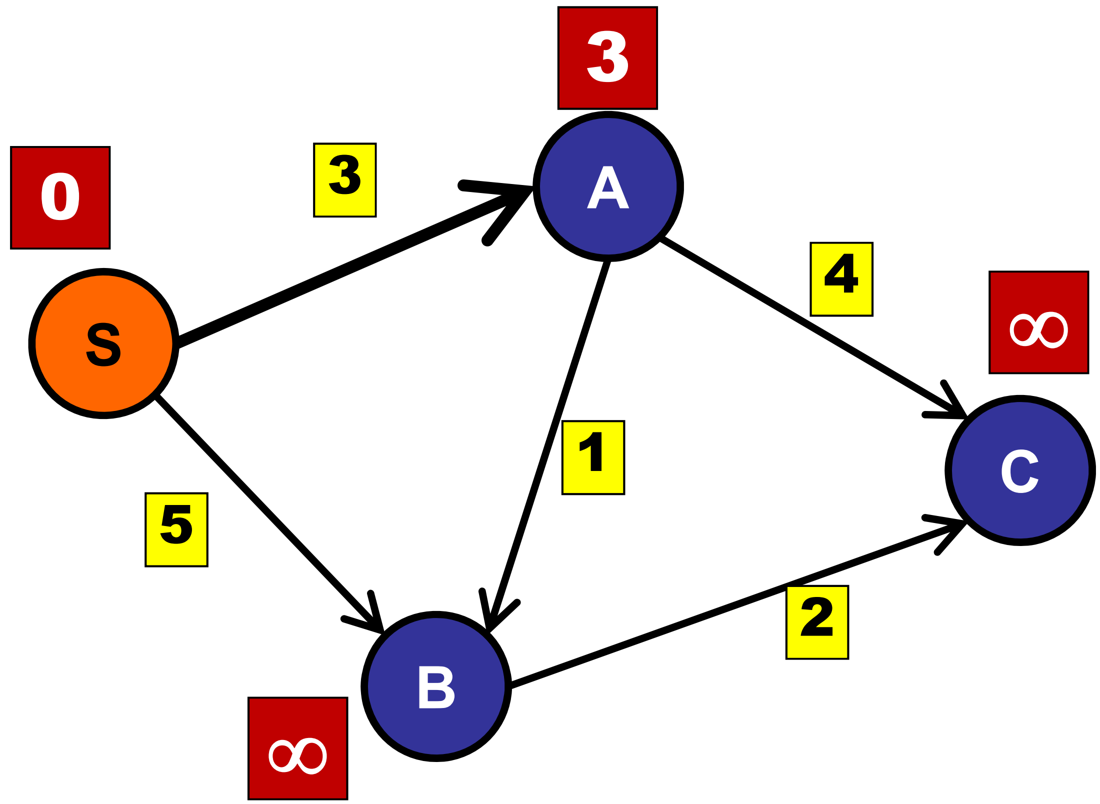
```
relax(int u, int v){
    if (dist[v] > dist[u] + weight(u,v))
        dist[v] = dist[u] + weight(u,v);
}
```



# Shortest Paths

Maintain estimate for each distance:

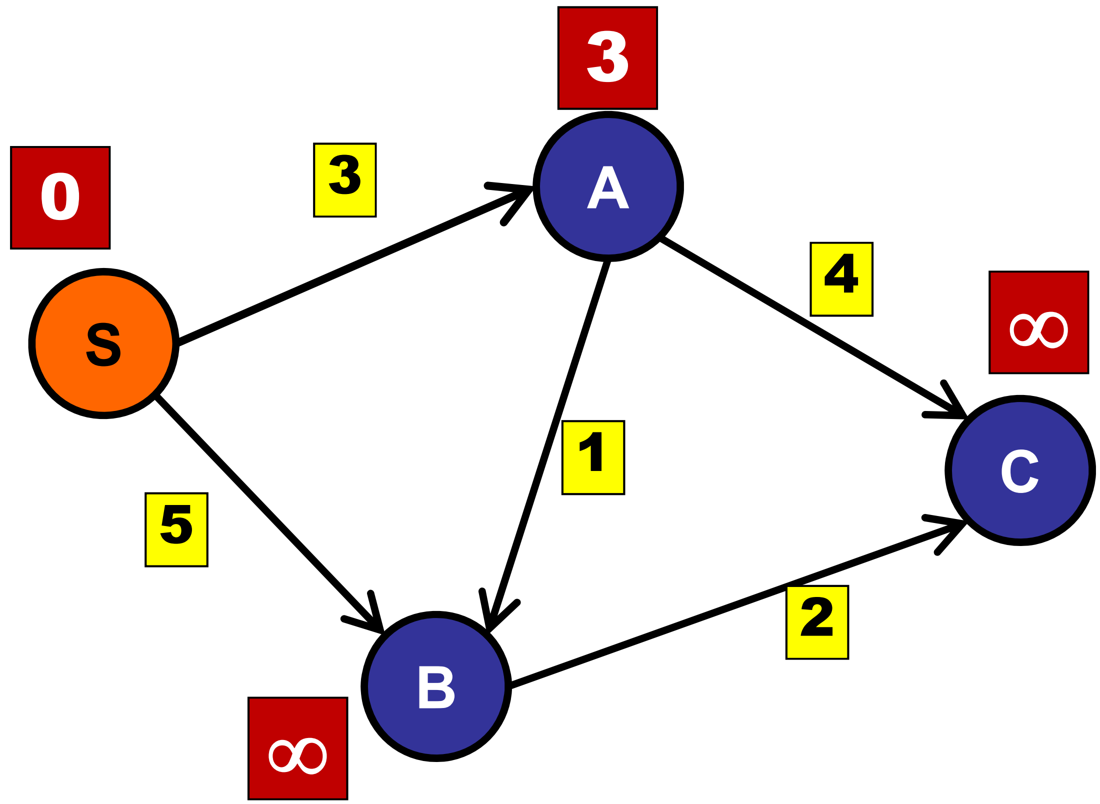
relax(S, A)



# Shortest Paths

Maintain estimate for each distance:

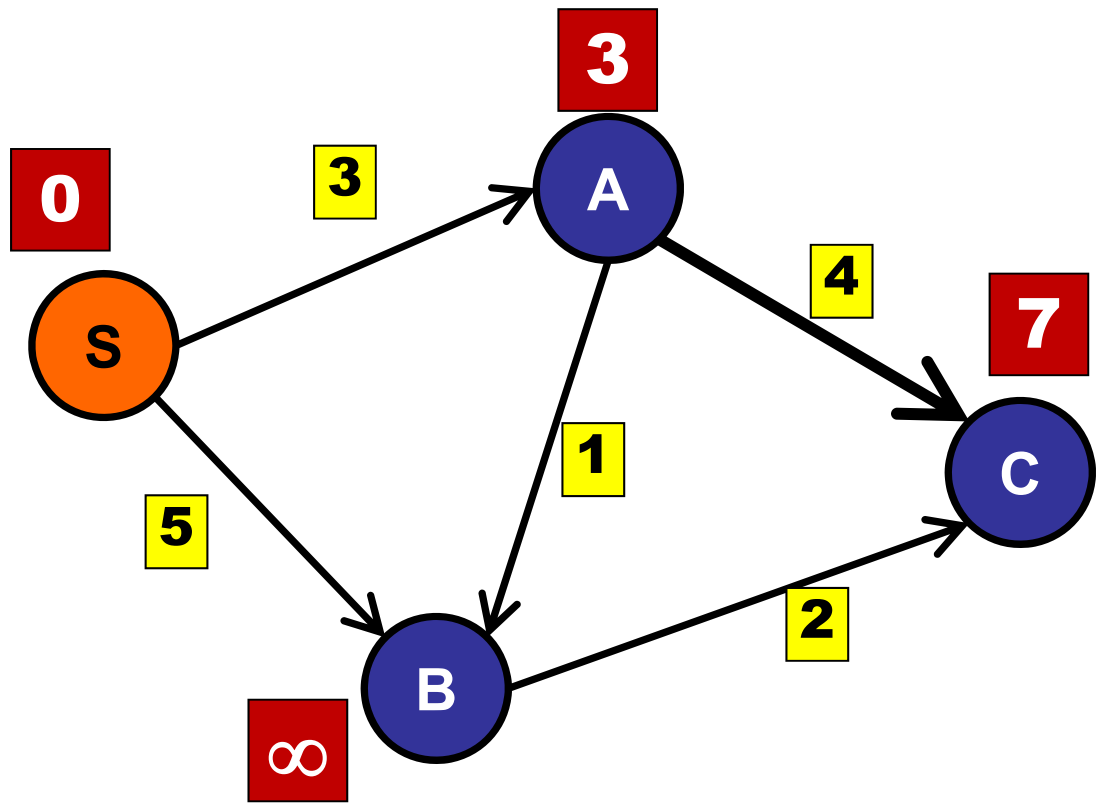
relax(A, C)



# Shortest Paths

Maintain estimate for each distance:

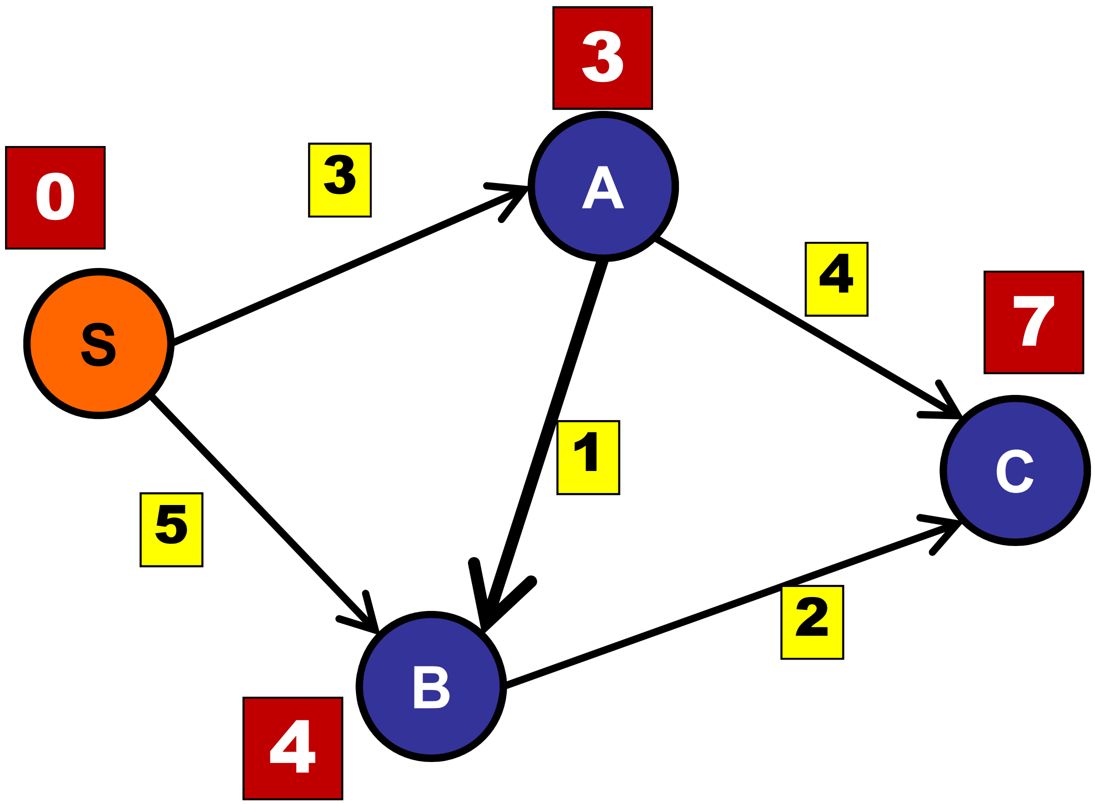
relax(A, C)



# Shortest Paths

Maintain estimate for each distance:

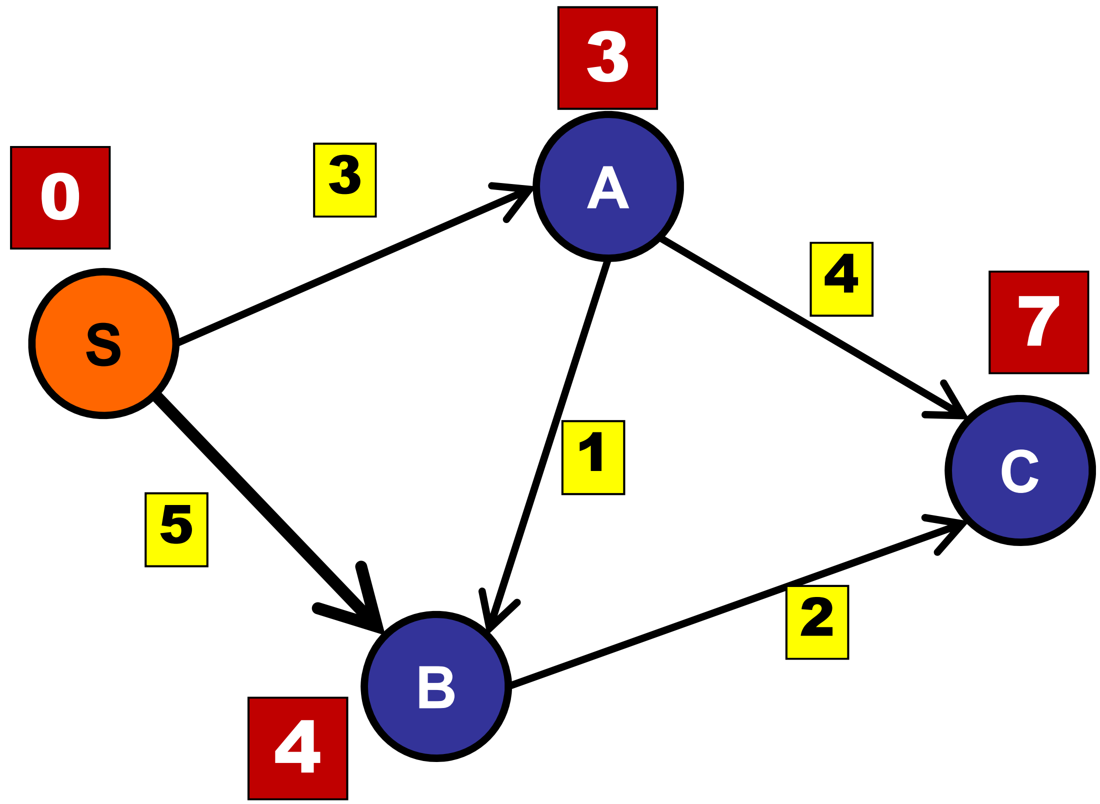
relax(A, B)



# Shortest Paths

Maintain estimate for each distance:

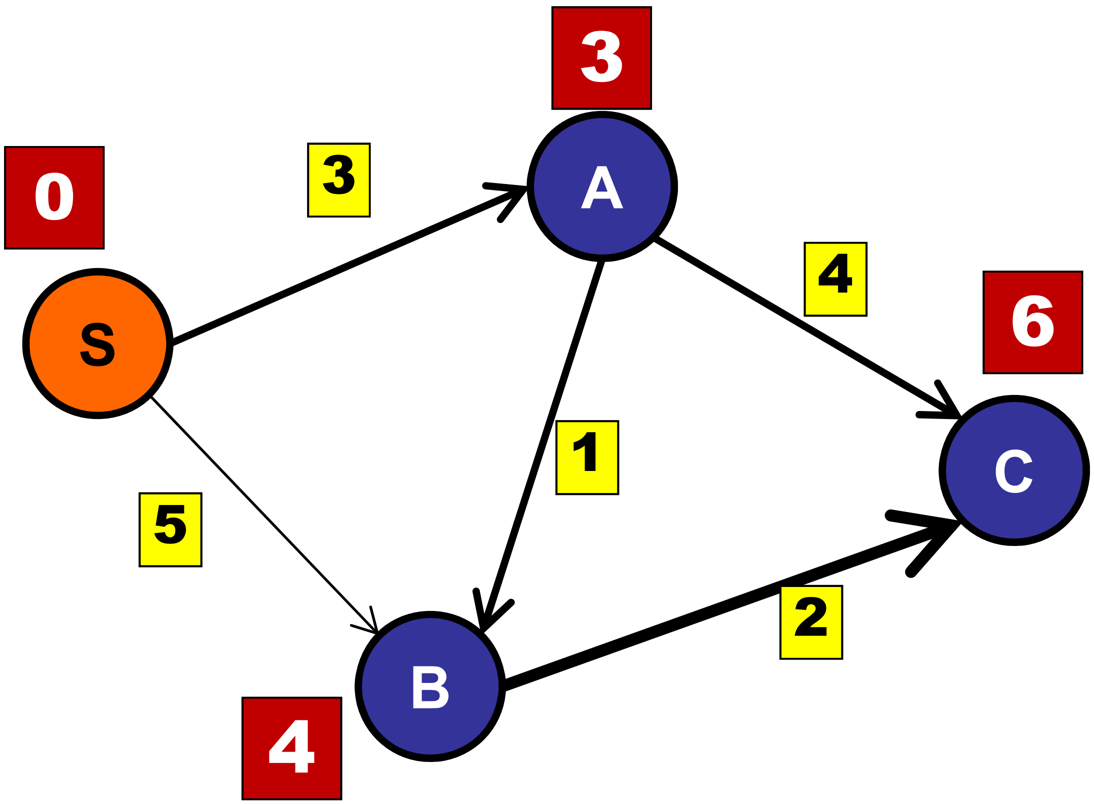
relax(S, B)



# Shortest Paths

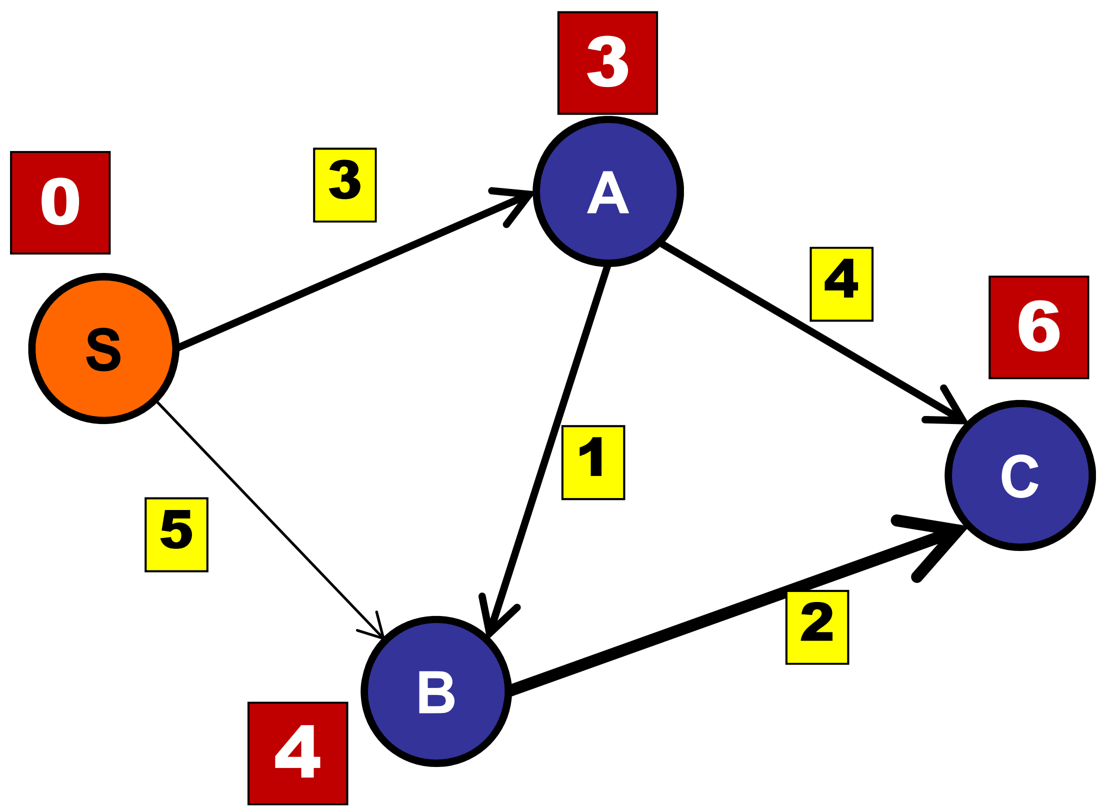
Maintain estimate for each distance:

relax(B, C)



# Shortest Paths

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

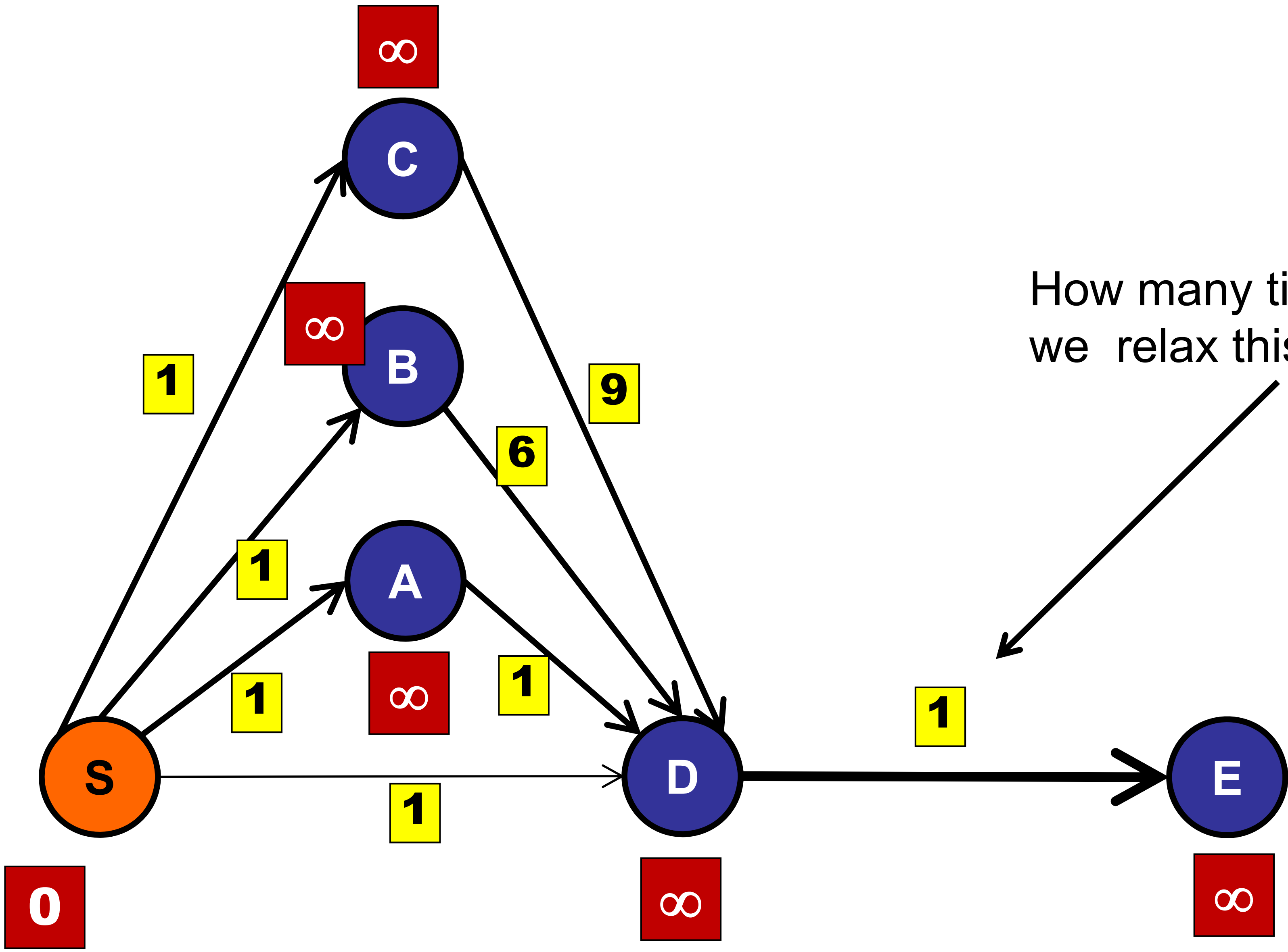


Does this algorithm work:

**for every edge e: relax(e)**

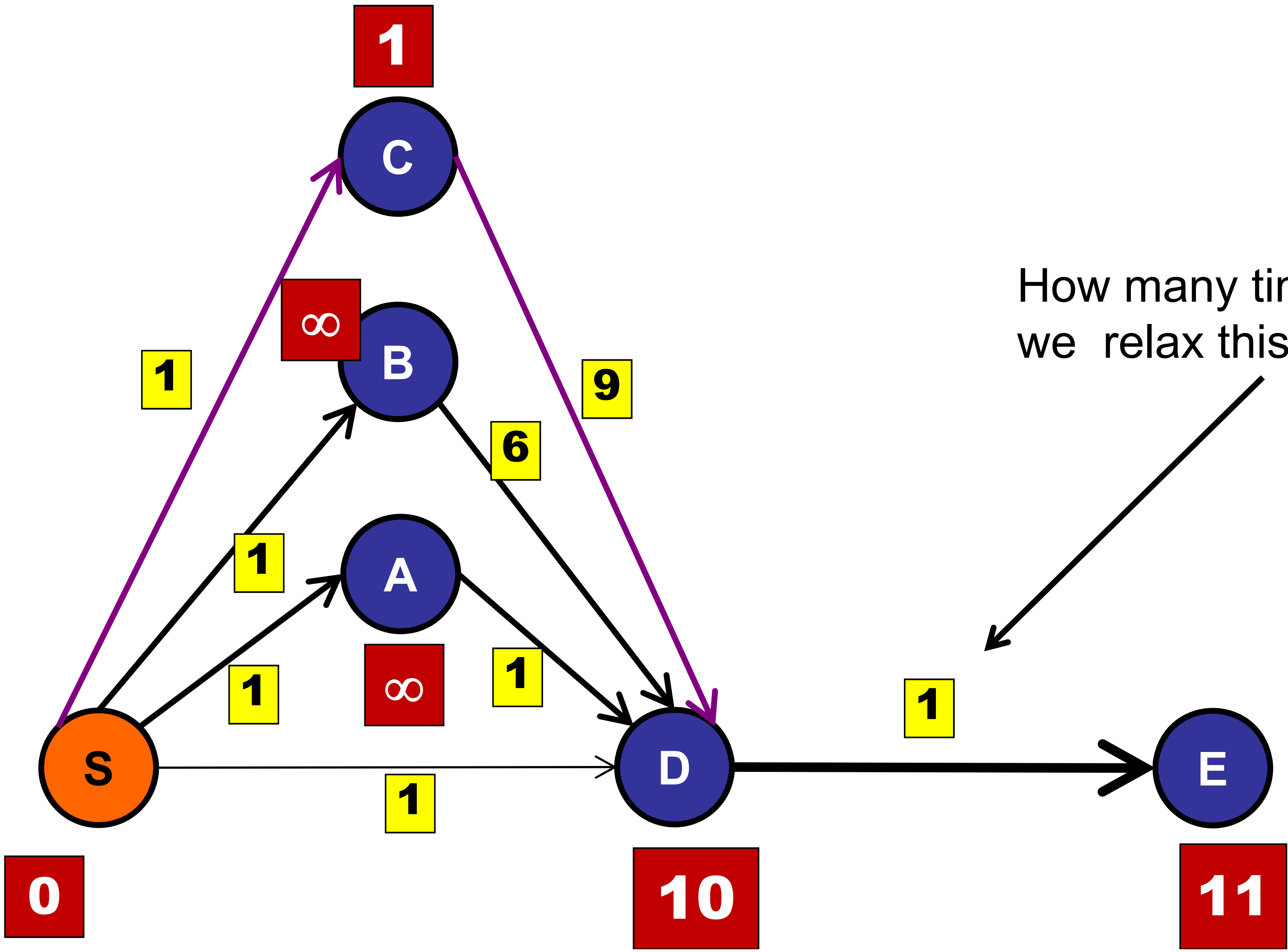
1. Yes
2. No
3. Only on trees.

# Shortest Paths



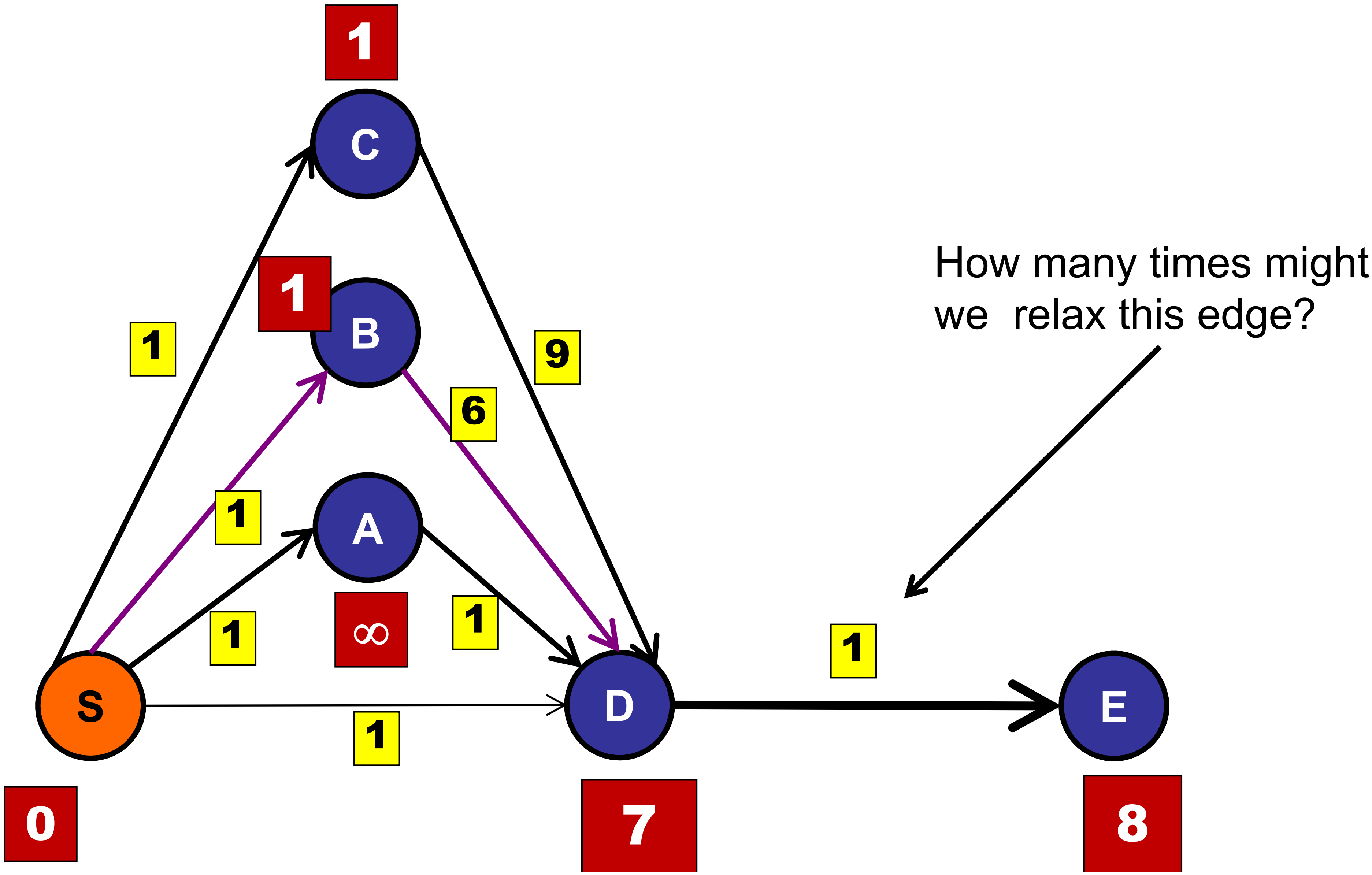

How many times might we relax this edge?

# Shortest Paths

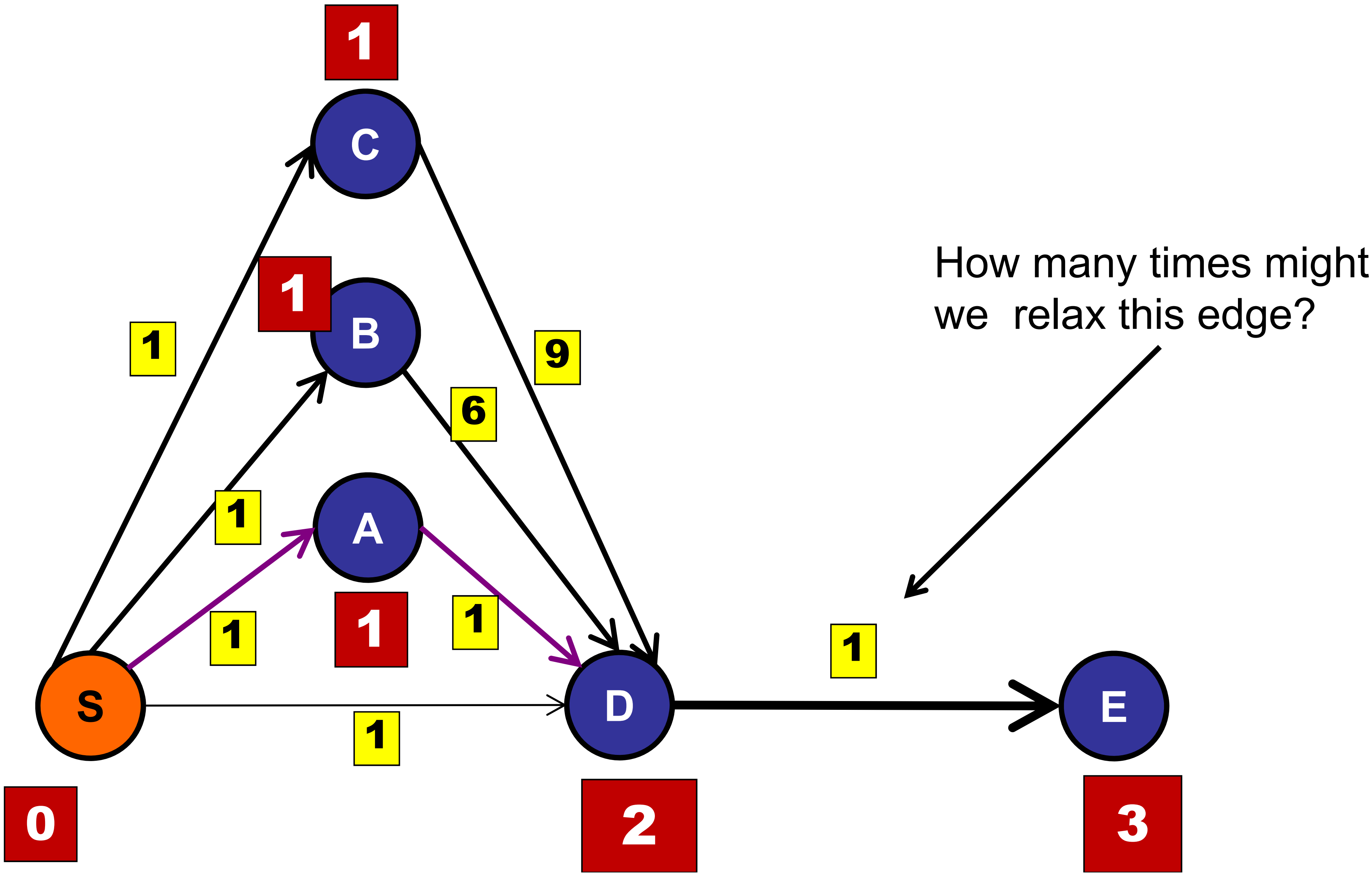



How many times might we relax this edge?

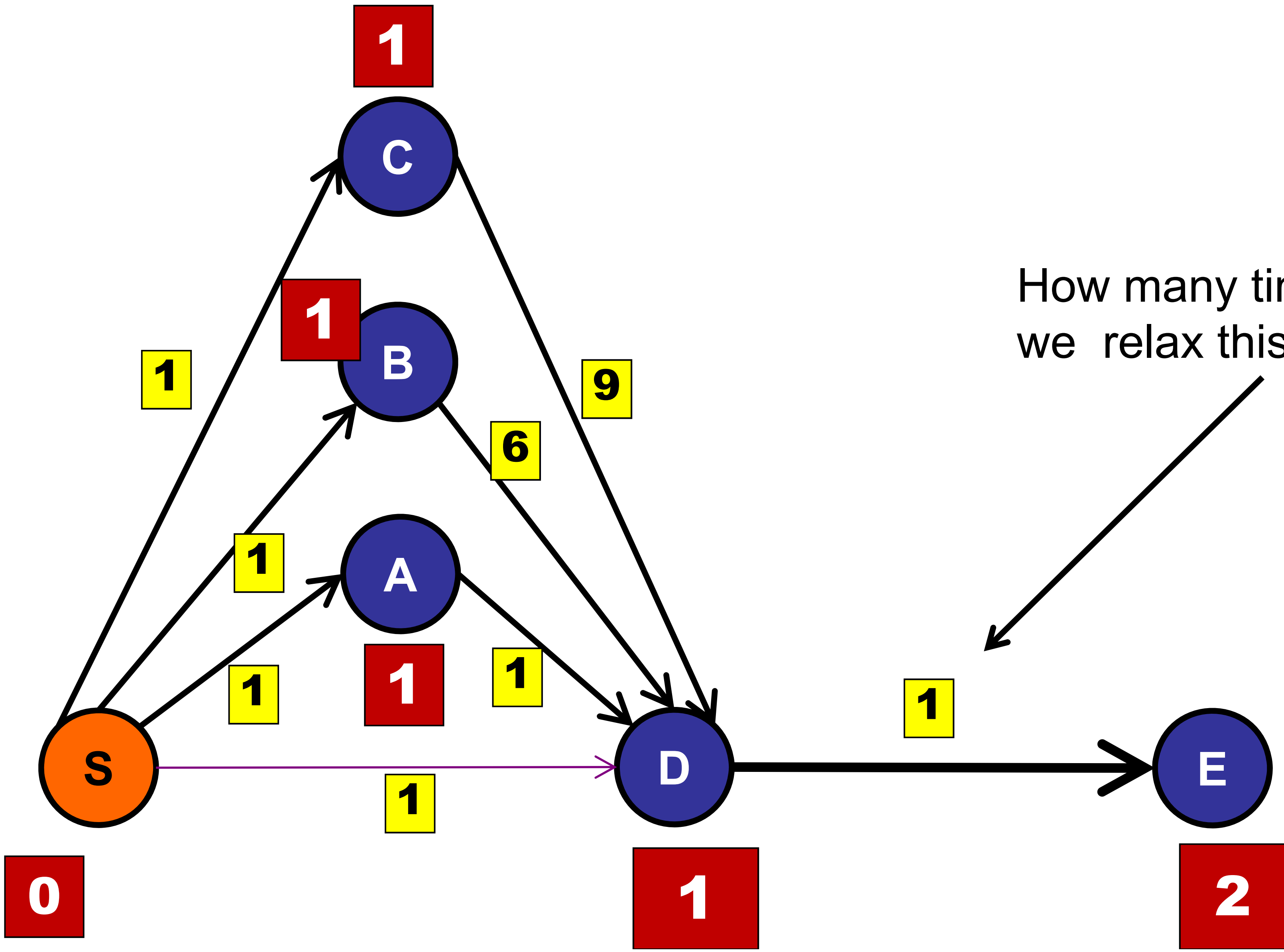
# Shortest Paths



# Shortest Paths



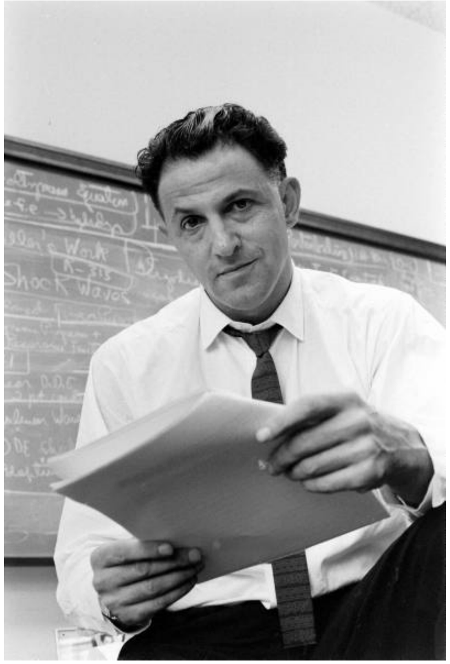
# Shortest Paths



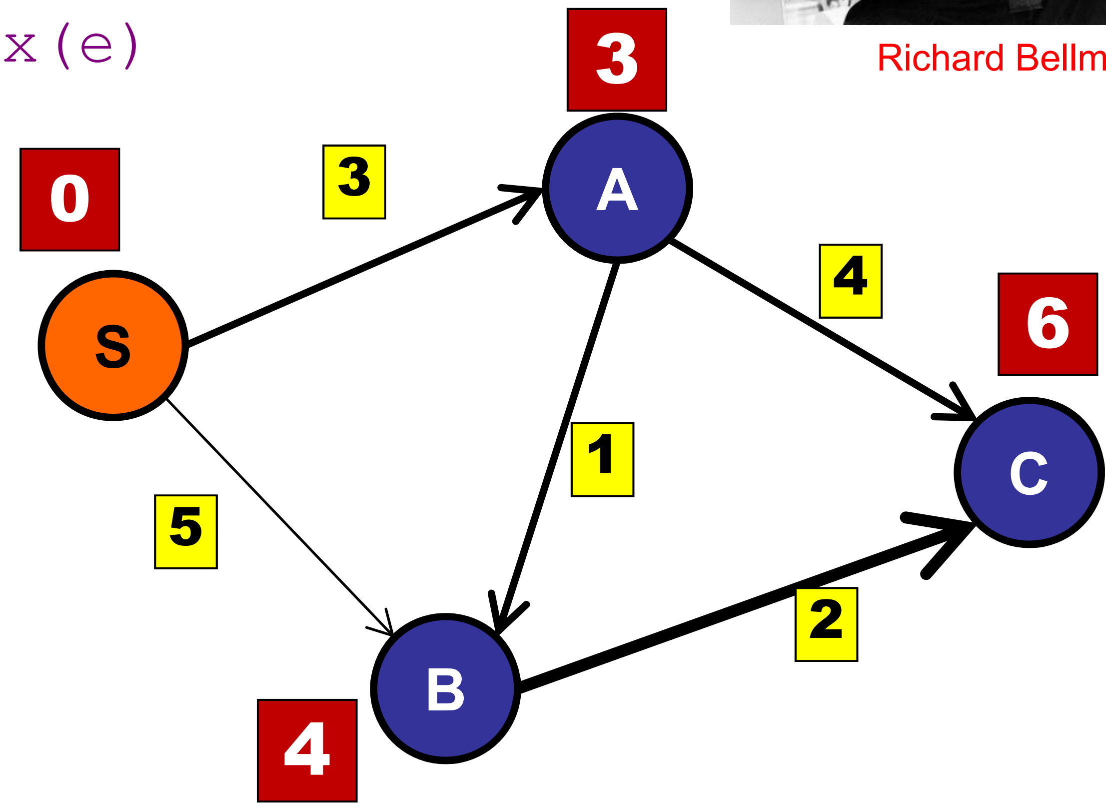

How many times might we relax this edge?

# Bellman-Ford

```
n = V.length;
for (i=0; i<n; i++)
    for (Edge e : graph)
        relax(e)
```



Richard Bellman



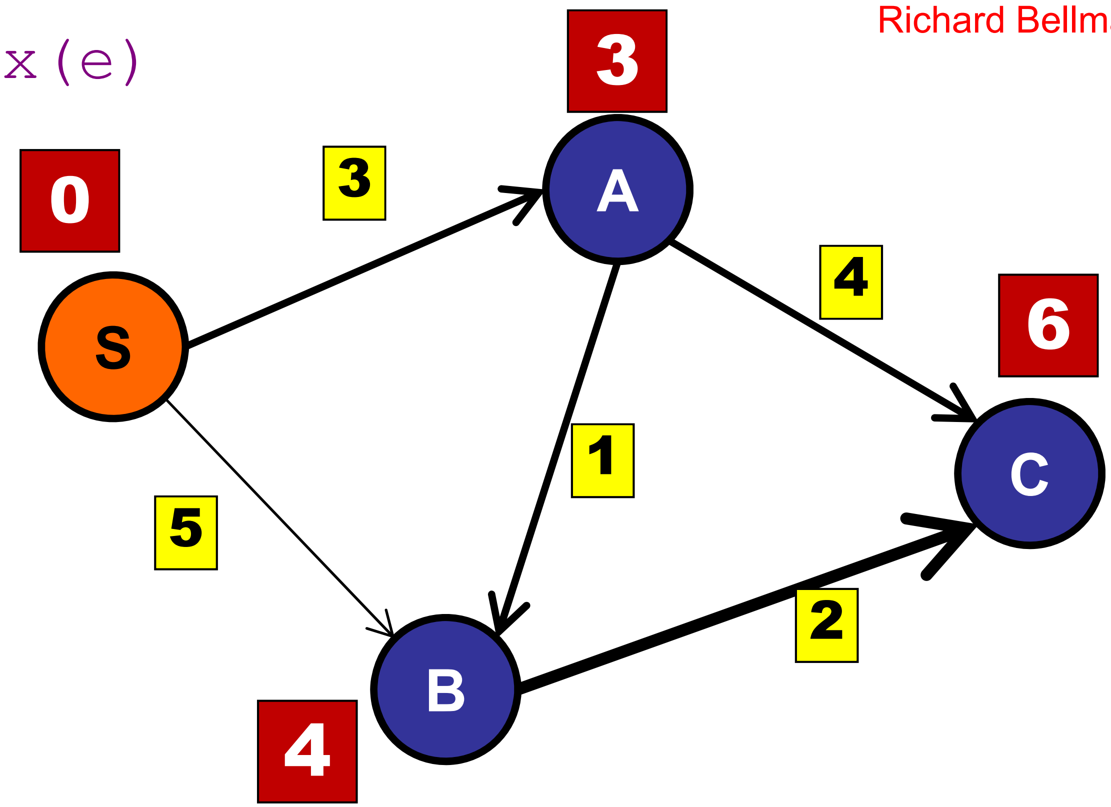
When can you terminate early?

1. When a relax operation has no effect.
2. When two consecutive relax operations have no effect.
3. When an entire sequence of |E| relax operations have no effect.
4. Never. Only after |V| complete iterations.

# Bellman-Ford

```
n = V.length;
for (i=0; i<n; i++)
    for (Edge e : graph)
        relax(e)
```

Richard Bellman

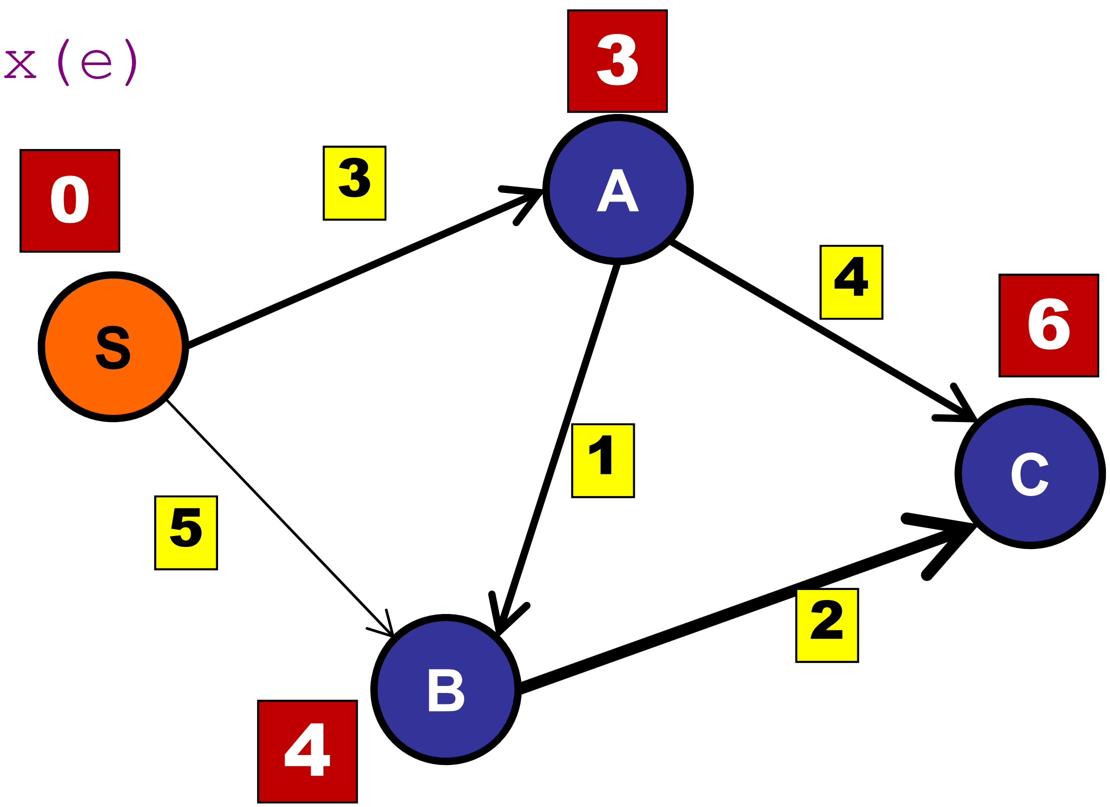


## What is the running time of Bellman-Ford?

1. O(V)
2. O(E)
3. O(V+E)
4. O(E log V)
5. ✓ O(EV)

# Bellman-Ford

```
n = V.length;
for (i=0; i<n; i++)
    for (Edge e : graph)
        relax(e)
```

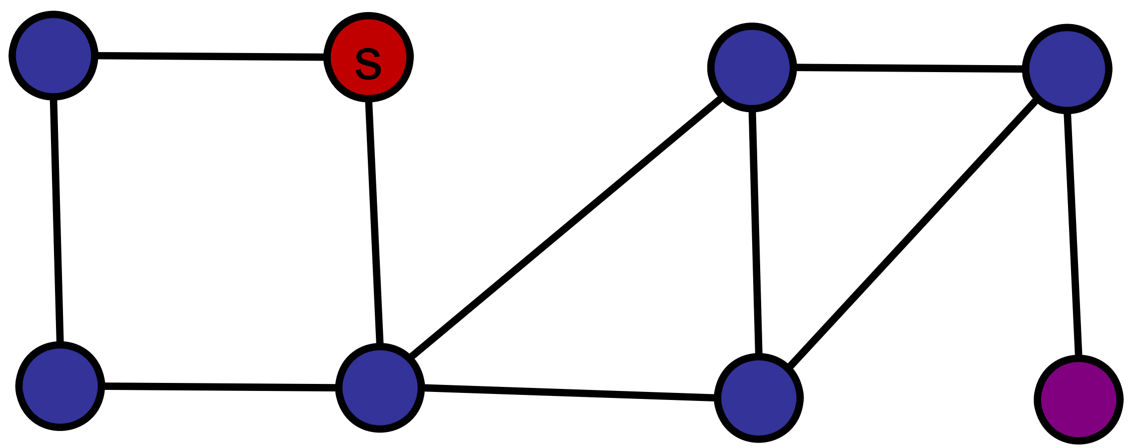


# Bellman-Ford

Why does this work?

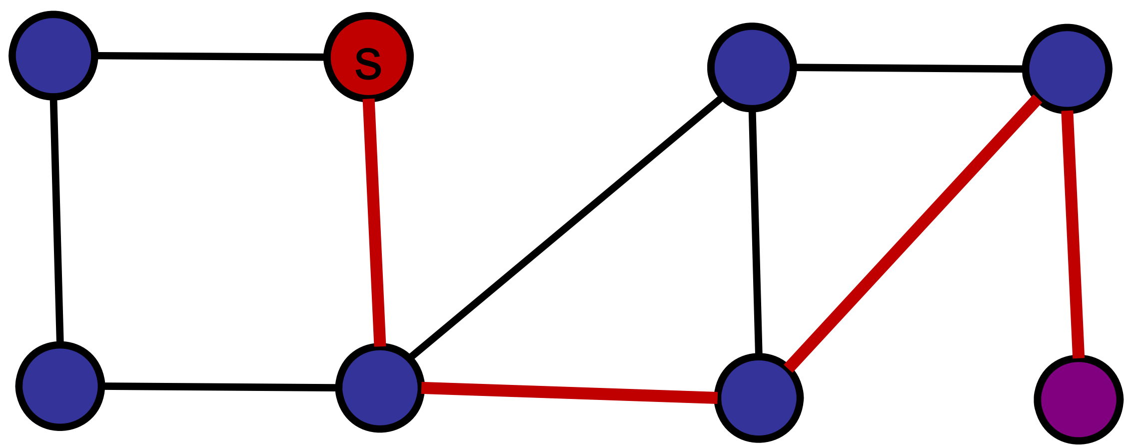
# Bellman-Ford

Why does this work?



# Bellman-Ford

Why does this work?

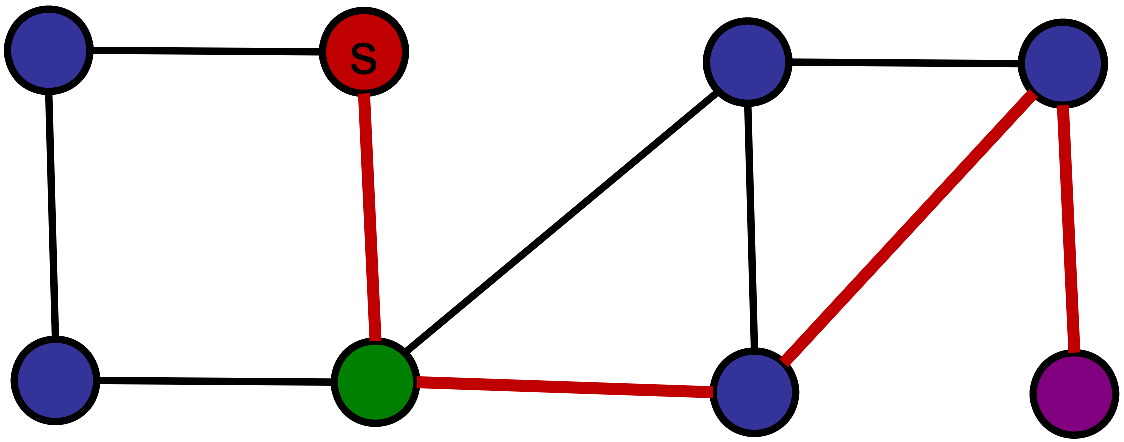



Look at minimum weight path from S to D.

(Path is simple: no loops.)

# Bellman-Ford

Why does this work?

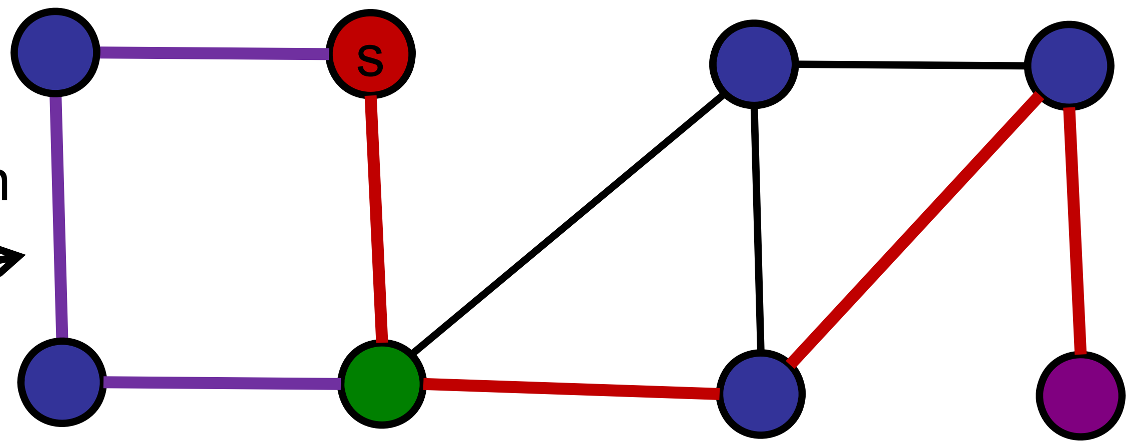



After 1 iteration, 1 hop estimate is correct.

# Bellman-Ford

## Why does this work?

What if this path is shorter?

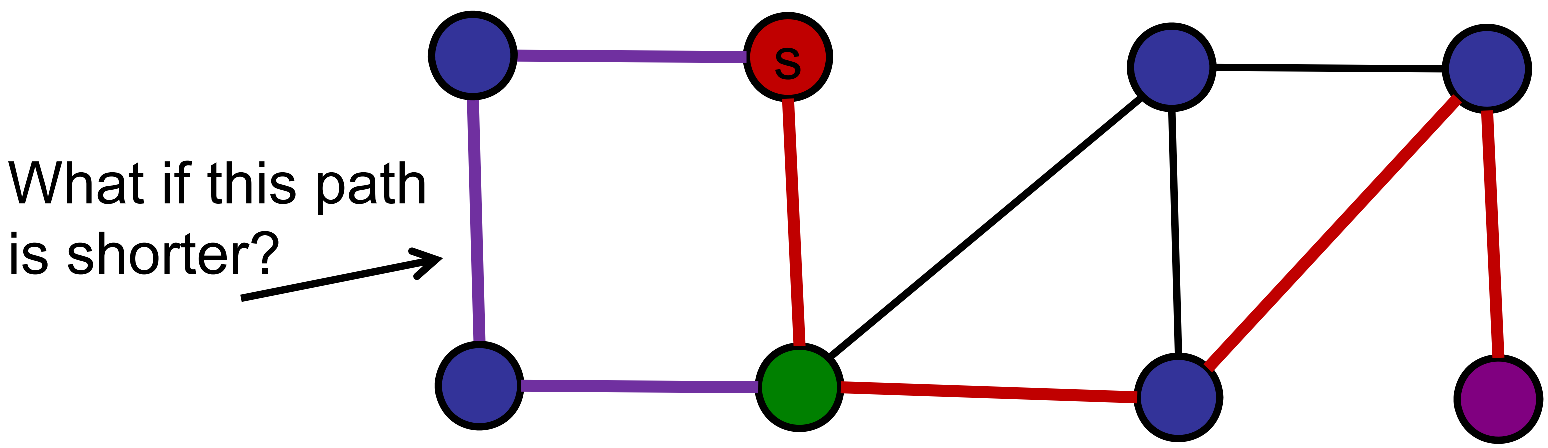



After 1 iteration, 1 hop estimate is correct.

# Shortest Paths

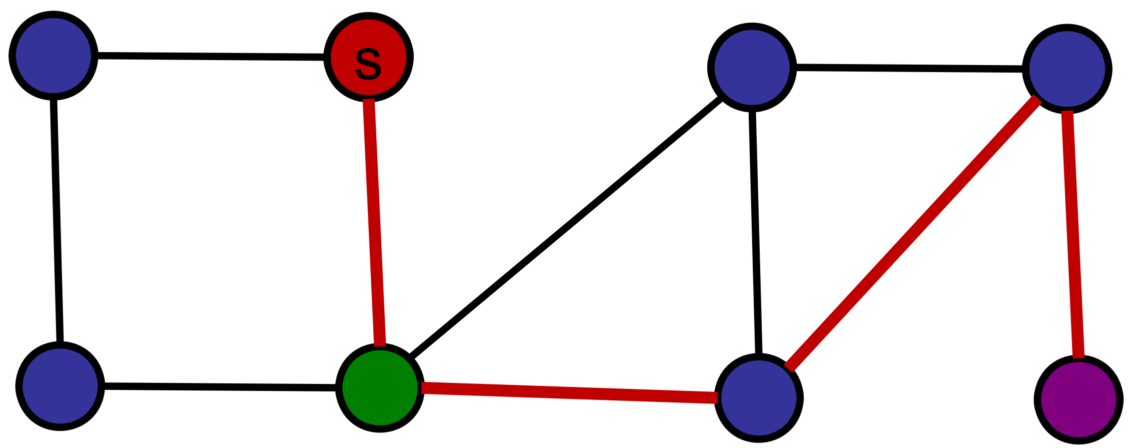
Key property:

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).



# Bellman-Ford

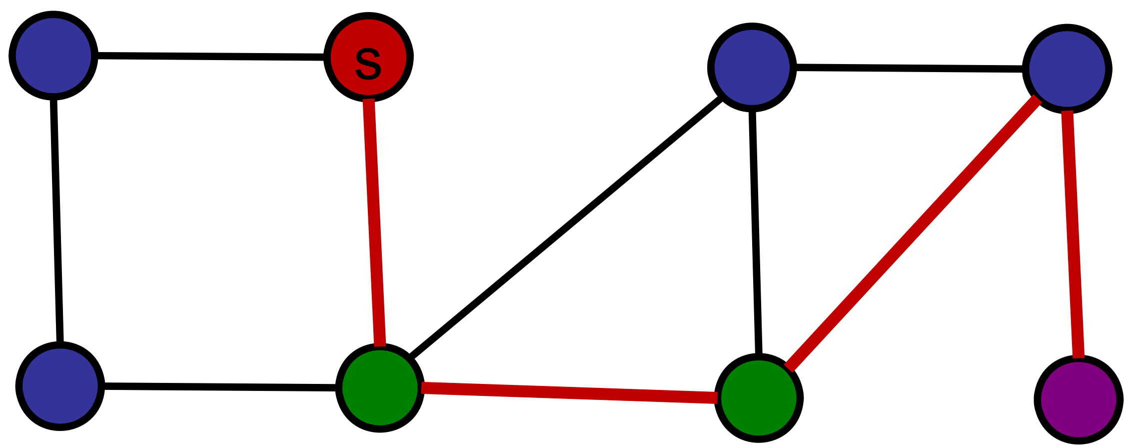
Why does this work?




After 1 iteration, 1 hop estimate is correct.

# Bellman-Ford

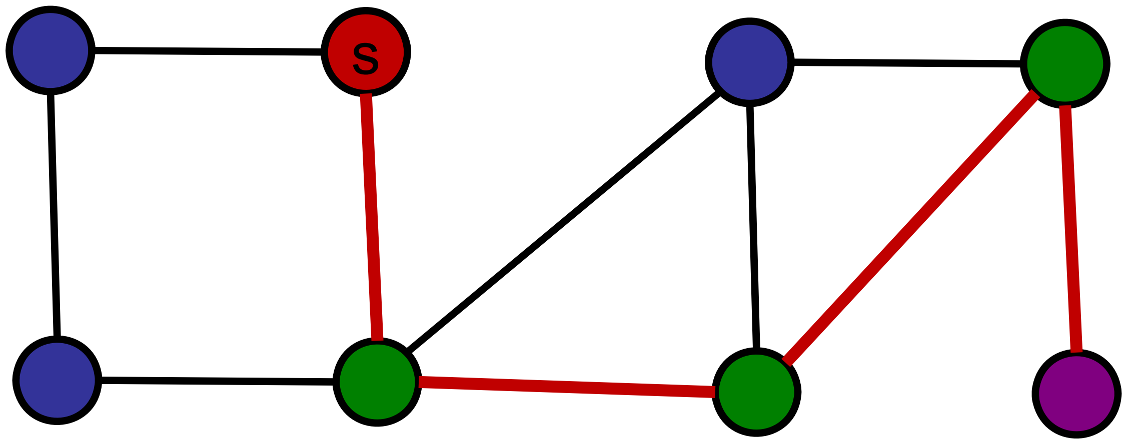
Why does this work?




After 2 iterations, 2 hop estimate is correct.

# Bellman-Ford

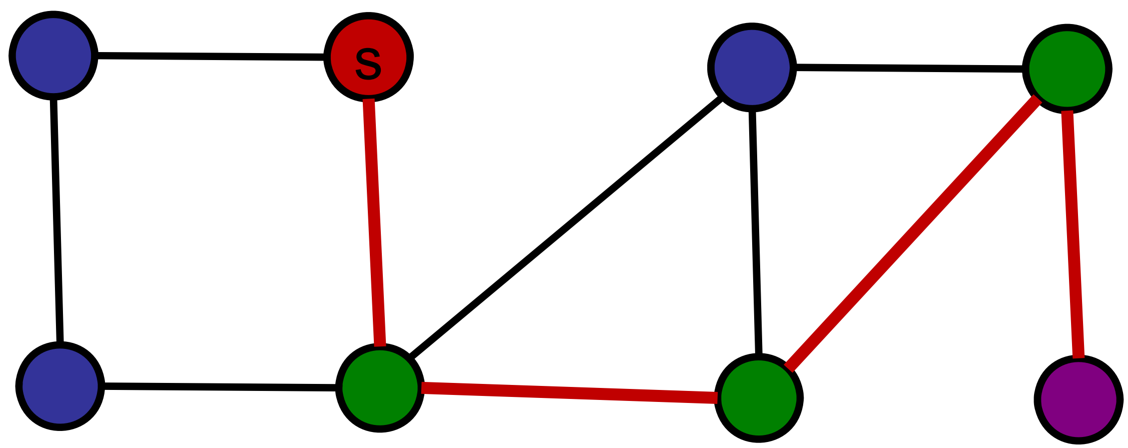
Why does this work?




After 3 iterations, 3 hop estimate is correct.

# Bellman-Ford

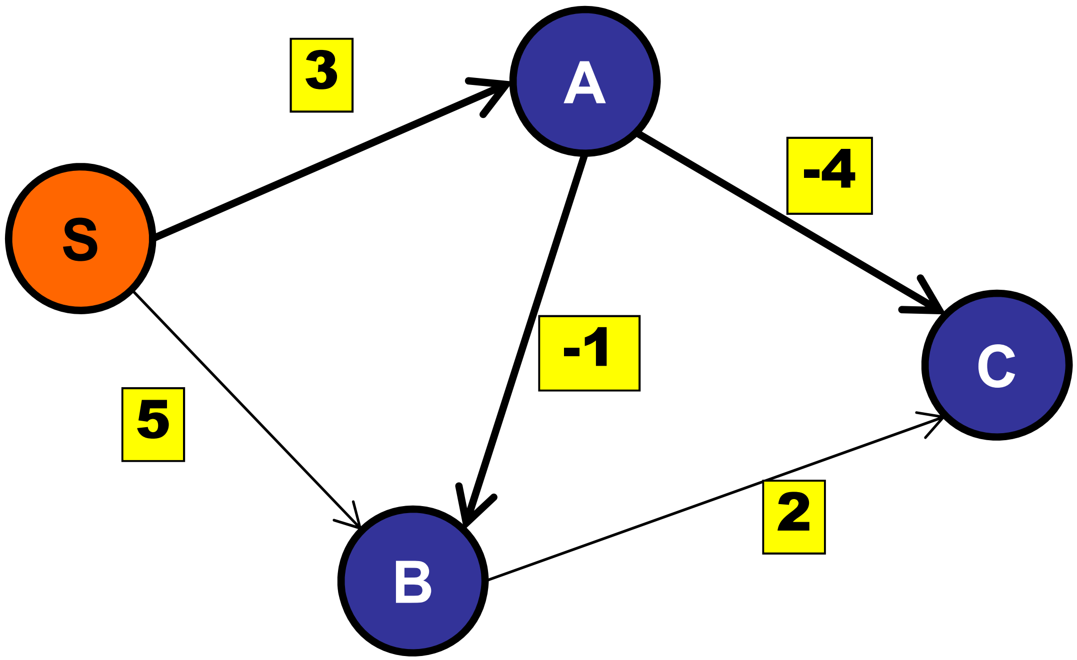
Why does this work?




After 4 iterations, D estimate is correct.

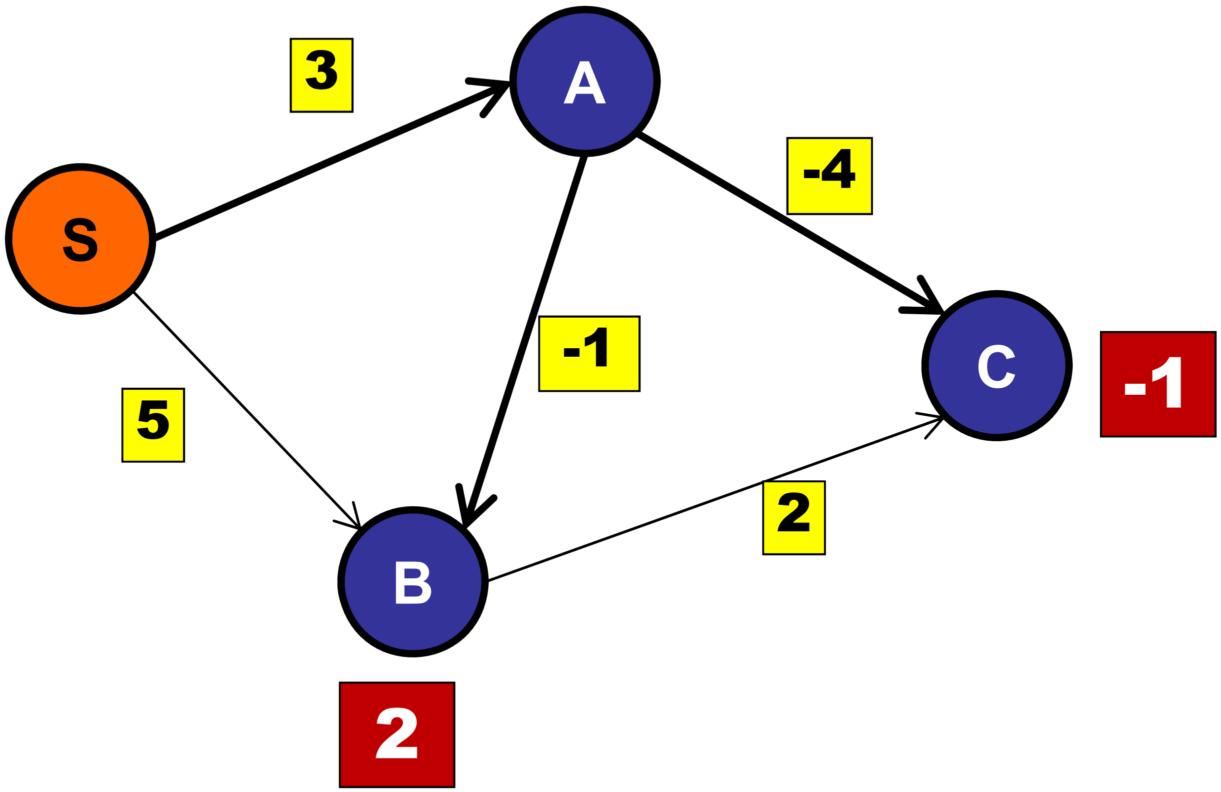
# Bellman-Ford

What if edges have negative weight?



# Bellman-Ford

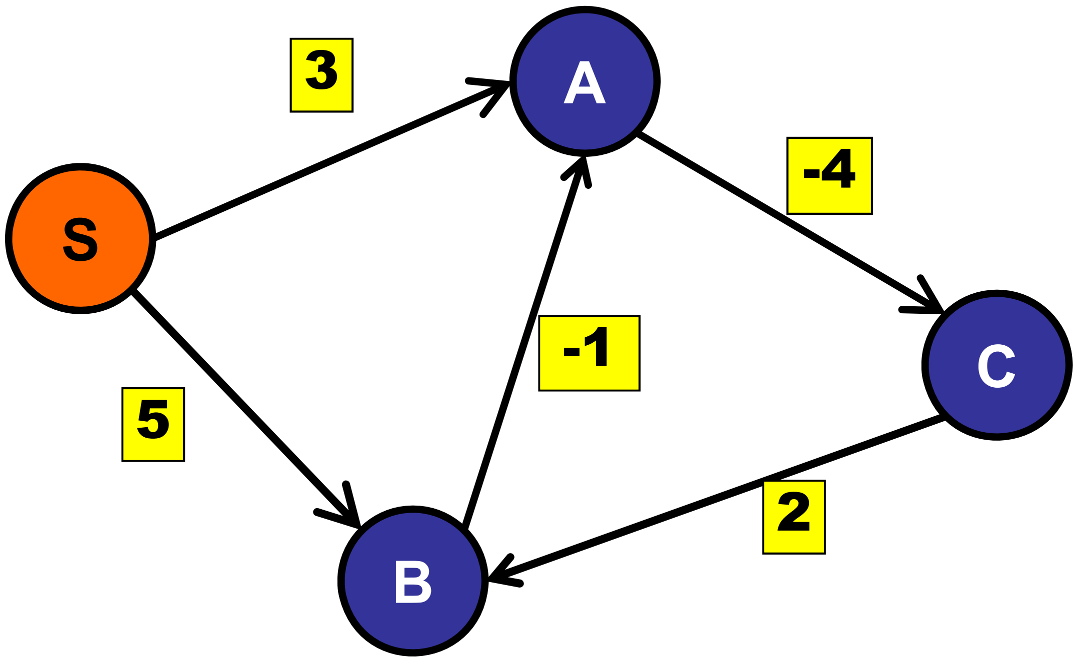
What if edges have negative weight?




No problem!

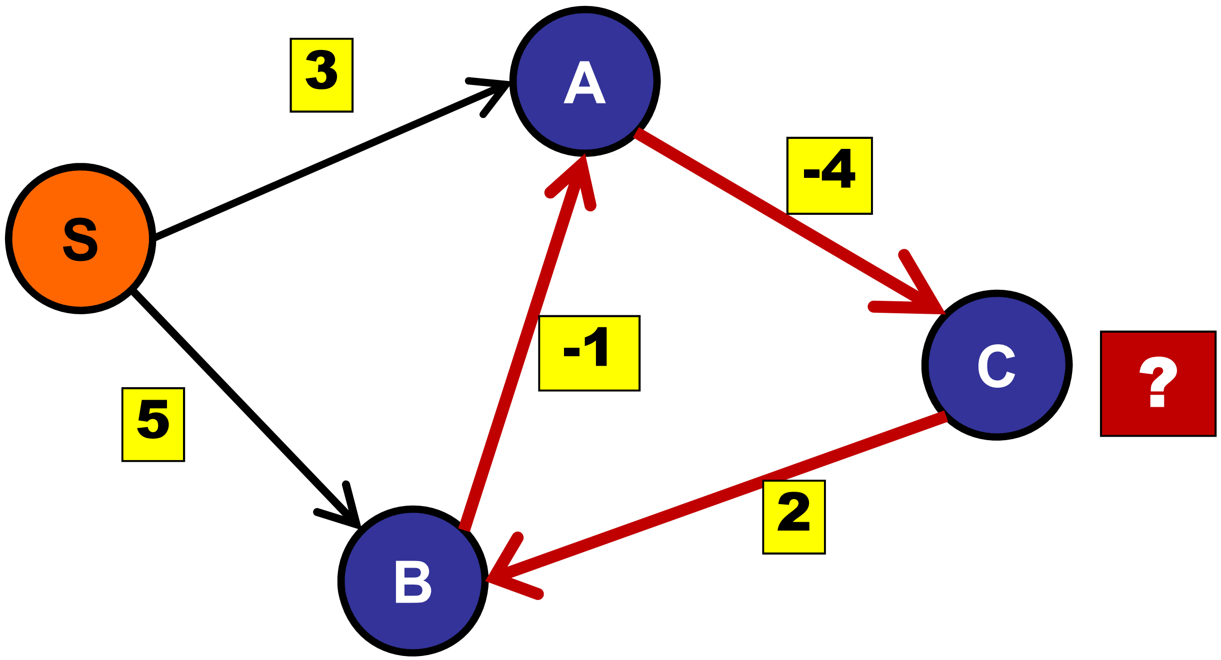
# Bellman-Ford

What if edges have negative weight?



# Bellman-Ford

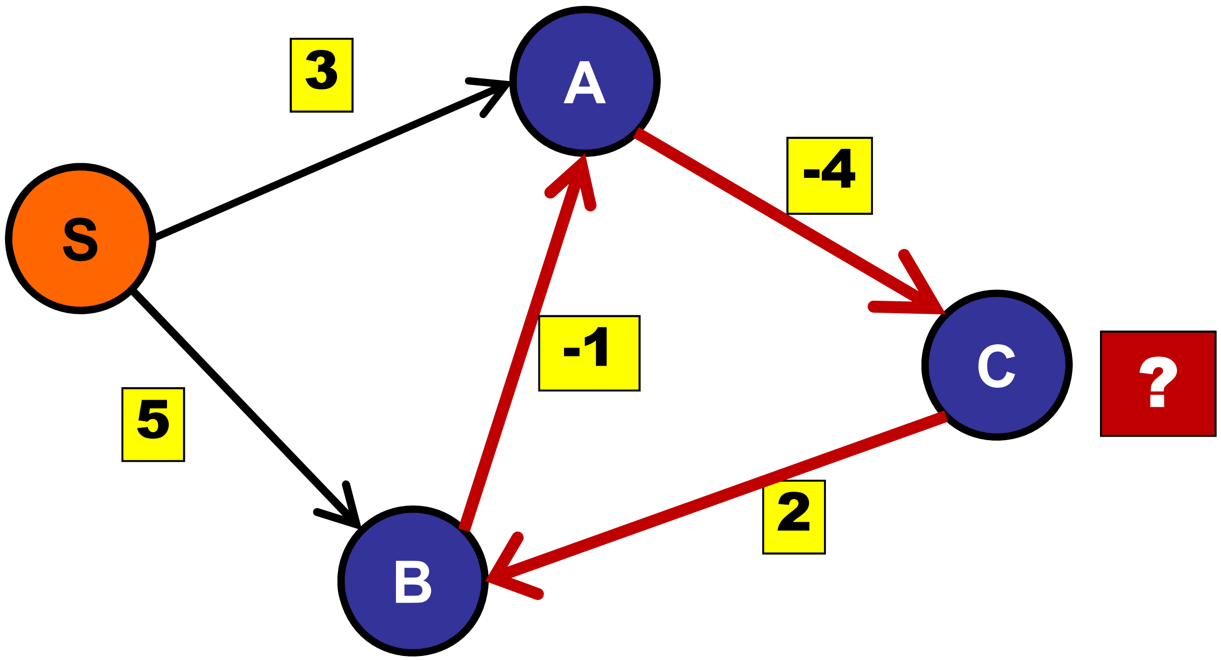
What if edges have negative weight?




d(S,C) is infinitely negative!

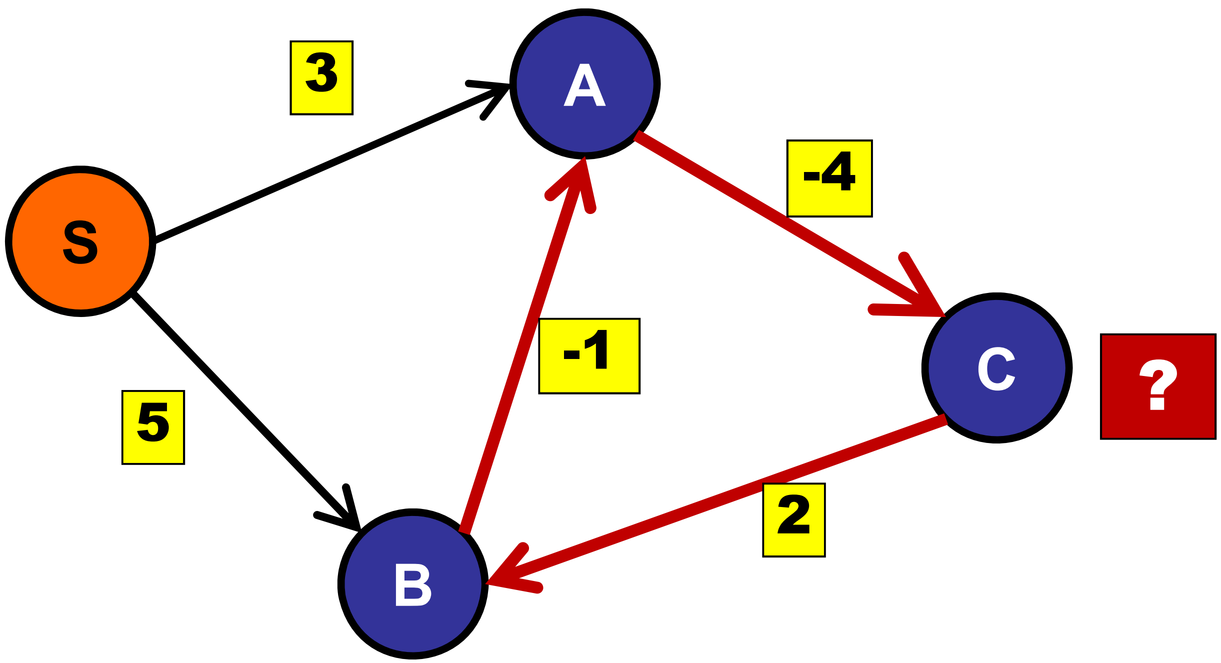
# Negative weight cycles

How to detect negative weight cycles?



# Negative weight cycles

How to detect negative weight cycles?

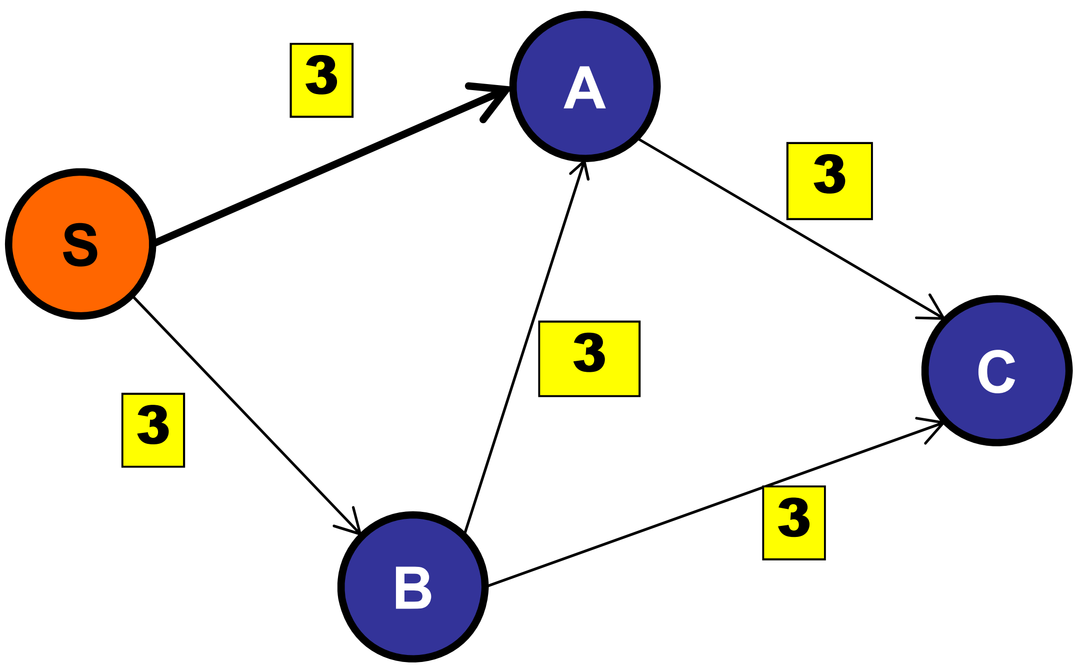



Run Bellman-Ford for |V|+1 iterations.

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

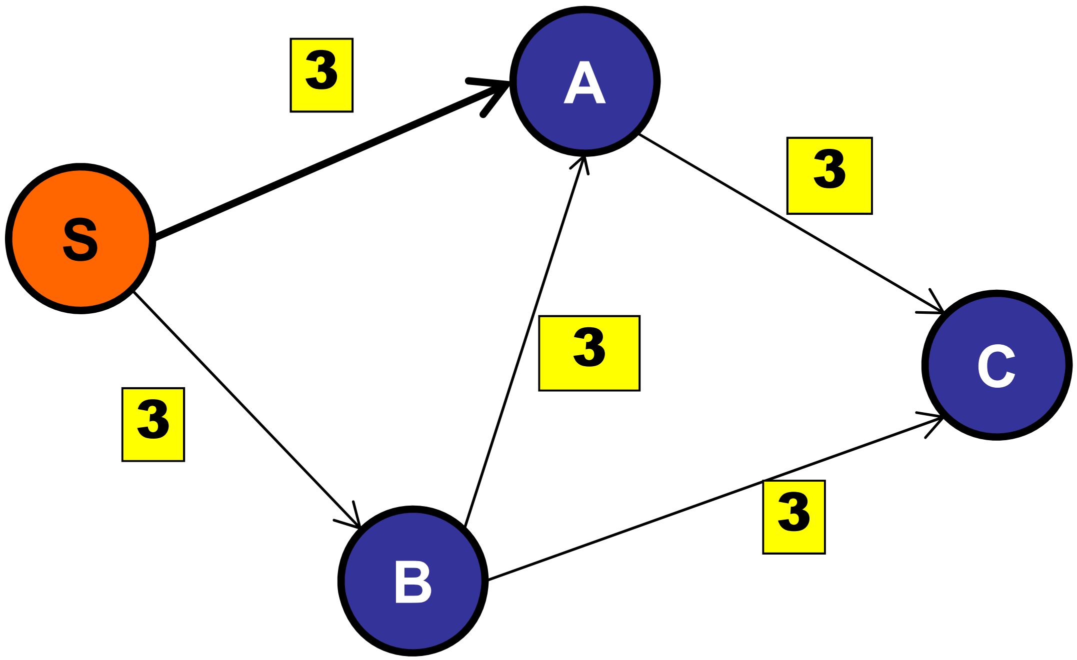
# Bellman-Ford

Special case: all edges have the same weight



# Bellman-Ford

Special case: all edges have the same weight.




Use regular Breadth-First Search.

# Bellman-Ford Summary

Basic idea:

- Repeat |V| times: relax every edge
- Stop when "converges".
- O(VE) time.

Special issues:

- If negative weight-cycle: impossible.
- Use Bellman-Ford to detect negative weight cycle.
- If all weights are the same, use BFS.

# Faster algorithms?

Key idea:

Relax the edges in the "right" order.

Only relax each edge once:

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

Necessary assumption:

All edges weights >= 0.

Extending a path does not make it shorter!

# Edsger W. Dijkstra

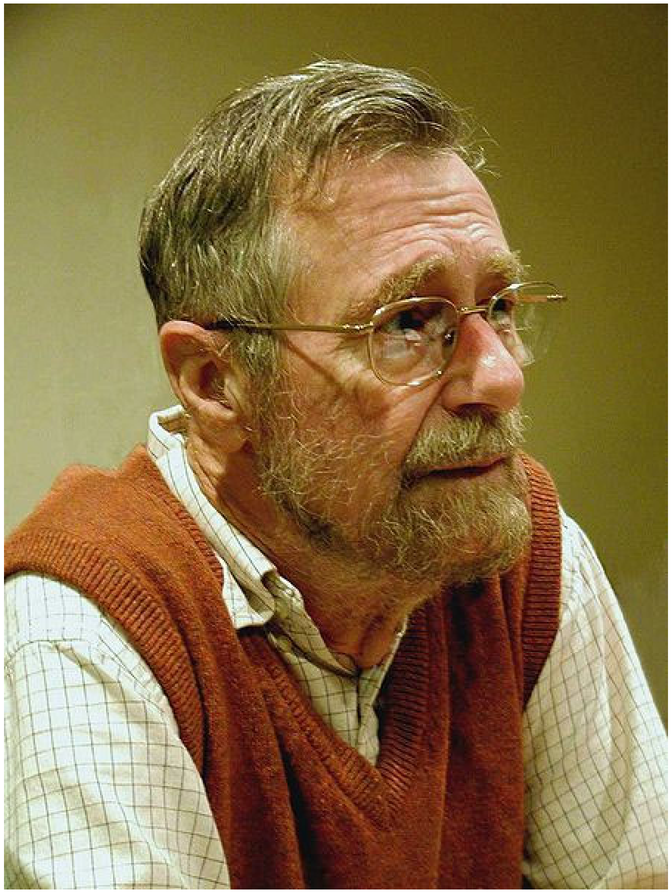
*"Computer science is no more about computers than astronomy is about telescopes."*

*"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."*



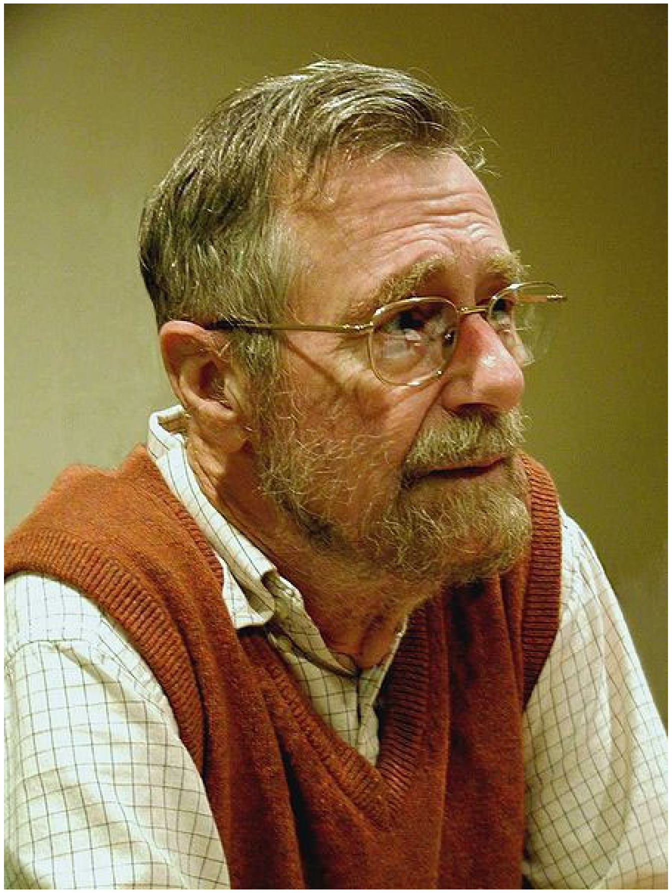
# Edsger W. Dijkstra

*"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."*

*"APL is a mistake, carried through to perfection. It is the language of the future for the programming techniques of the past: it creates a new generation of coding bums."*

*"Object-oriented programming is an exceptionally bad idea which could only have originated in California."*



# Faster algorithms?

Key idea:

Relax the edges in the “right” order.

Only relax each edge once:

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

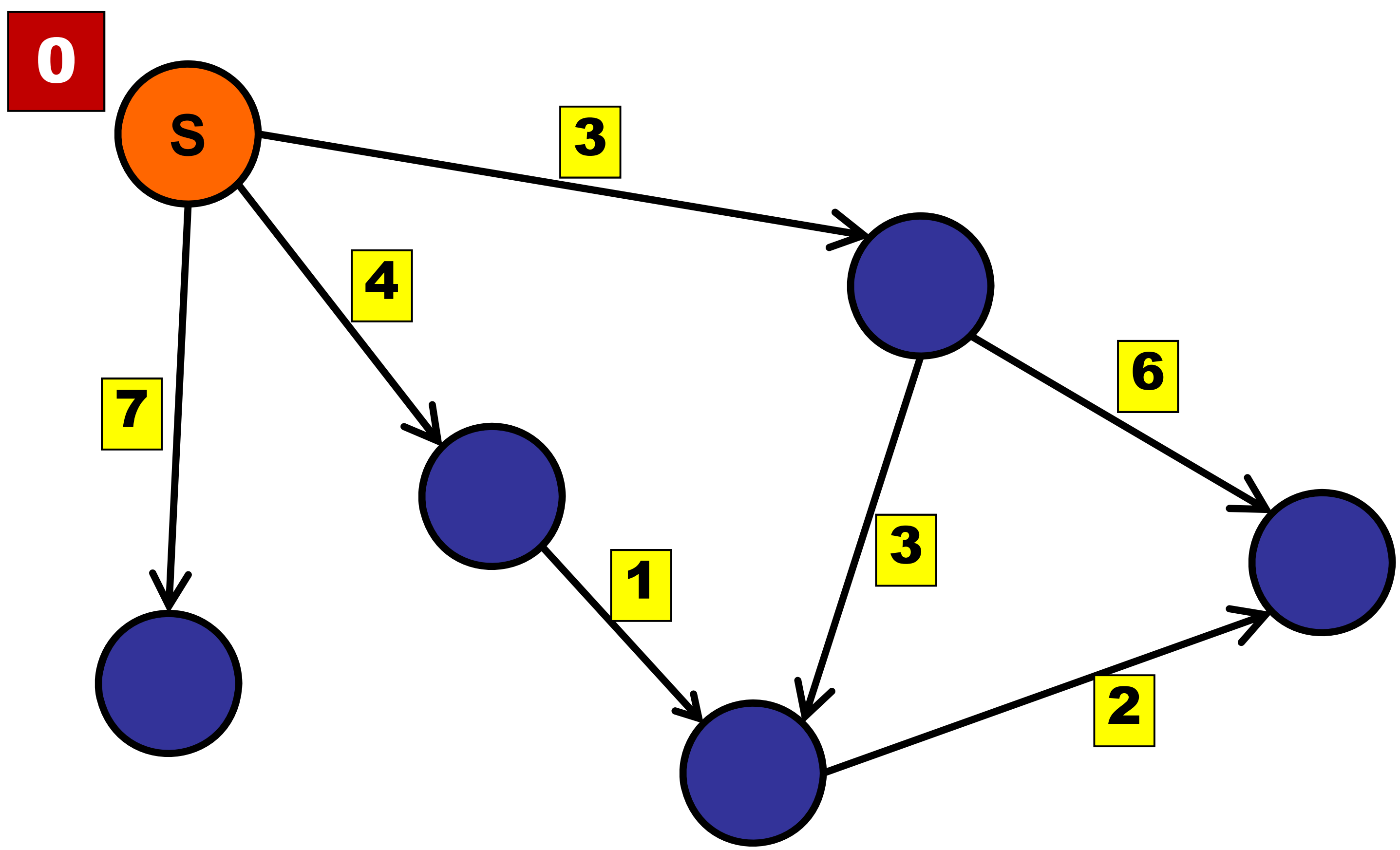
Necessary assumption:

All edges weights >= 0.

Extending a path does not make it shorter!

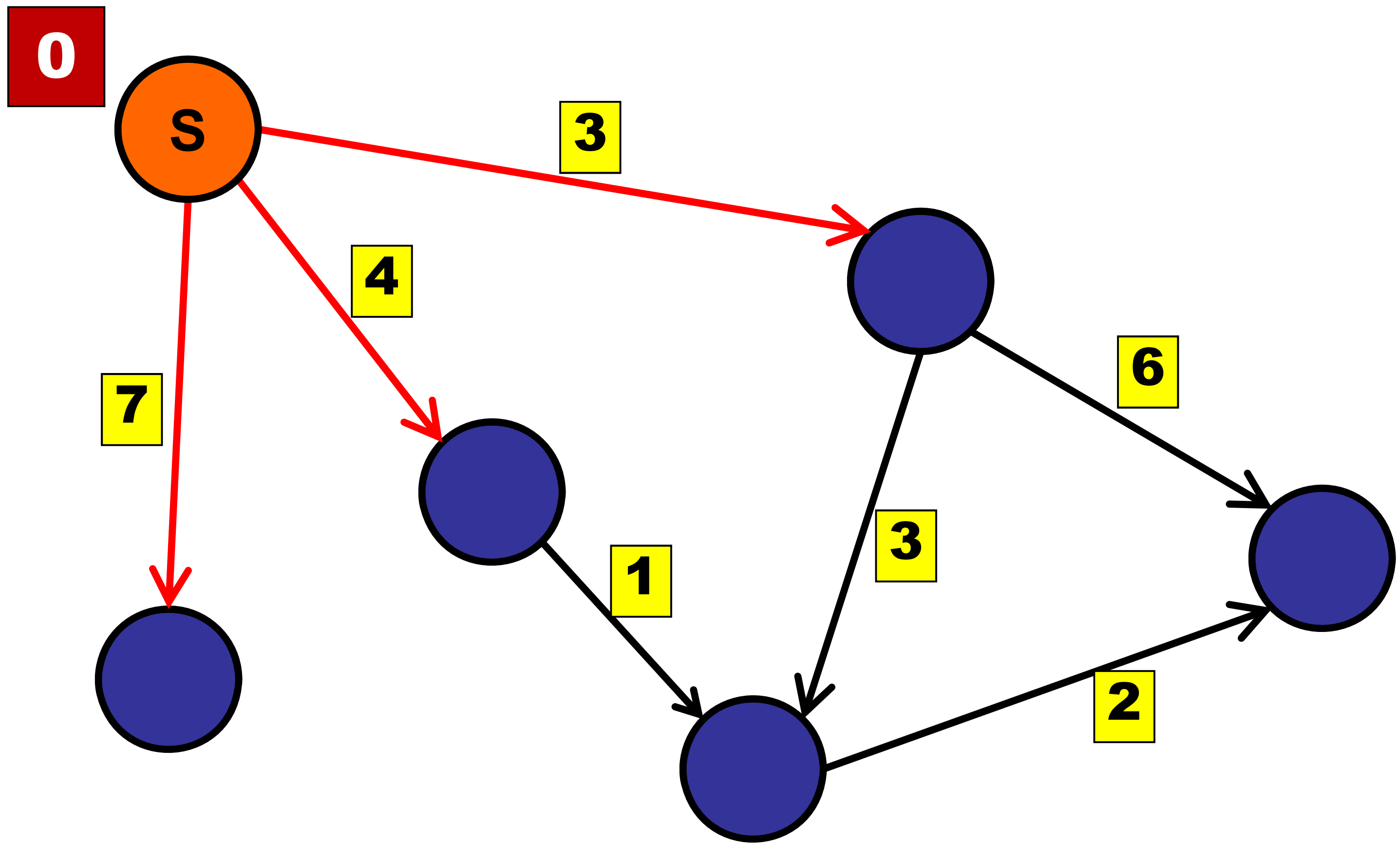
# Dijkstra's Algorithm (First Try)

Relax shortest edge first



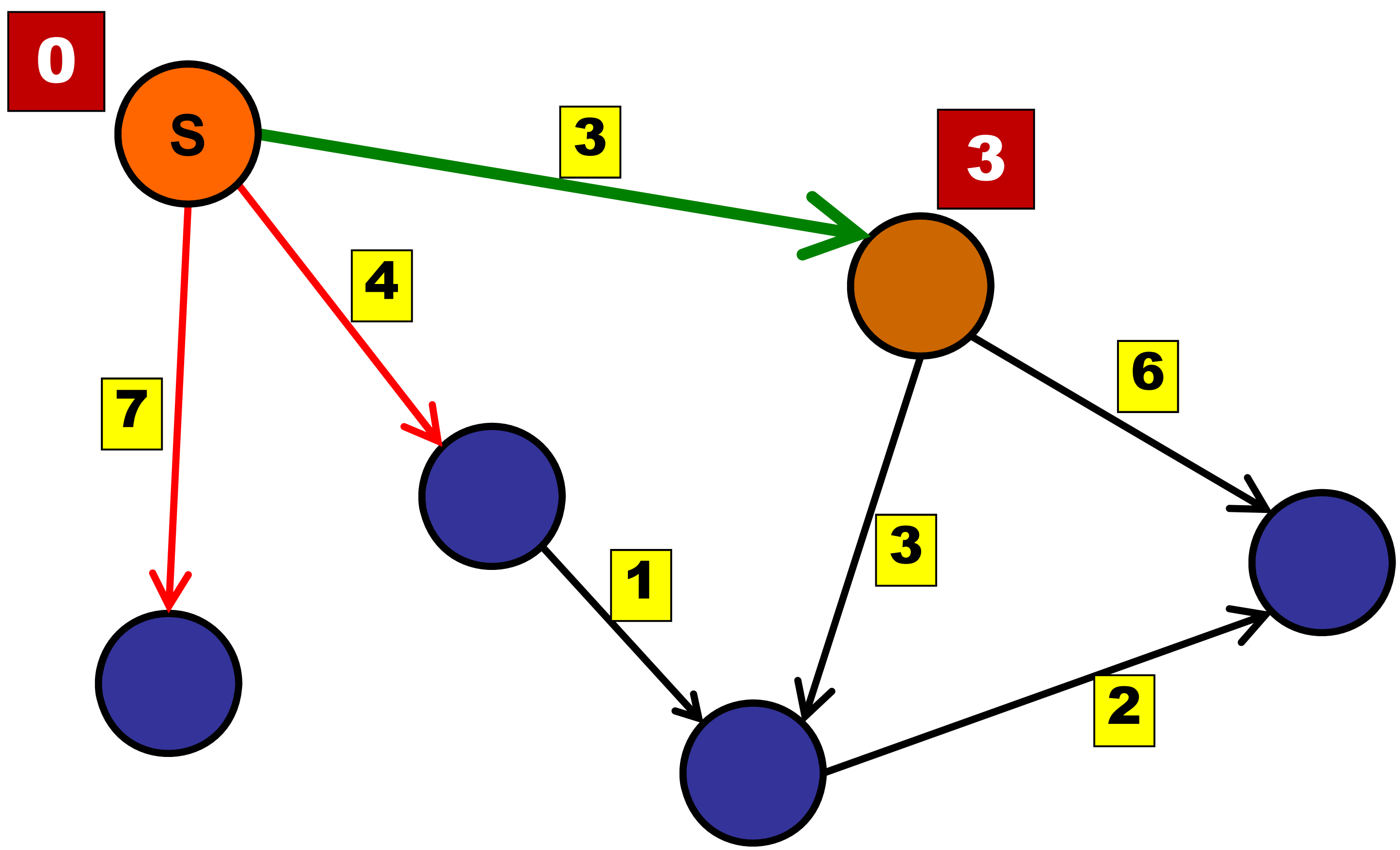
# Dijkstra's Algorithm (First Try)

Relax shortest edge first



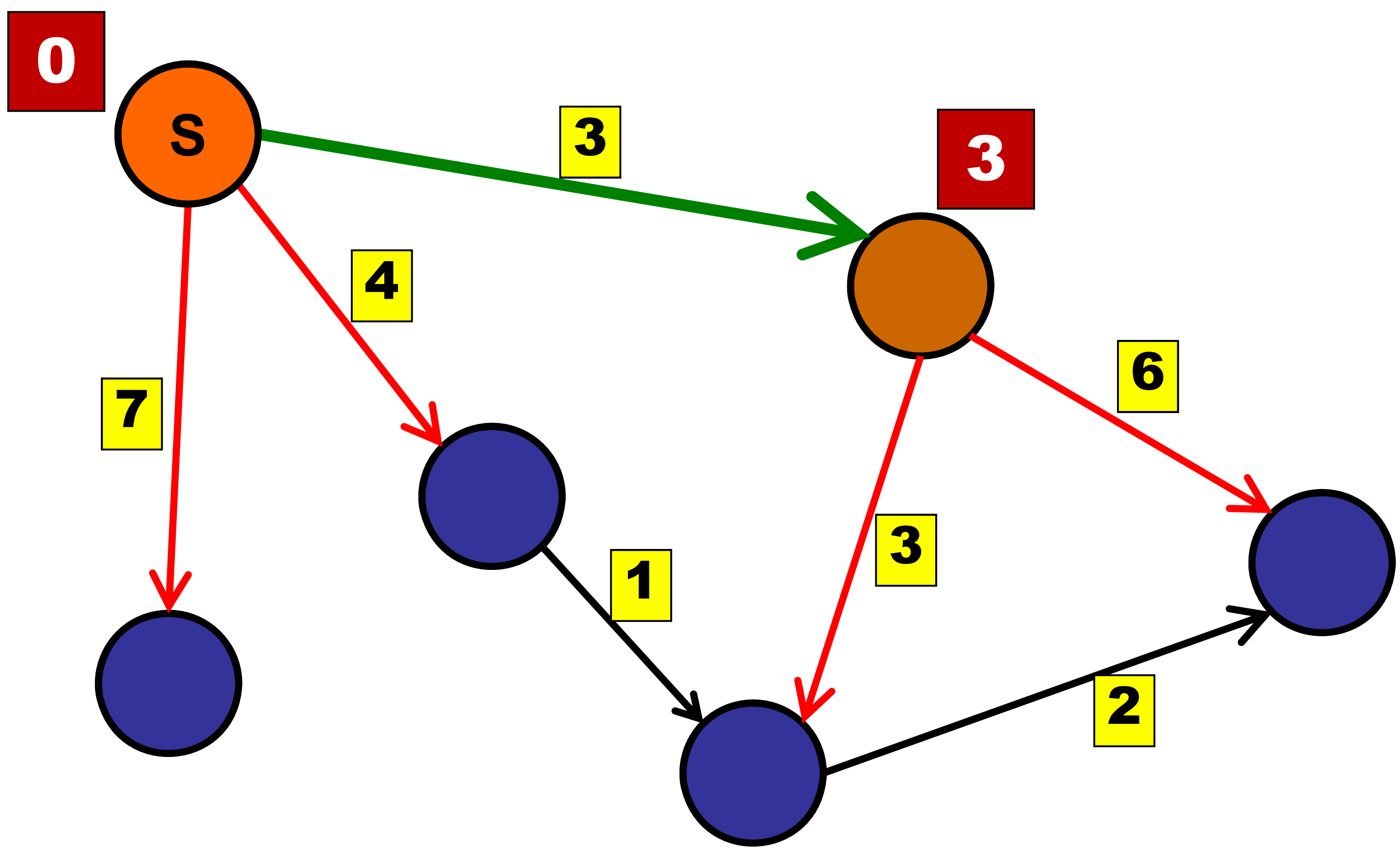
# Dijkstra’s Algorithm (First Try)

Relax shortest edge first



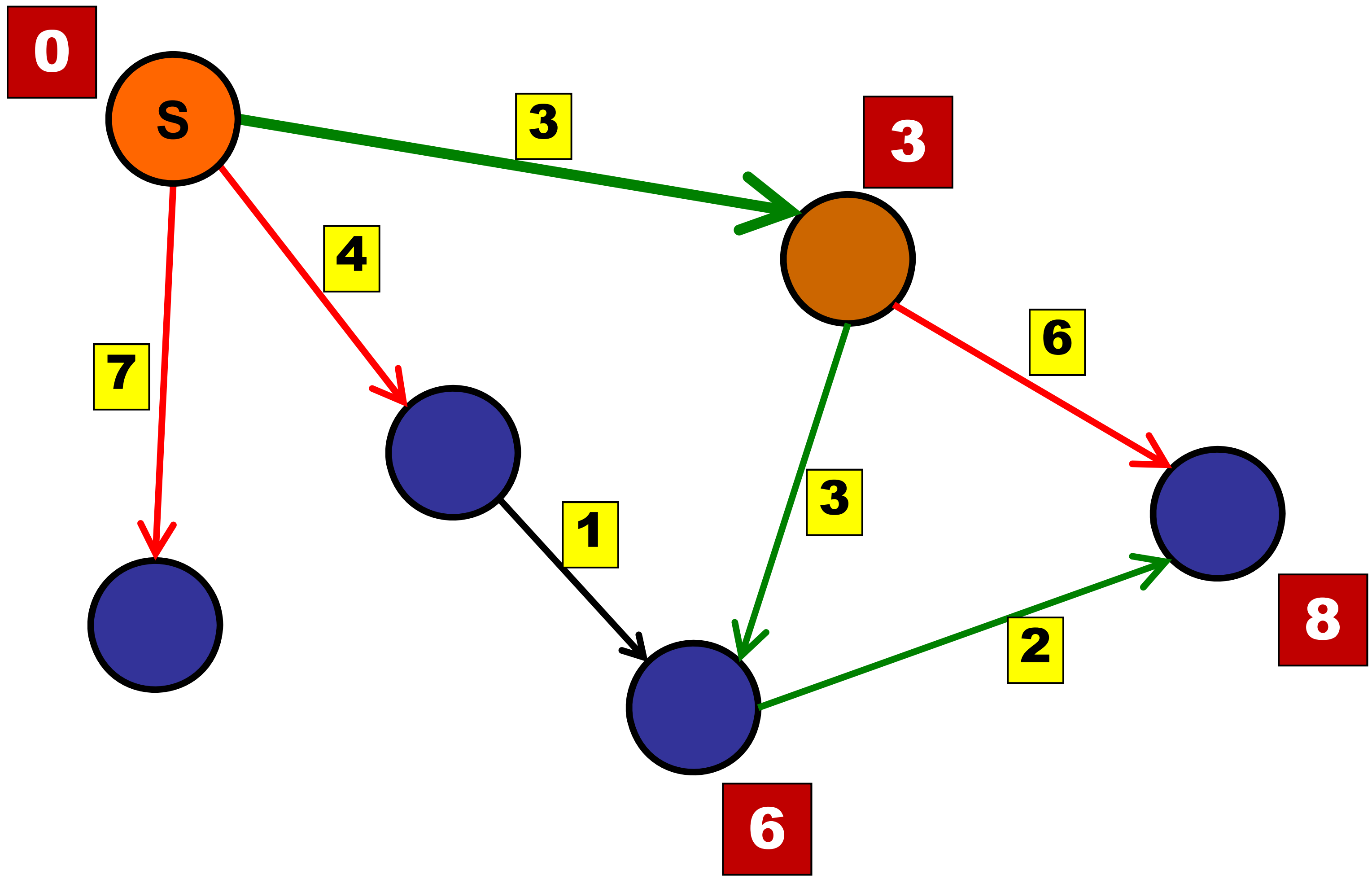
# Dijkstra's Algorithm (First Try)

Relax shortest edge first



# Dijkstra's Algorithm (Failed Try)

Oops….

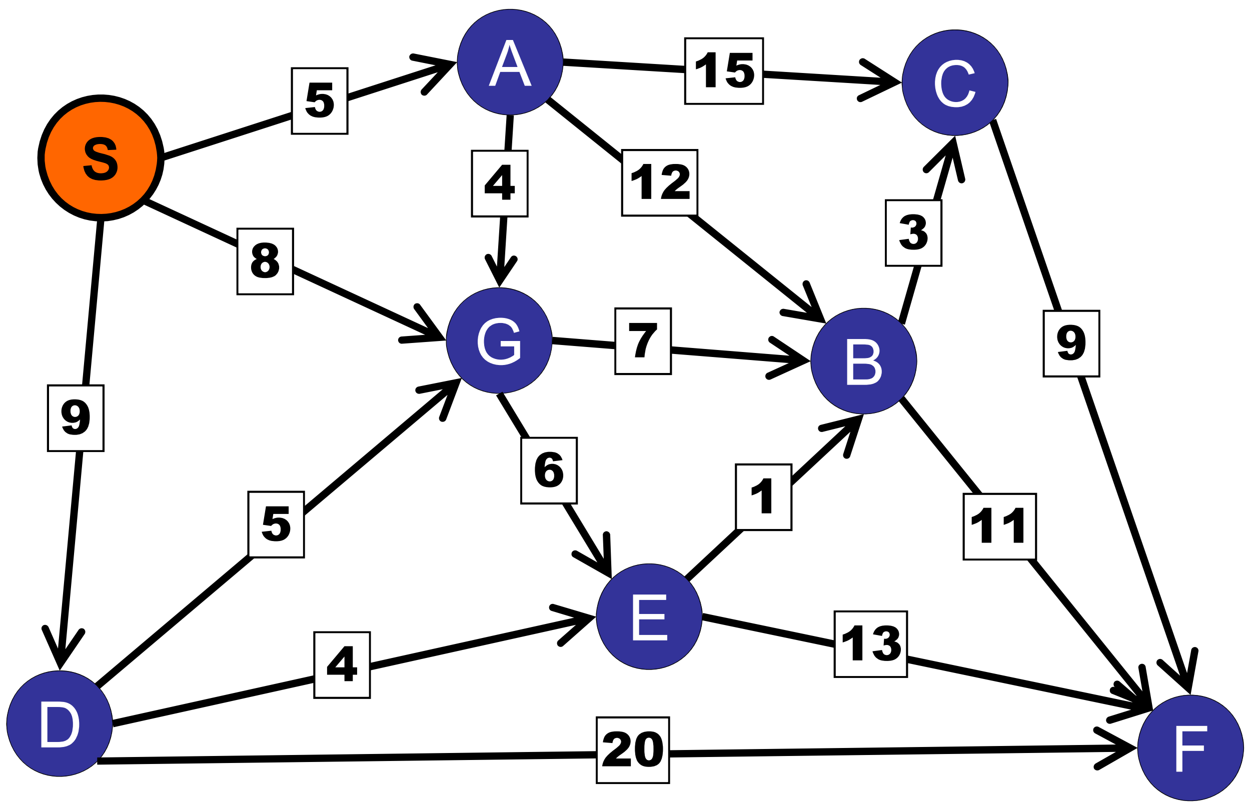


# Dijkstra’s Algorithm

Basic idea:

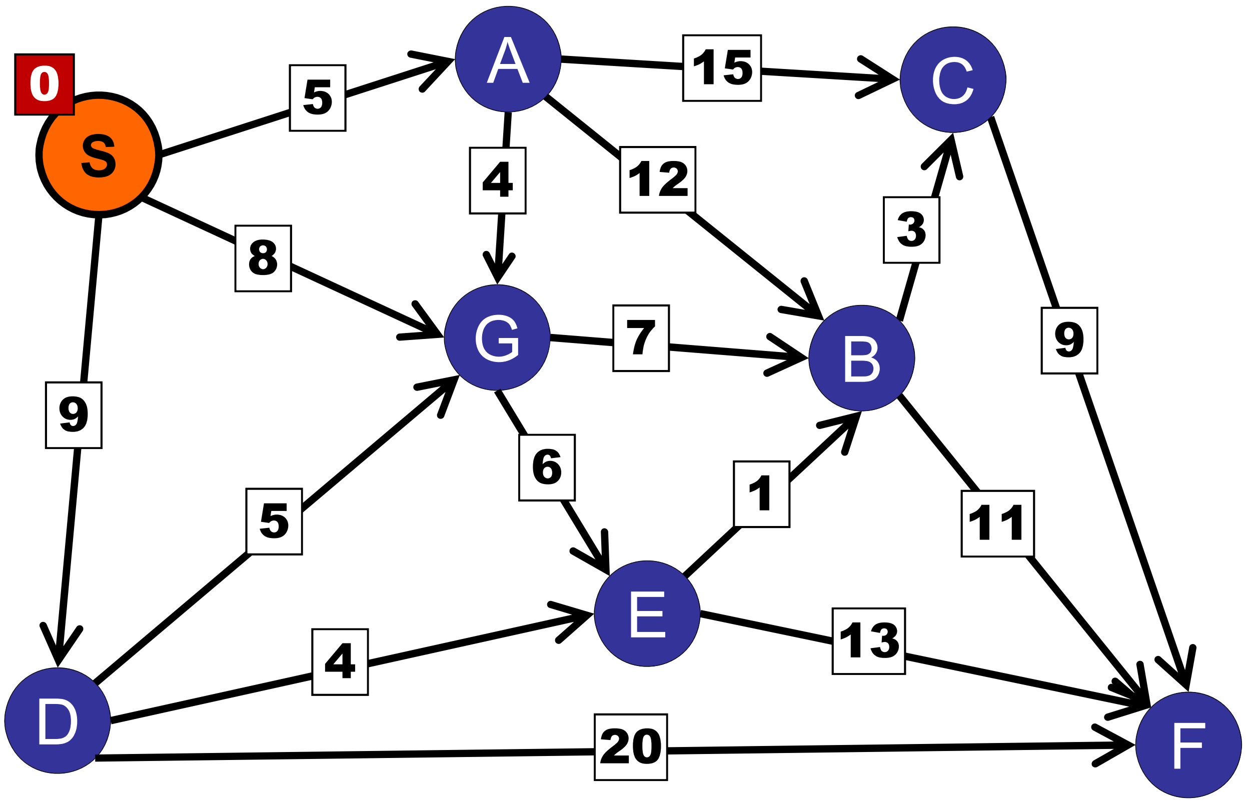
- Maintain distance estimate for every node.
- Begin with empty shortest-path-tree.
- Repeat:
  - Consider vertex with minimum estimate.
  - Add vertex to shortest-path-tree.
  - Relax all outgoing edges.

# Shortest Paths



# Dijkstra's Algorithm

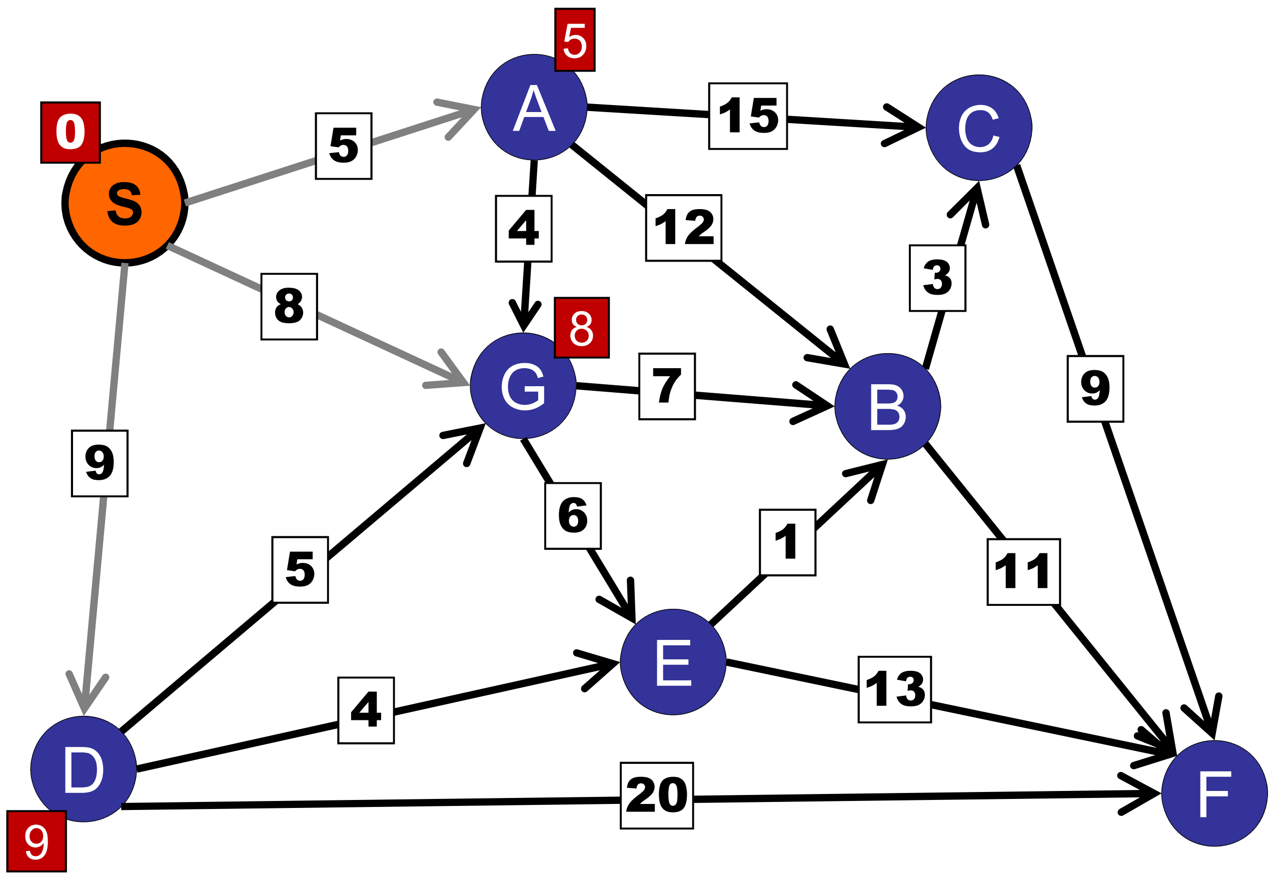
## Step 1: Add source




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

# Dijkstra's Algorithm

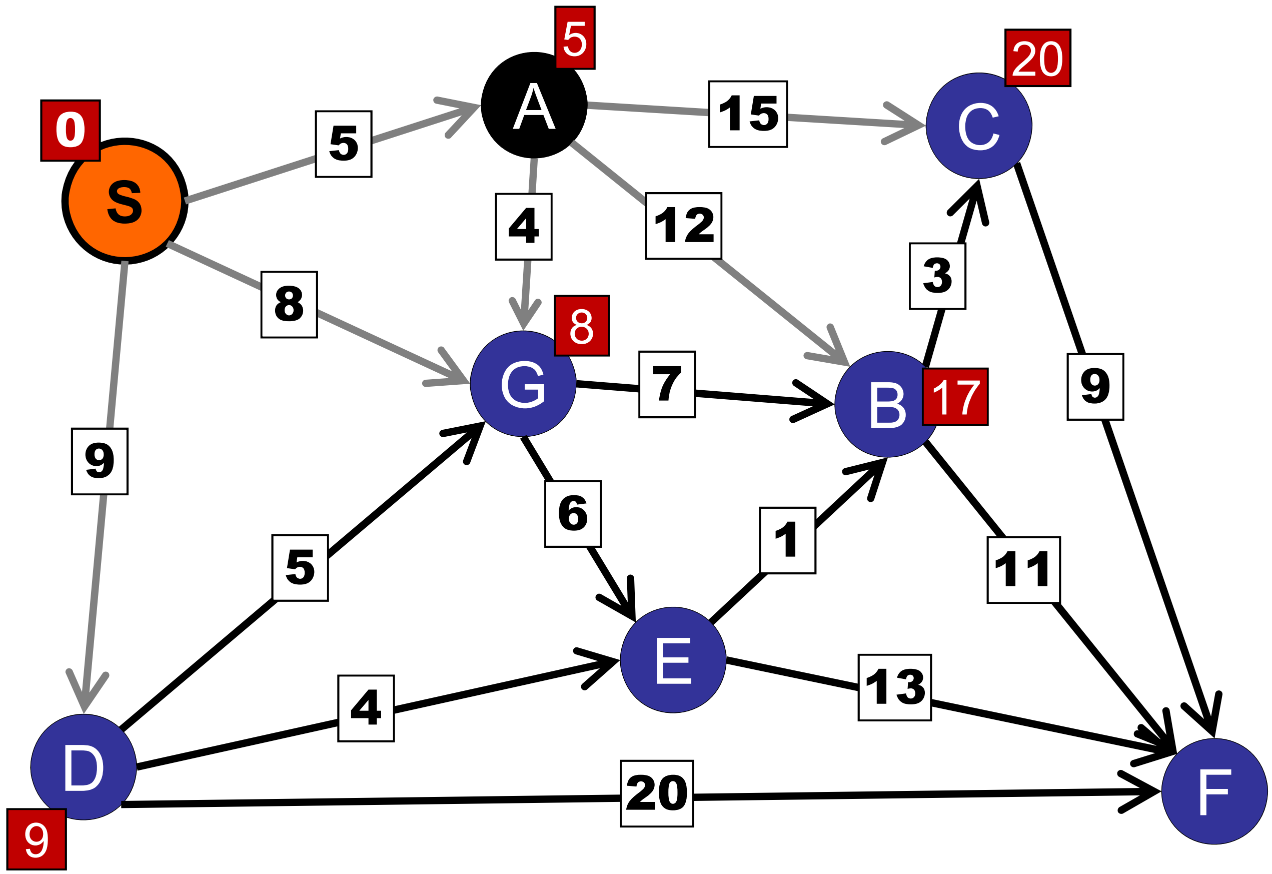
## Step 2: Remove S and relax.




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

# Dijkstra's Algorithm

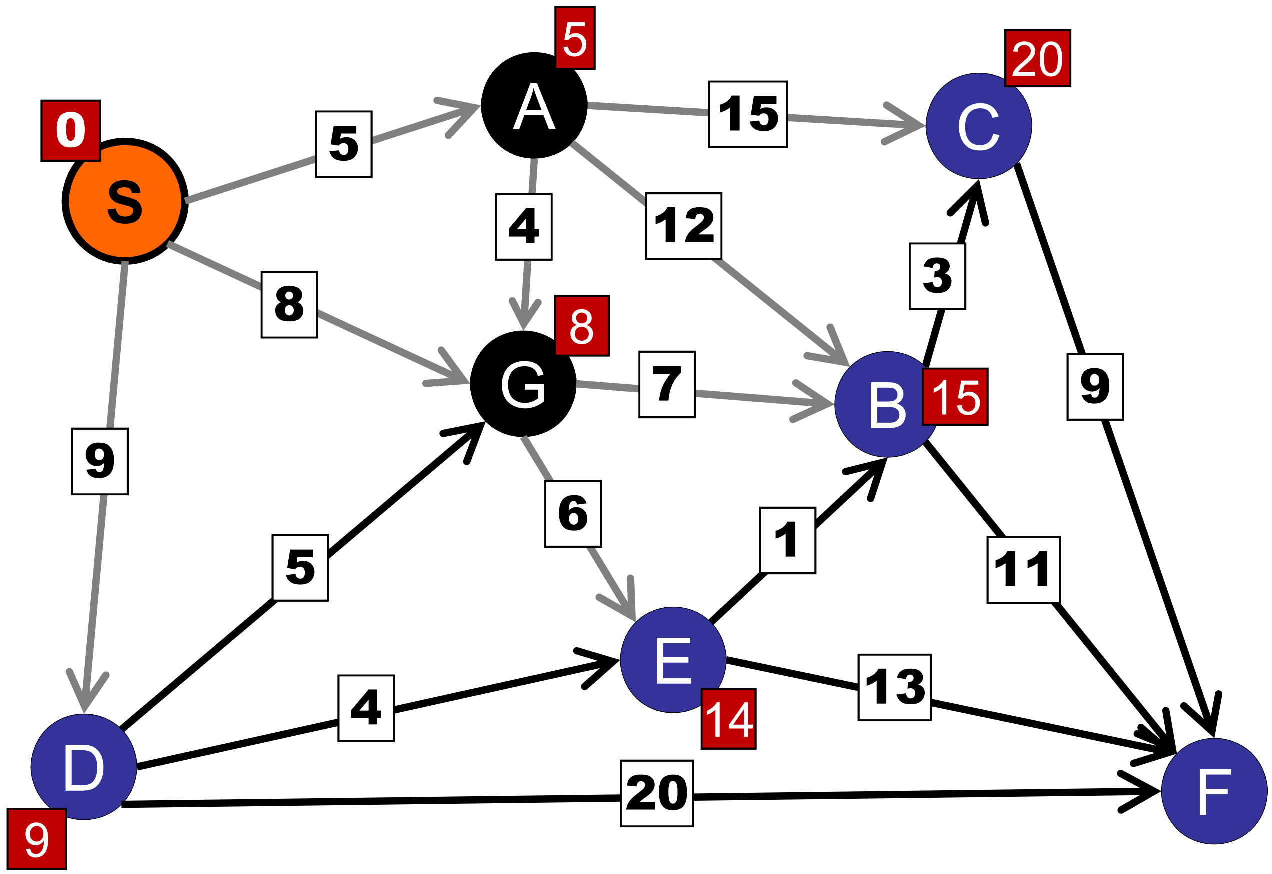
## Step 3: Remove A and relax.




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

# Dijkstra's Algorithm

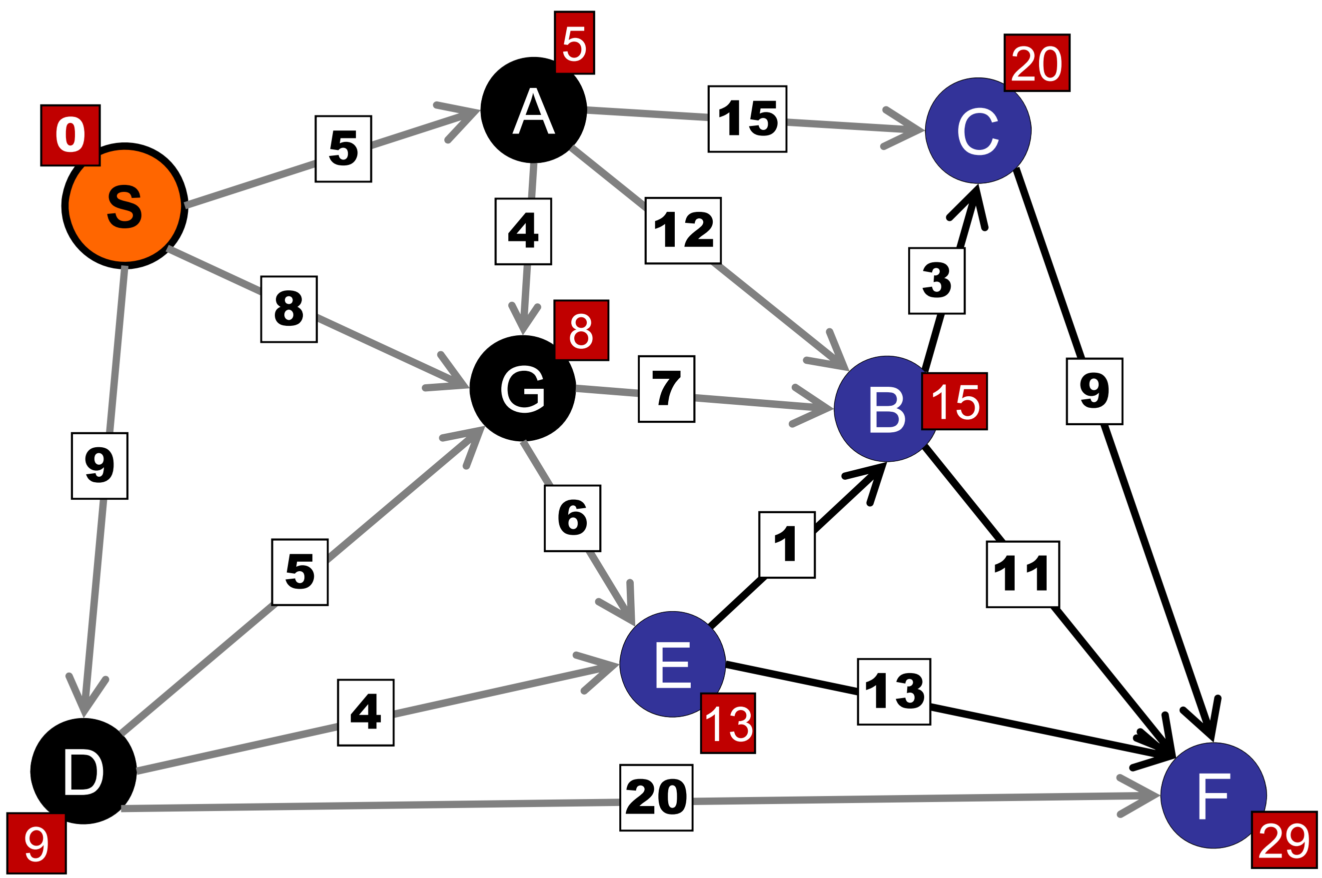
## Step 4: Remove G and relax.




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

# Dijkstra's Algorithm

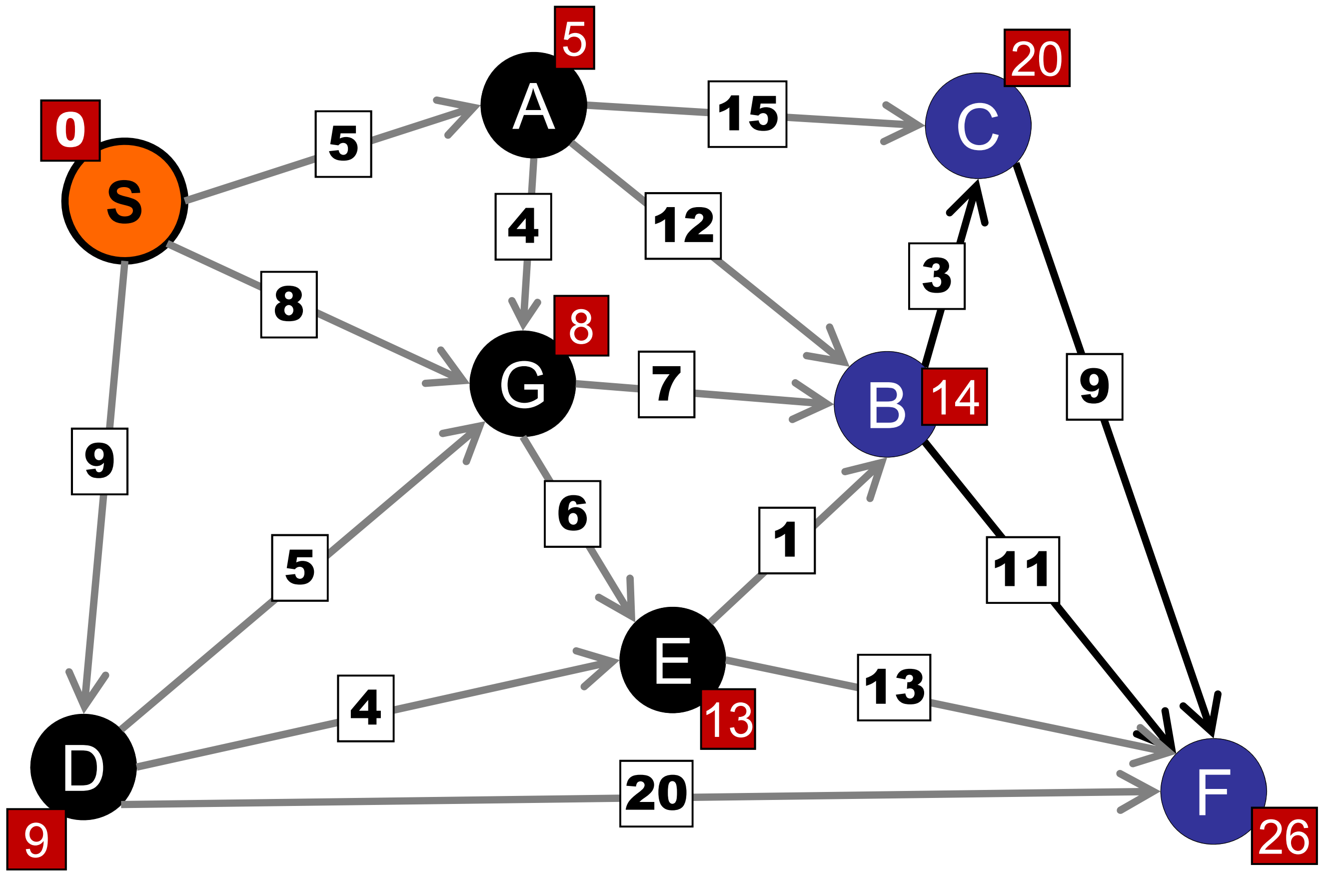
## Step 5: Remove D and relax.




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

# Dijkstra's Algorithm

## Step 5: Remove E and relax.

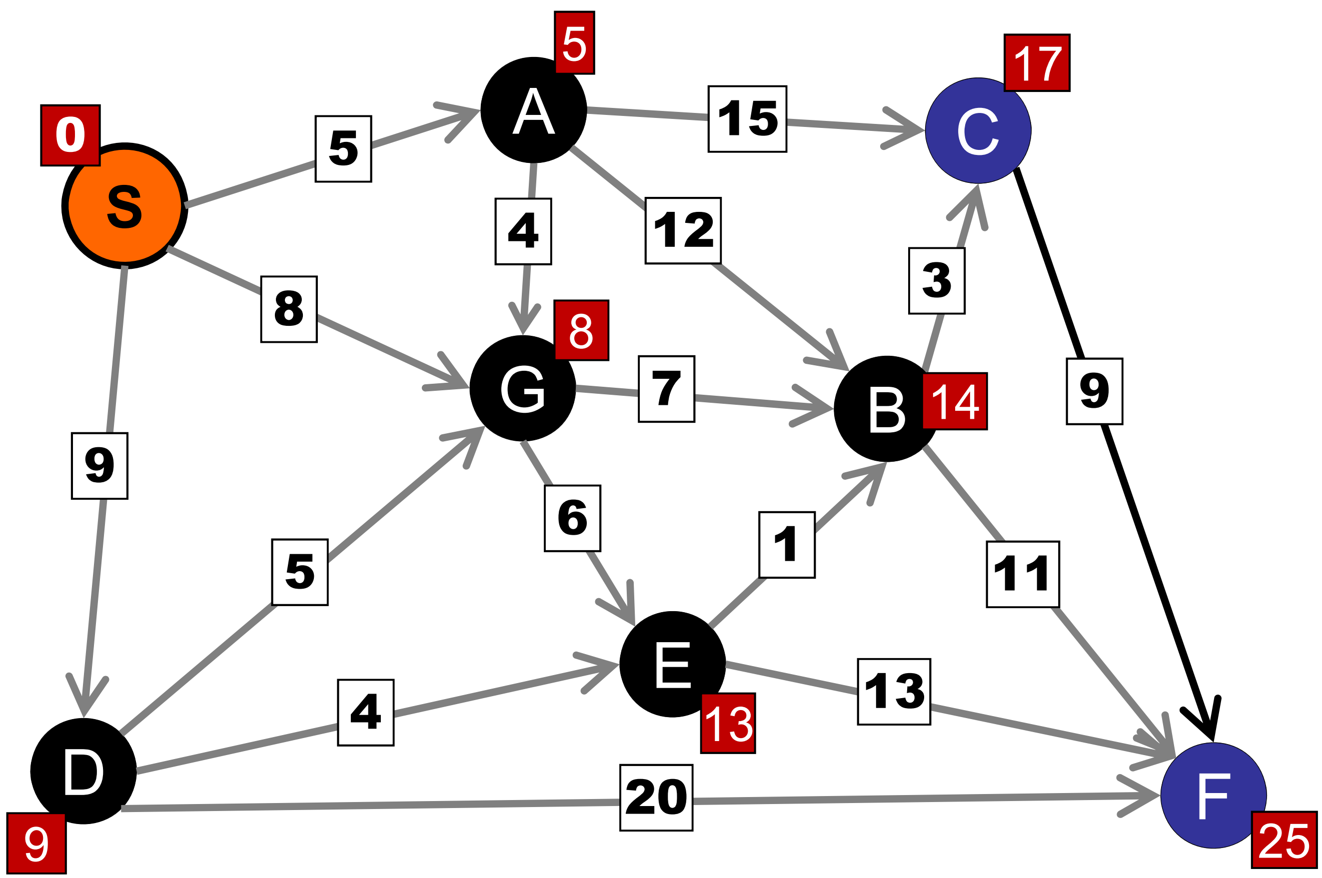



| Vertex | Dist. |
|---|---|
| **B** | **14** |
| C | 20 |
| **F** | **26** |
| | |

# Dijkstra's Algorithm

## Step 5: Remove B and relax.

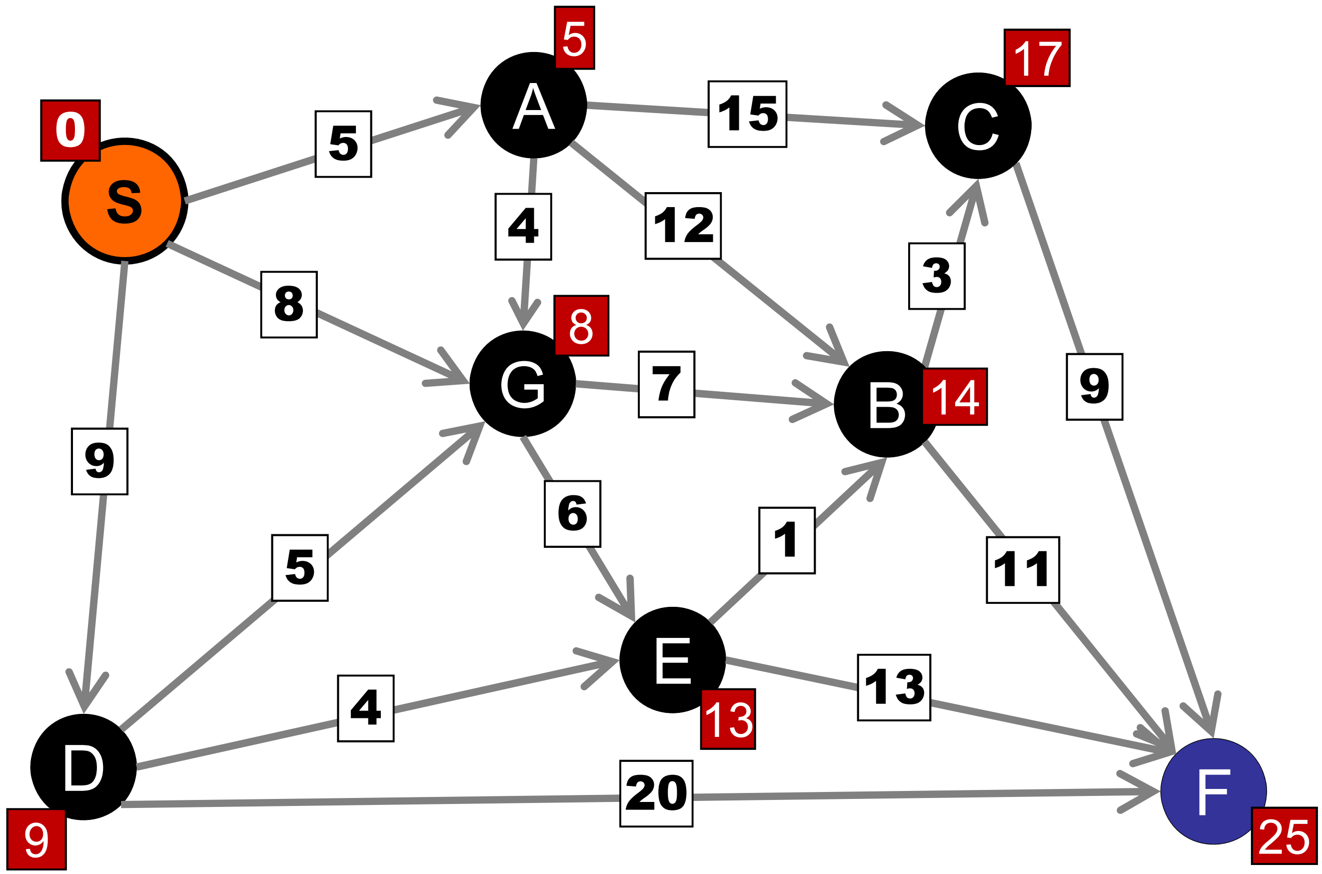
| Vertex | Dist. |
|---|---|
| C | 20 |
| F | 25 |
| | |



# Dijkstra's Algorithm

## Step 5: Remove C and relax.

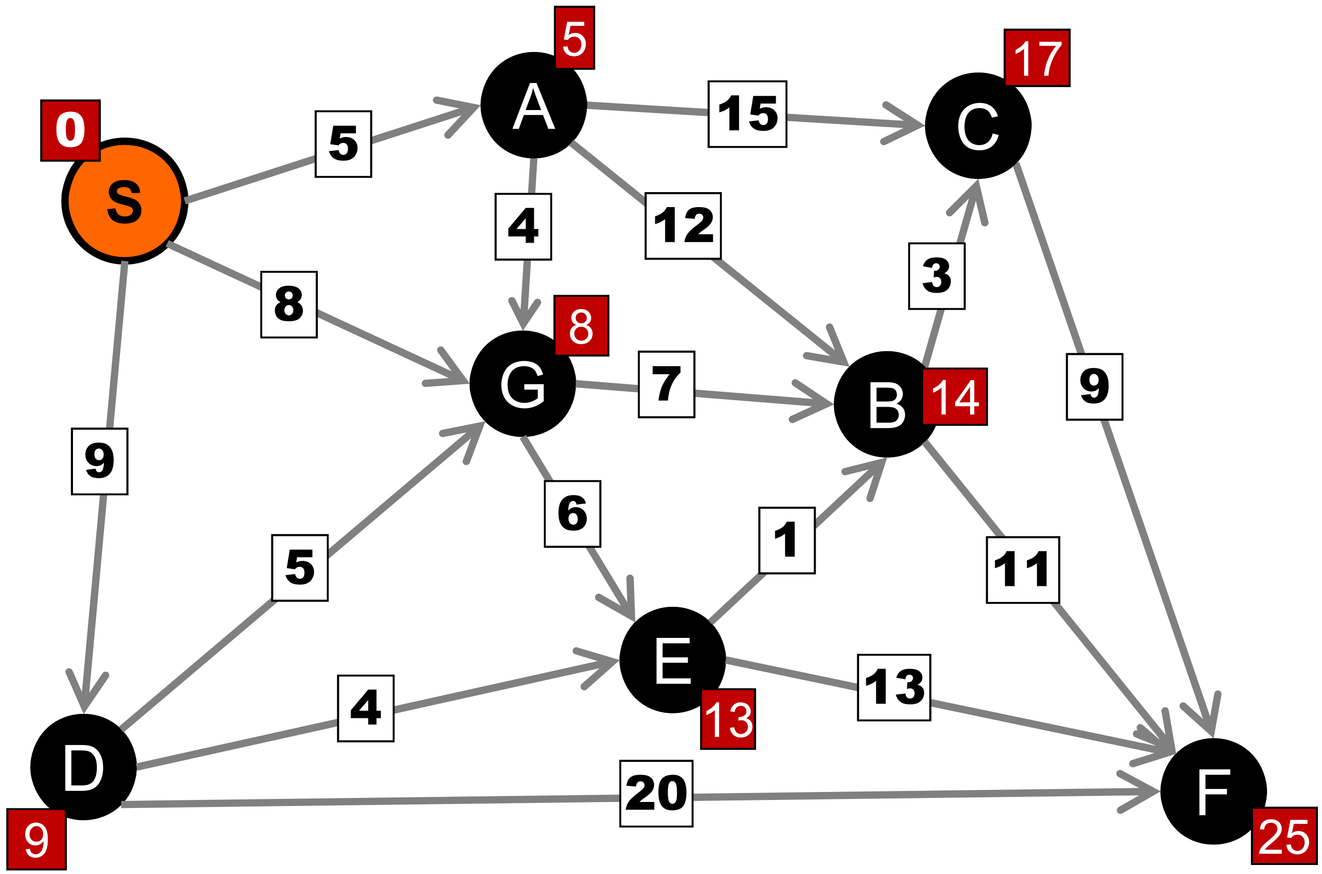
| Vertex | Dist. |
|---|---|
| F | 25 |
| | |



# Dijkstra's Algorithm

## Step 5: Remove F and relax.

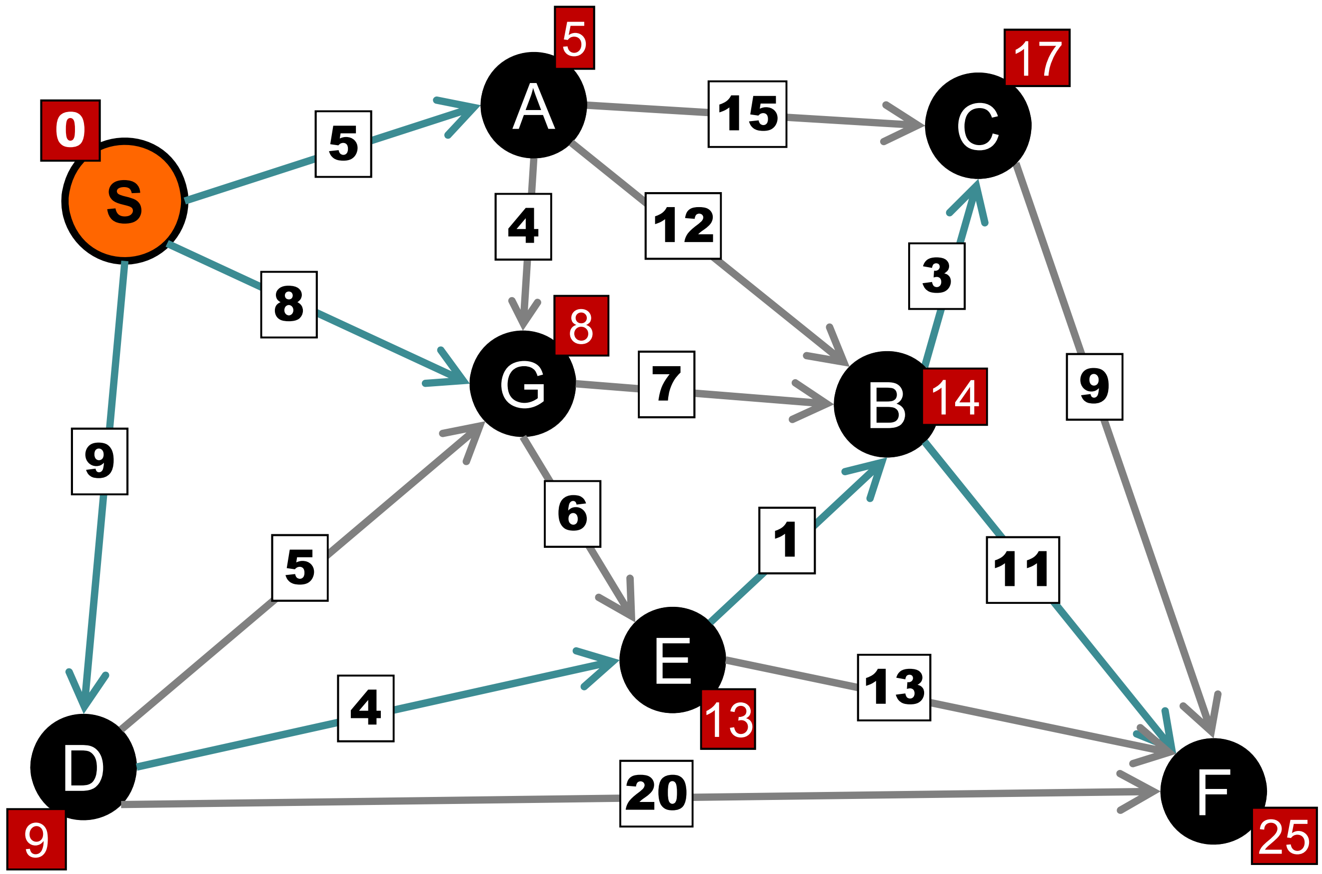
| Vertex | Dist. |
|---|---|
| | |



# Dijkstra's Algorithm

## Done

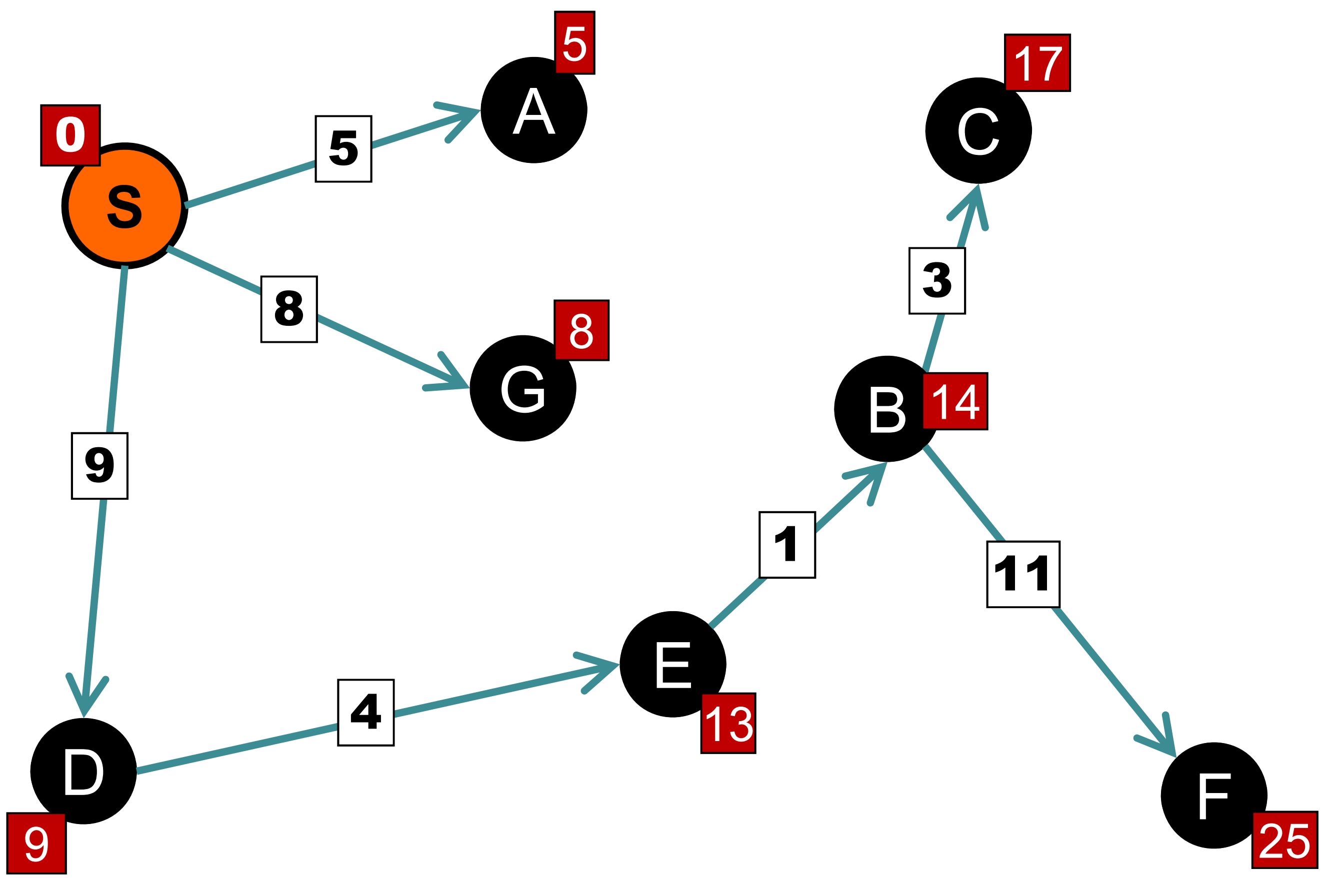
| Vertex | Dist. |
|---|---|
| | |



# Dijkstra’s Algorithm

## Shortest Path Tree

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



What data structure to store vertices/distances?

1. Array
2. Linked list
3. Stack
4. Queue
5. AVL Tree
6. Huh?

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

# Abstract Data Type

## Priority Queue

```
interface  IPriorityQueue<Key, Priority>
```

| | | |
|---|---|---|
| `void` | `insert(Key k, Priority p)` | *insert k with priority p* |
| `Data` | `extractMin()` | *remove key with minimum priority* |
| `void` | `decreaseKey(Key k, Priority p)` | *reduce the priority of key k to priority p* |
| `boolean` | `contains(Key k)` | *does the priority queue contain key k?* |
| `boolean` | `isEmpty()` | *is the priority queue empty?* |

Notes:

Assume data items are unique.

```
public Dijkstra{
      private Graph G;
      private IPriorityQueue pq = new PriQueue();
      private double[] distTo;

      searchPath(int start) {
            pq.insert(start, 0.0);
            distTo = new double[G.size()];
            Arrays.fill(distTo, INFTY);
            distTo[start] = 0;
            while (!pq.isEmpty()) {
                  int w = pq.deleteMin();
                  for (Edge e : G[w].nbrList)
                        relax(e);
            }
      }
```

# Dijkstra’s Algorithm

```
relax(Edge e) {
    int v = e.from();
    int w = e.to();
    double weight = e.weight();
    if (distTo[w] > distTo[v] + weight) {
        distTo[w] = distTo[v] + weight;
        parent[w] = v;
        if (pq.contains(w))
            pq.decreaseKey(w, distTo[w]);
        else
            pq.insert(w, distTo[w]);
    }
}
```

# Abstract Data Type

## Priority Queue

```
interface  IPriorityQueue<Key, Priority>
```

| | | |
|---|---|---|
| `void` | `insert(Key k, Priority p)` | *insert k with priority p* |
| `Data` | `extractMin()` | *remove key with minimum priority* |
| `void` | `decreaseKey(Key k, Priority p)` | *reduce the priority of key k to priority p* |
| `boolean` | `contains(Key k)` | *does the priority queue contain key k?* |
| `boolean` | `isEmpty()` | *is the priority queue empty?* |

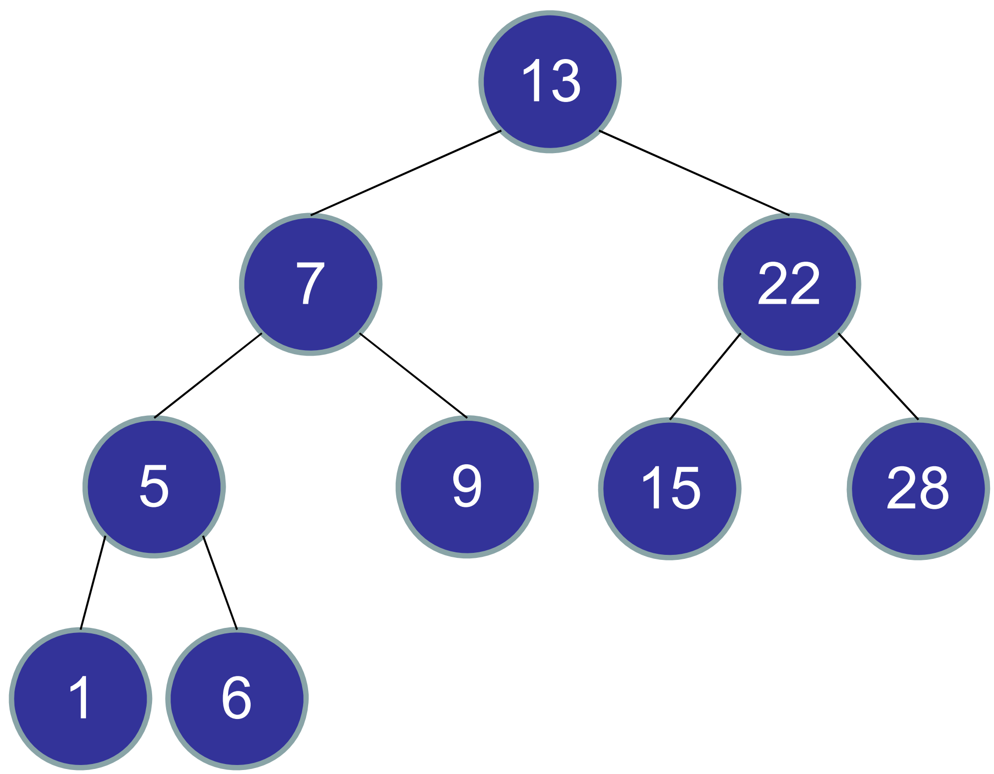
<u>Notes:</u>

Assume data items are unique.

# Priority Queue

AVL Tree

- Indexed by: priority
- Existing operations:
  - deleteMin()
  - insert(key, priority)

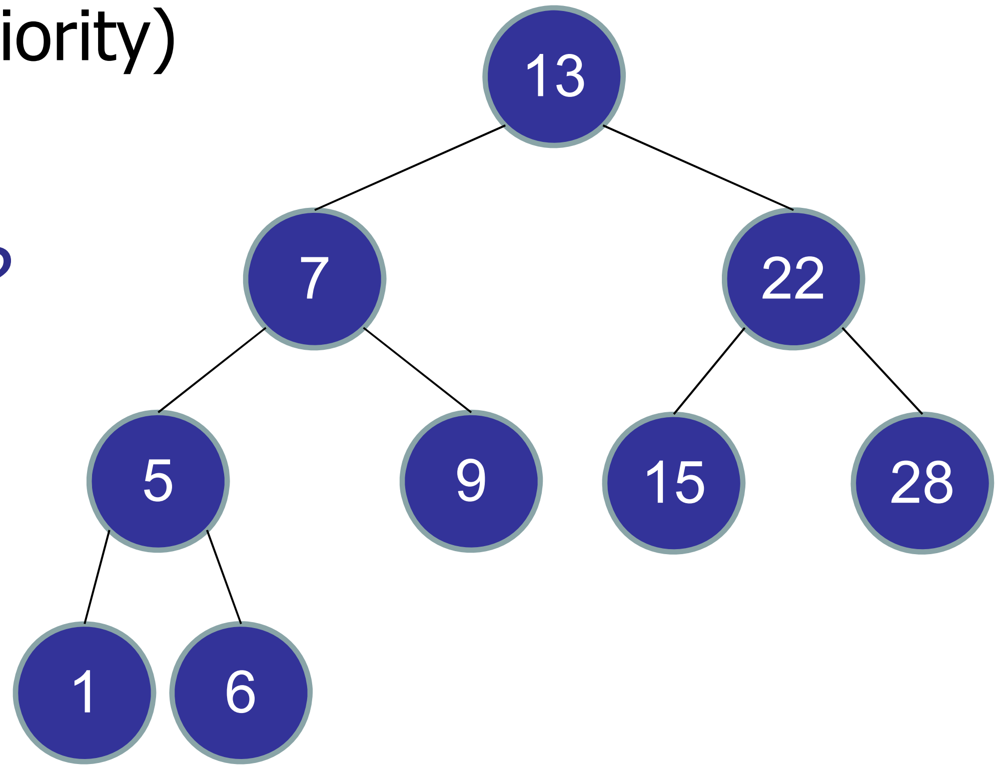


# Priority Queue

AVL Tree

- Other operations:
  - contains()
  - decreaseKey(key, priority)

- How to find a vertex?

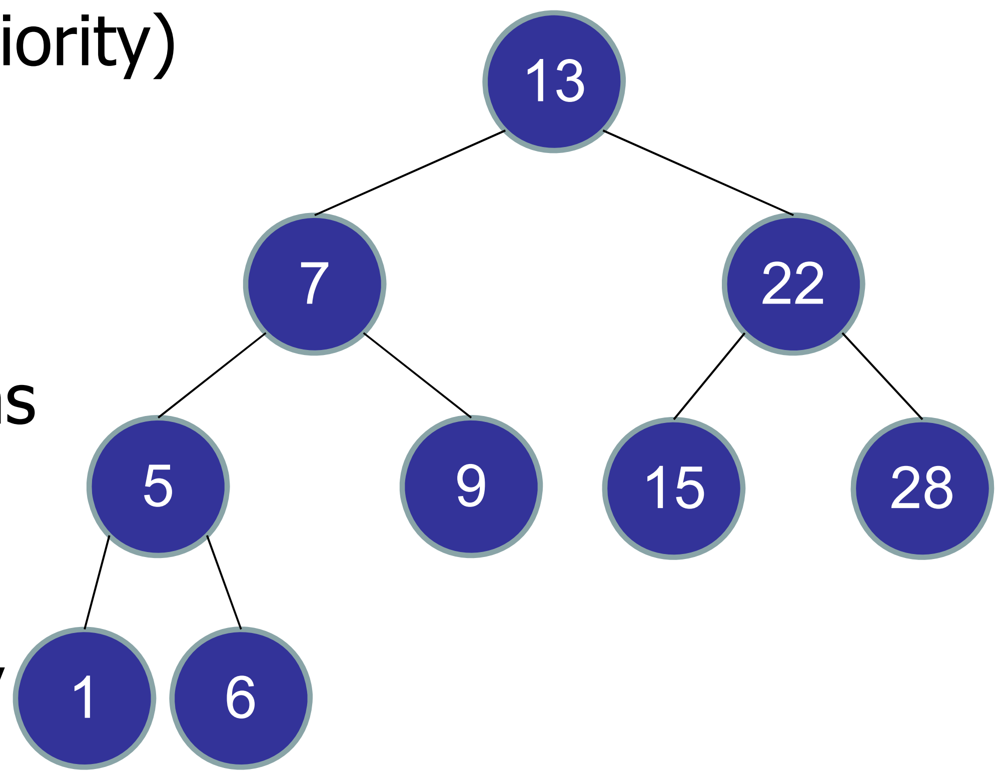


# Priority Queue

AVL Tree

- Other operations:
  - contains()
  - decreaseKey(key, priority)

- Hash Table:
  - Map keys to locations in the binary tree.
  - Update hash table whenever the binary tree changes.



# Dijkstra's Algorithm

Priority Queue by AVL tree:

- insert(key, priority): O(log n)
- deleteMin(): O(log n)
- decreaseKey(key, priority): O(log n)
- contains(key): O(1)

What is the running time of Dijkstra's Algorithm, using an AVL tree Priority Queue?

1. $O(V + E)$
2. $O(E \log V)$
3. $O(V \log E)$
4. $O(V^2)$
5. $O(VE)$
6. None of the above

```
public Dijkstra{
     private Graph G;
     private MinPriQueue pq = new MinPriQueue();
     private double[] distTo;

     searchPath(int start) {
          pq.insert(start, 0.0);
          distTo = new double[G.size()];
          Arrays.fill(distTo, INFTY);
          distTo[start] = 0;
          while (!pq.isEmpty()) {
               int w = pq.deleteMin();
               for (Edge e : G[w].nbrList)
                    relax(e);
          }
     }
```

How many times?

How many times?

# Dijkstra’s Algorithm

```
relax(Edge e) {
    int v = e.from();
    int w = e.to();
    double weight = e.weight();
    if (distTo[w] > distTo[v] + weight) {
        distTo[w] = distTo[v] + weight;
        parent[w] = v;
        if (pq.contains(w))
            pq.decreaseKey(w, distTo[w]);
        else
            pq.insert(w, distTo[w]);
    }
}
```

# Dijkstra's Algorithm

Analysis:

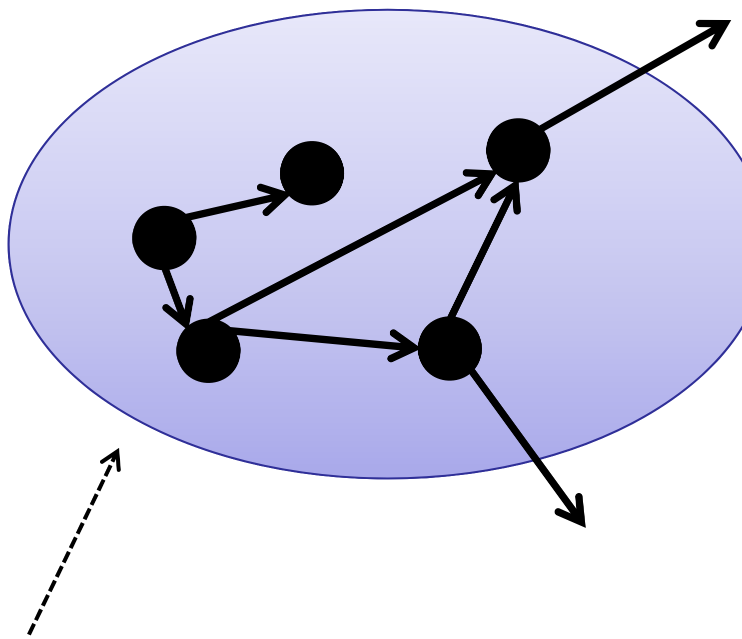
- insert / deleteMin: |V| times each
  - Each node is added to the priority queue **once**.
- relax / decreaseKey: |E| times
  - Each edge is relaxed once.
- Priority queue operations: O(log V)
- Total: O((V+E)log V) = O(E log V)

# Dijkstra’s Algorithm

Why does it work?

# Dijkstra’s Algorithm

Every edge crossing the boundary has been relaxed.

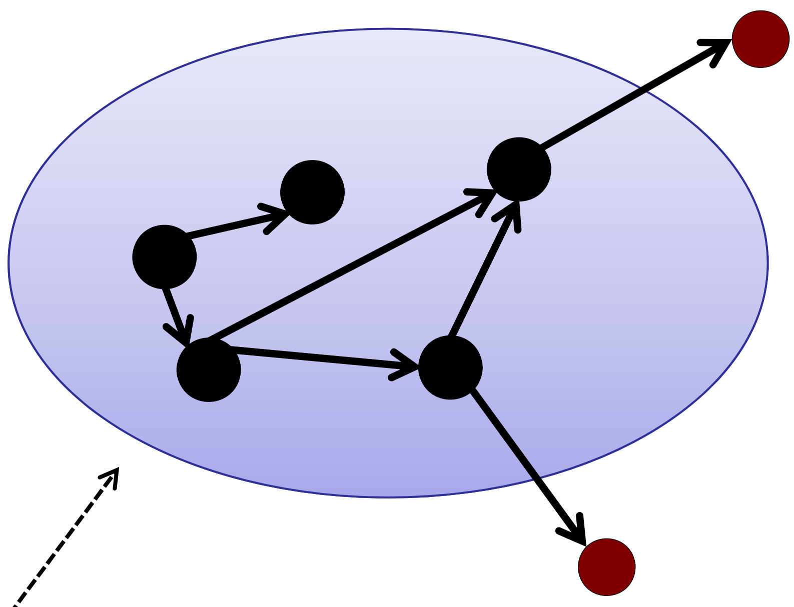


**finished vertices:**
**distance is accurate.**
**Initially: just the source.**

# Dijkstra's Algorithm

**fringe vertices: neighbor of a finished vertex.**

Every edge crossing the boundary has been relaxed.



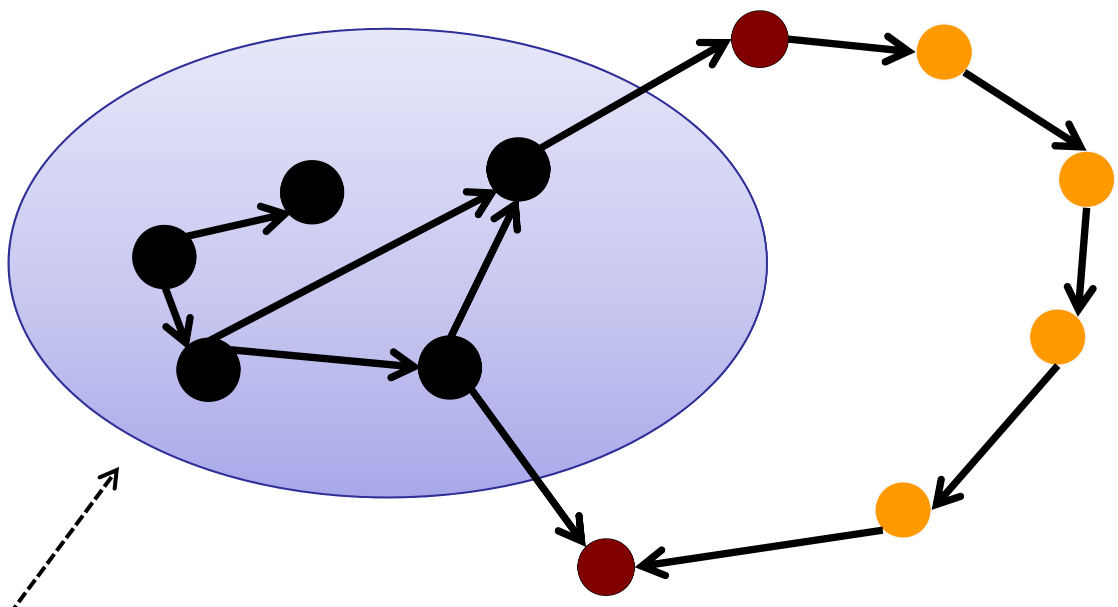
**finished vertices: distance is accurate.**

**fringe vertices: in priority queue neighbor of a finished vertex.**

# Dijkstra’s Algorithm

Every edge crossing the boundary has been relaxed.

**fringe vertices: neighbor of a finished vertex.**



**other vertices: no known estimate**

**finished vertices: distance is accurate.**

**fringe vertices: in priority queue neighbor of a finished vertex.**

# Dijkstra’s Algorithm

Proof by induction:

- Every “finished” vertex has correct estimate.
- Initially: only “finished” vertex is start.

# Dijkstra’s Algorithm

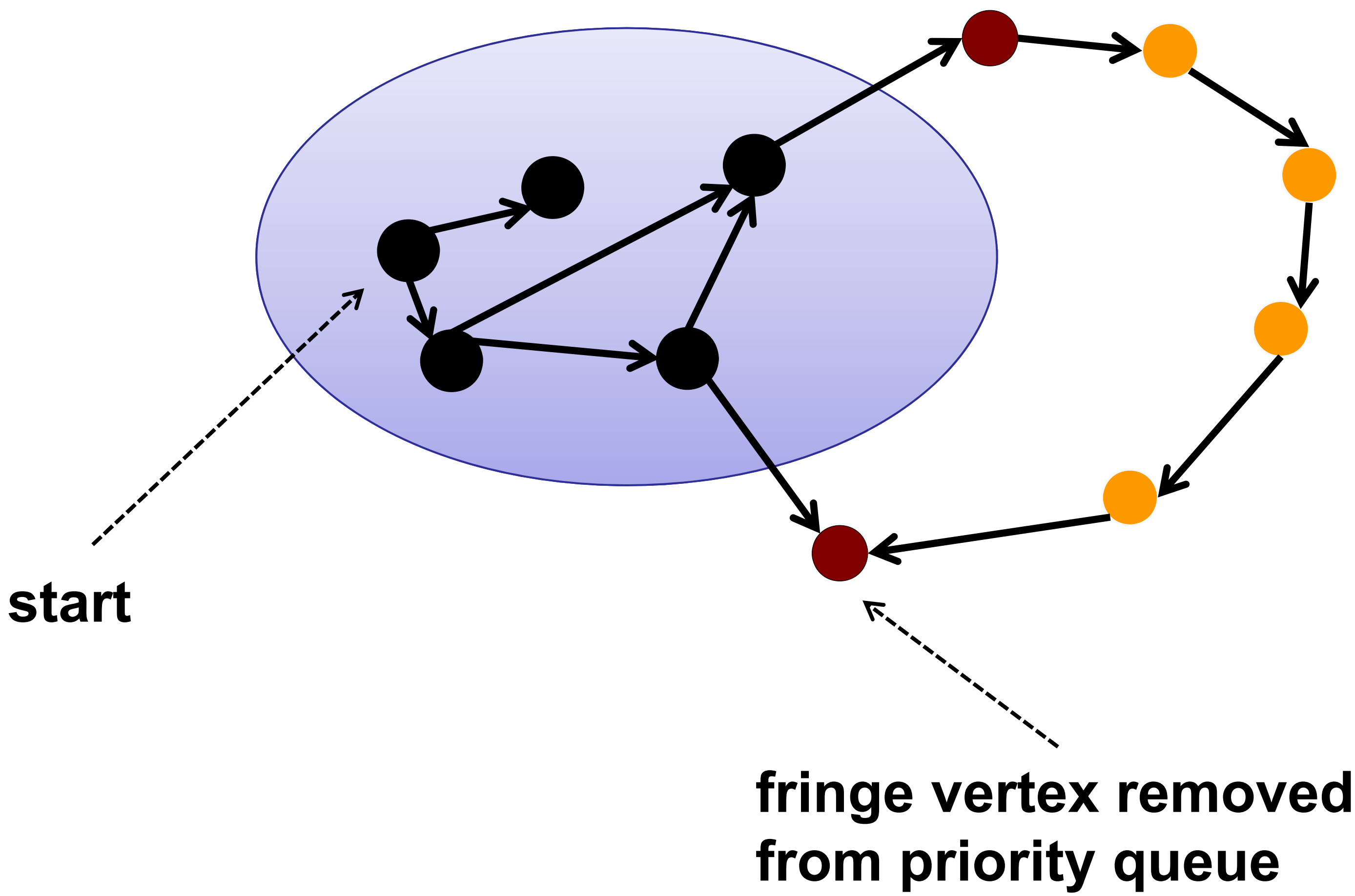
Proof by induction:

- Every “finished” vertex has correct estimate.
- Initially: only “finished” vertex is start.

- Inductive step:
  - Remove vertex from priority queue.
  - Relax its edges.
  - Add it to finished.
  - **Claim: it has a correct estimate.**

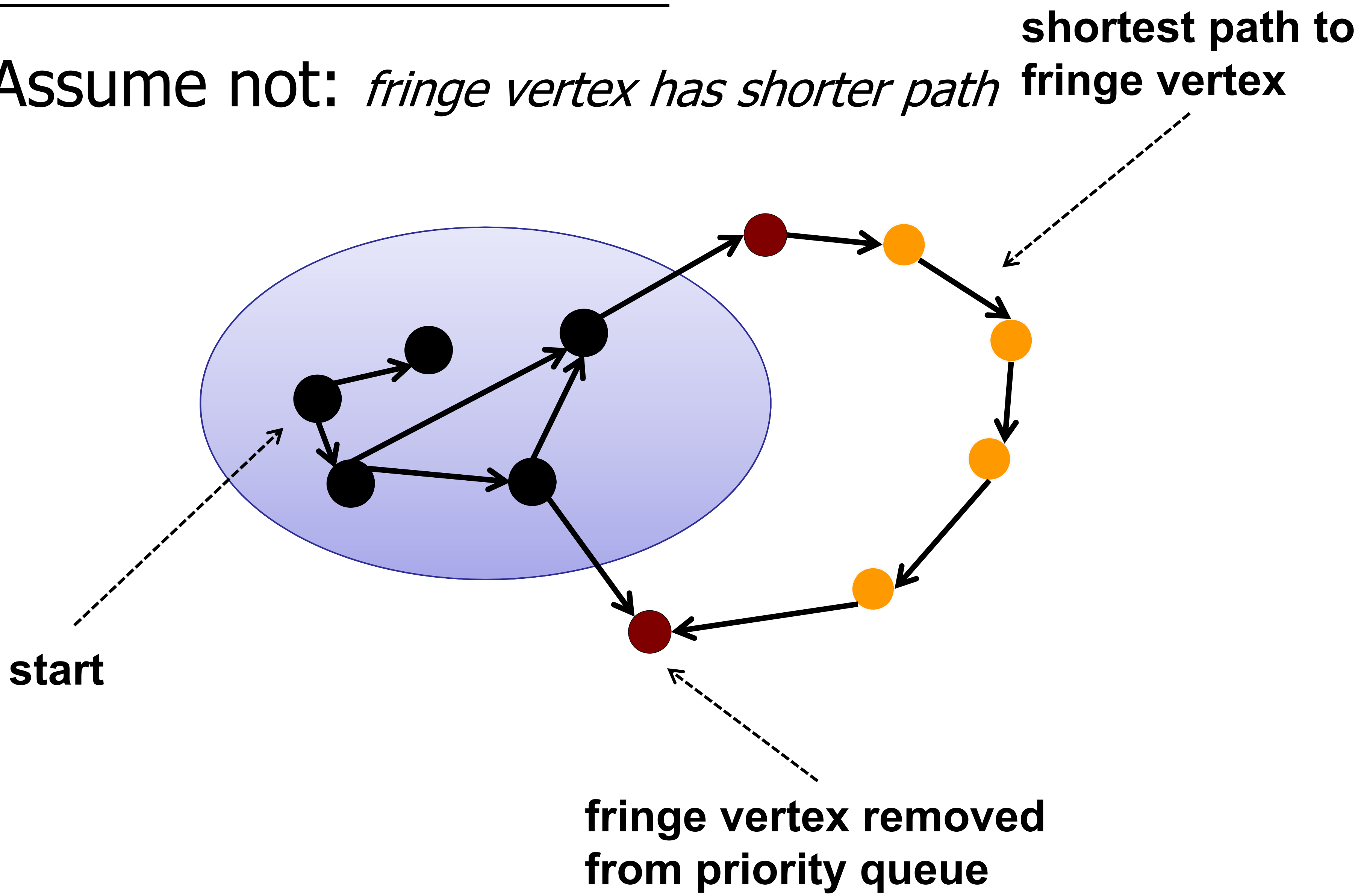
# Dijkstra’s Algorithm

Assume not: *fringe vertex is removed but not done*



# Dijkstra's Algorithm

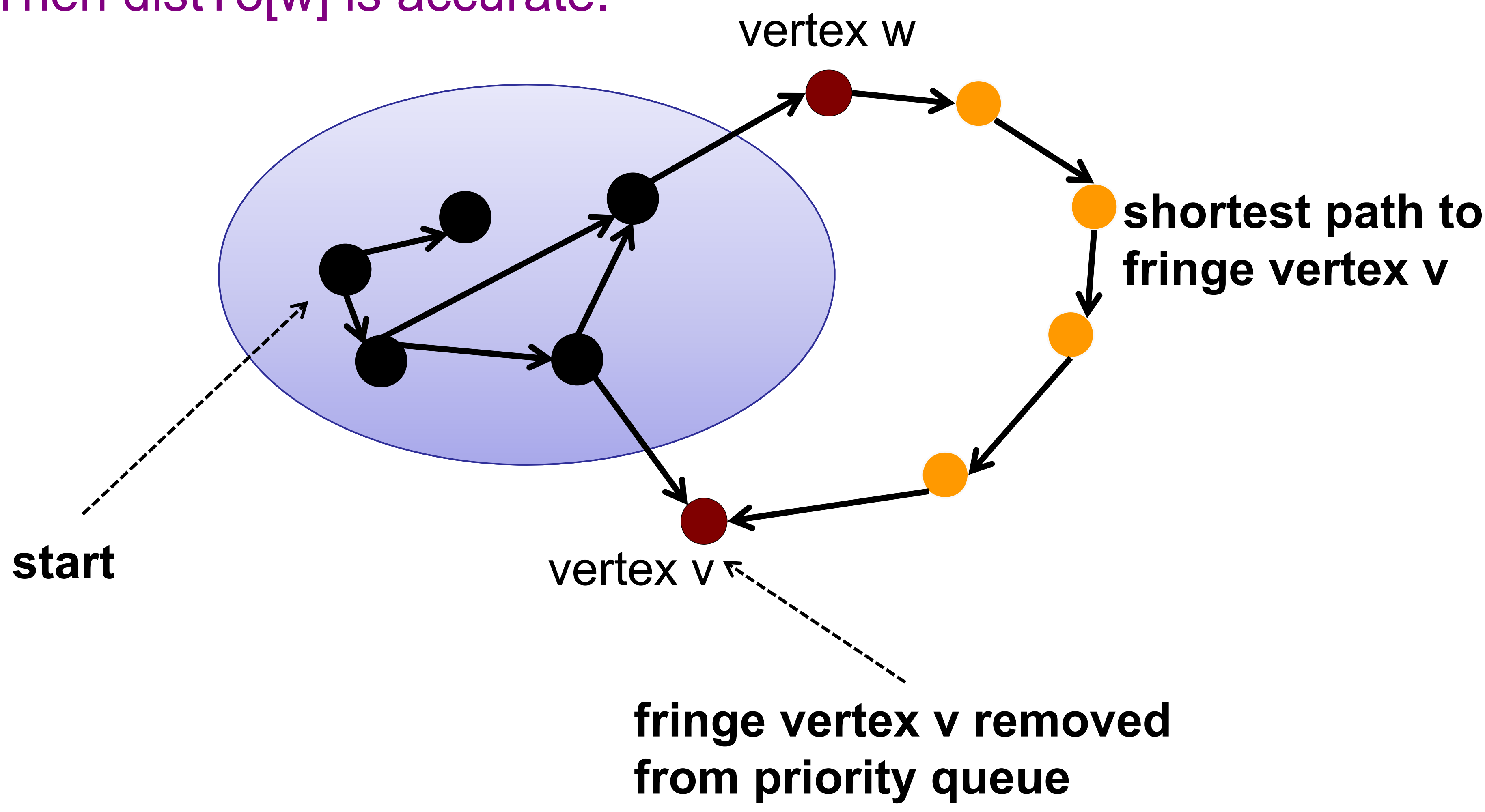
Assume not: *fringe vertex has shorter path*



# Dijkstra's Algorithm

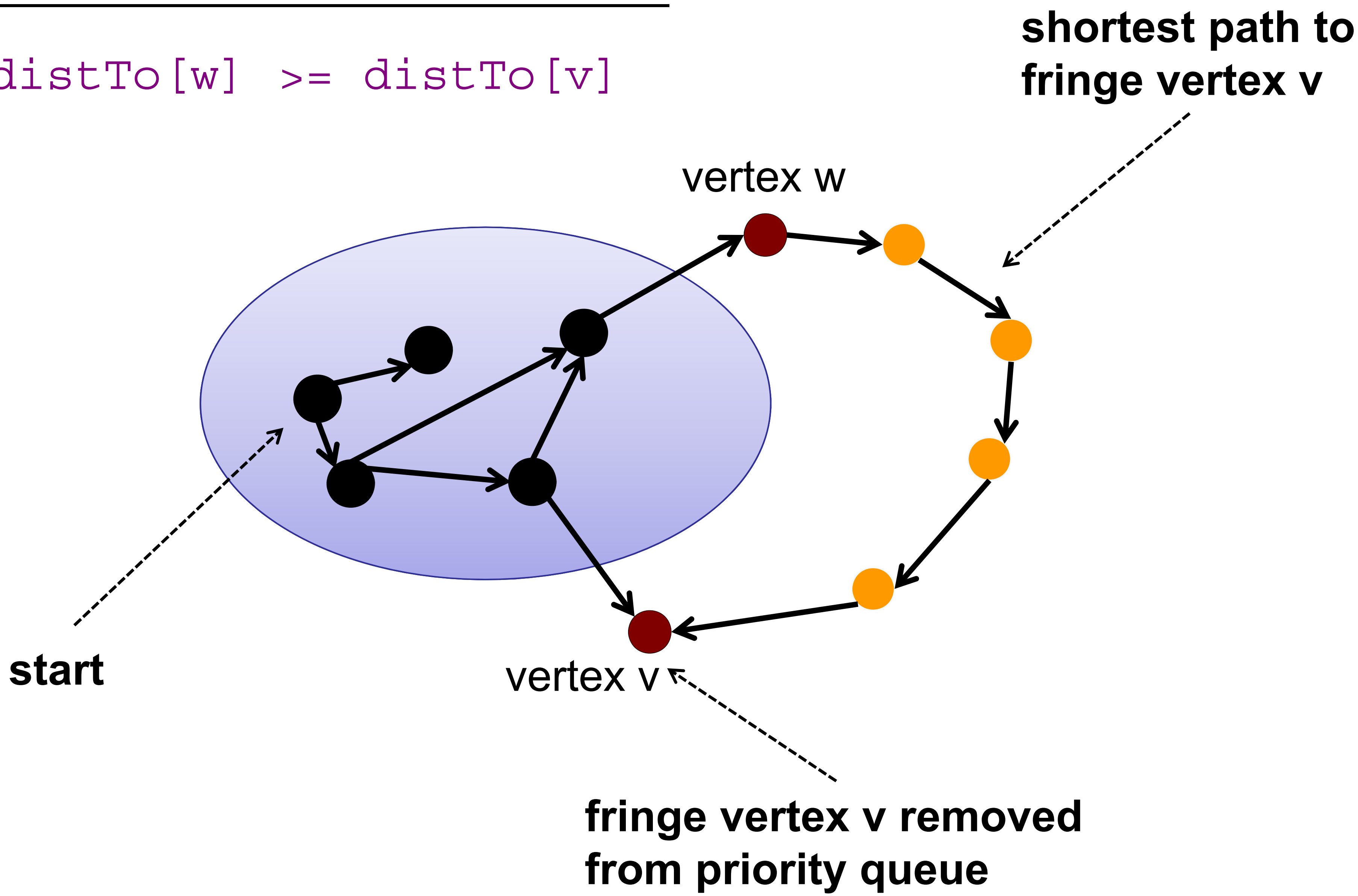
If P is shortest path to v, then prefix of P is shortest path to w.

Then distTo[w] is accurate.



# Dijkstra's Algorithm

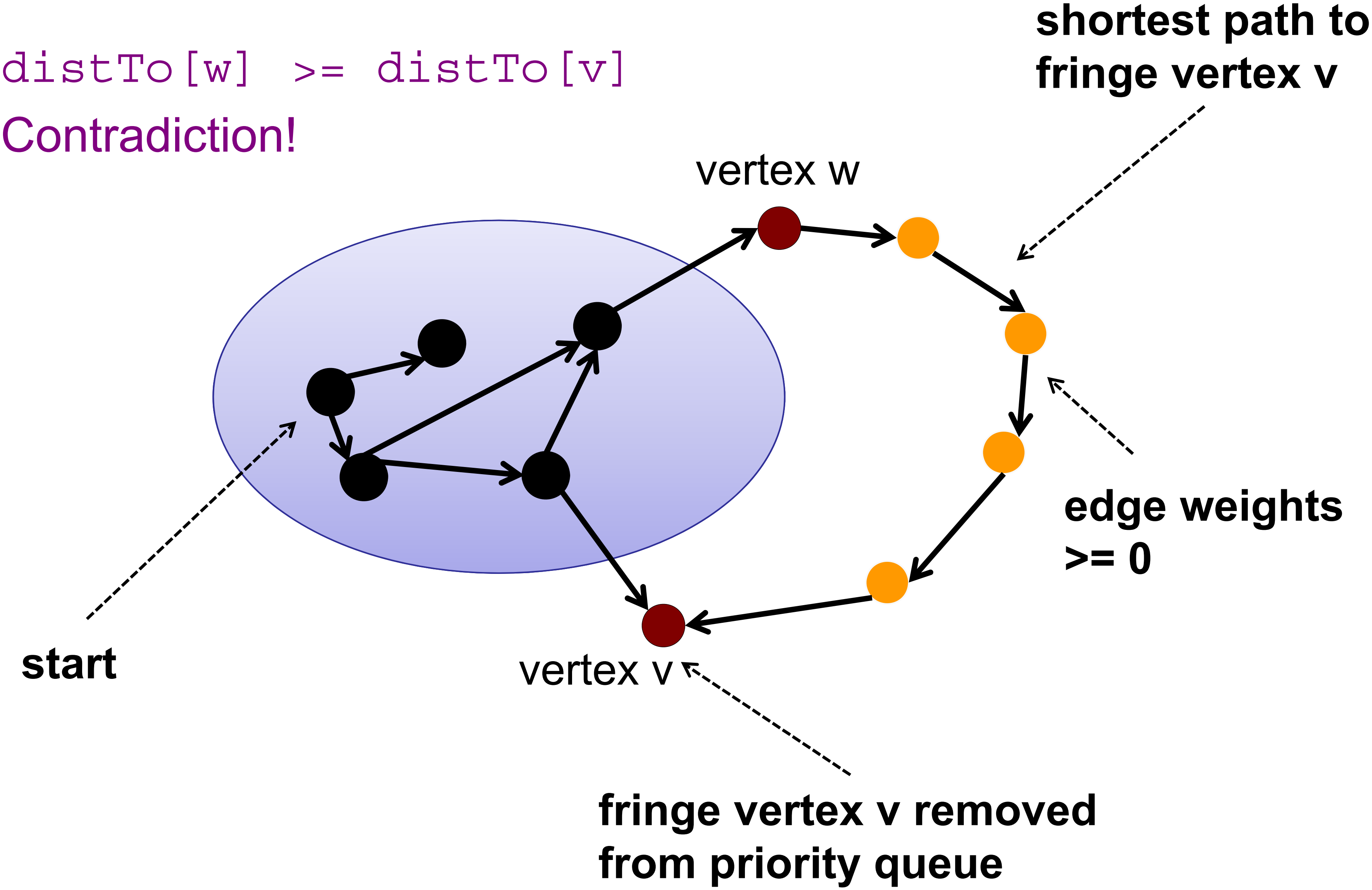
`distTo[w] >= distTo[v]`



# Dijkstra's Algorithm

`distTo[w] >= distTo[v]`

Contradiction!



# Dijkstra’s Algorithm

Proof by induction:

- Every “finished” vertex has correct estimate.
- Initially: only “finished” vertex is start.

- Inductive step:
  - Remove vertex from priority queue.
  - Relax its edges.
  - Add it to finished.
  - **Claim: it has a correct estimate.**

# Dijkstra’s Algorithm

```
relax(Edge e) {
    int v = e.from();
    int w = e.to();
    double weight = e.weight();
    if (distTo[w] > distTo[v] + weight) {
        distTo[w] = distTo[v] + weight;
        parent[w] = v;
        if (pq.contains(w))
            pq.decreaseKey(w, distTo[w]);
        else
            pq.insert(w, distTo[w]);
    }
}
```

Extending a path does not make it shorter!

# Dijkstra's Algorithm

Analysis:

- insert / deleteMin: |V| times each
  - Each node is added to the priority queue **once**.
- decreaseKey: |E| times
  - Each edge is relaxed once.
- Priority queue operations: O(log V)
- Total: O((V+E)log V) = O(E log V)

## Source-to-Destination Dijkstra

Can we stop as soon as we dequeue the destination?

1. ✓ Yes.
2. Only if the graph is sparse.
3. No.

# Dijkstra's Algorithm

Source-to-Destination:

- What if you stop the first time you dequeue the destination?

- Recall:
  - a vertex is "finished" when it is dequeued
  - if the destination is finished, then stop

# Dijkstra Summary

Basic idea:

- Maintain distance estimates.
- Repeat:
  - Find unfinished vertex with smallest estimate.
  - Relax all outgoing edges.
  - Mark vertex finished.

- O(E log V) time (with AVL tree).

# Dijkstra's Performance

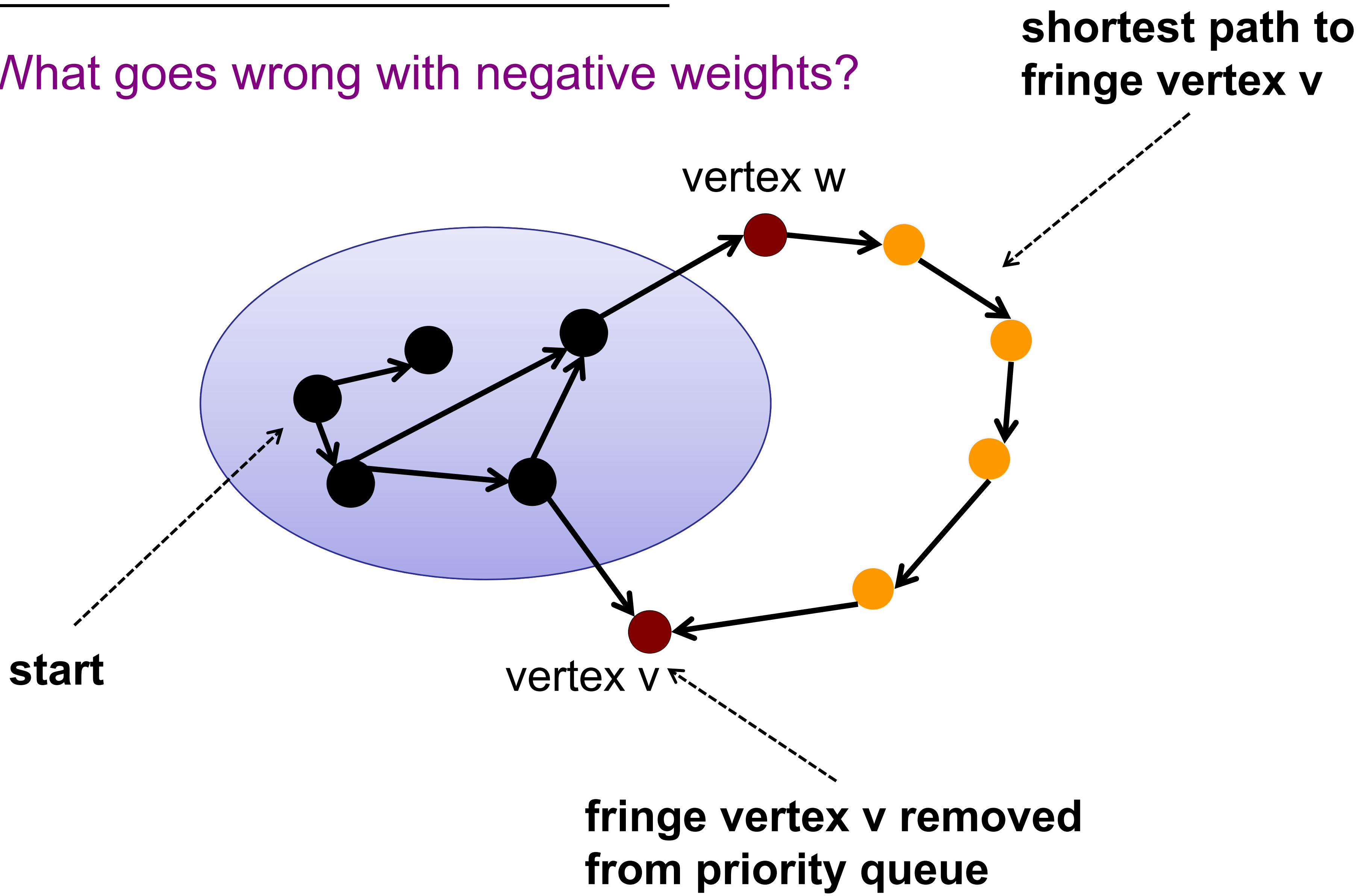
| PQ Implementation | insert | deleteMin | decreaseKey | Total |
|---|---|---|---|---|
| Array | 1 | V | 1 | **$O(V^2)$** |
| AVL Tree | log V | log V | log V | **O(E log V)** |
| d-way Heap | $d\log_d V$ | $d\log_d V$ | $\log_d V$ | **$O(E\log_{E/V} V)$** |
| Fibonacci Heap | 1 | log V | 1 | **O(E + V log V)** |

# Dijkstra Summary

Edges with negative weights?

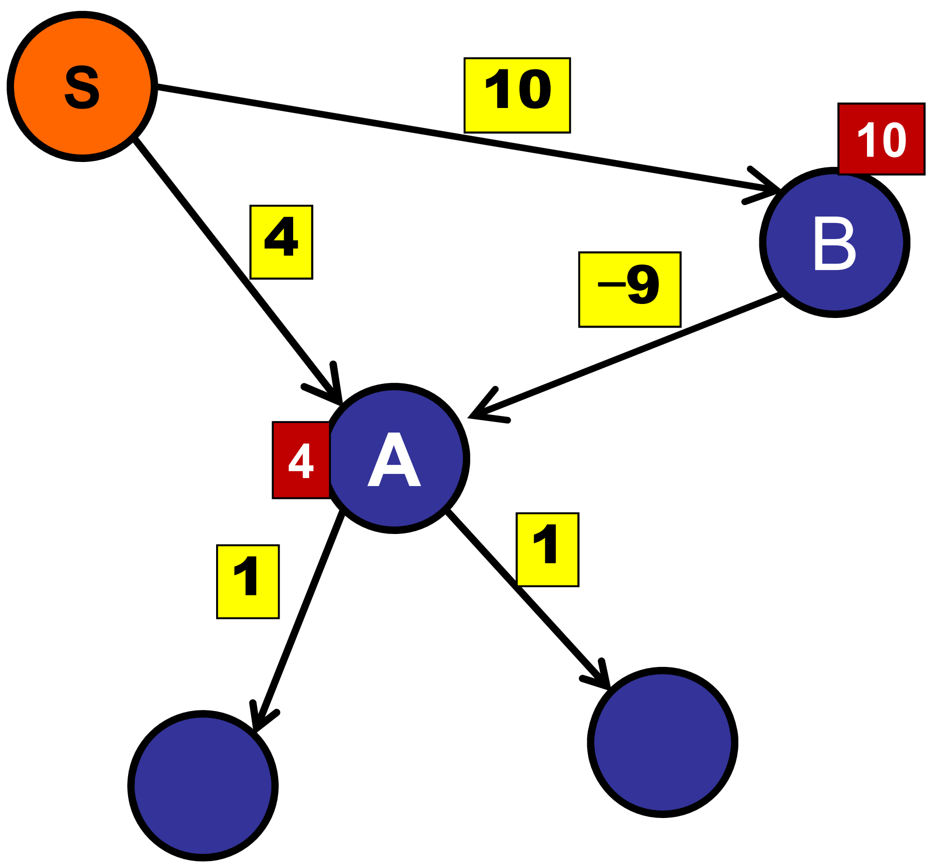
# Dijkstra's Algorithm

What goes wrong with negative weights?



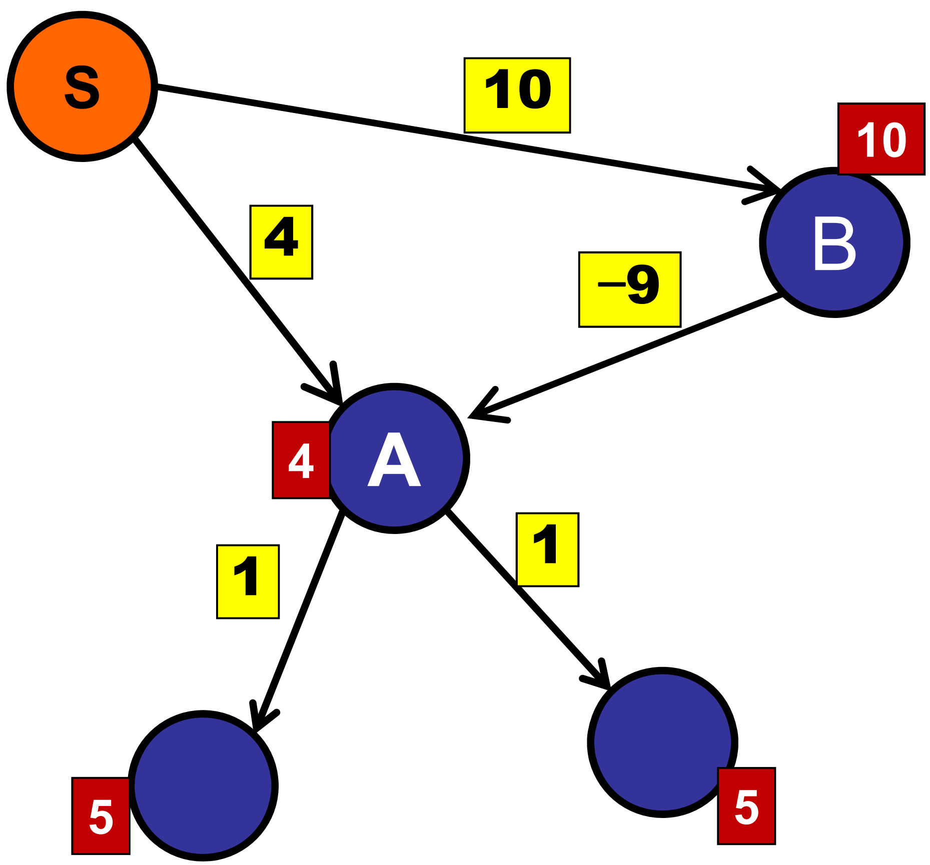
# Dijkstra's Algorithm

Edges with negative weights?



# Dijkstra's Algorithm

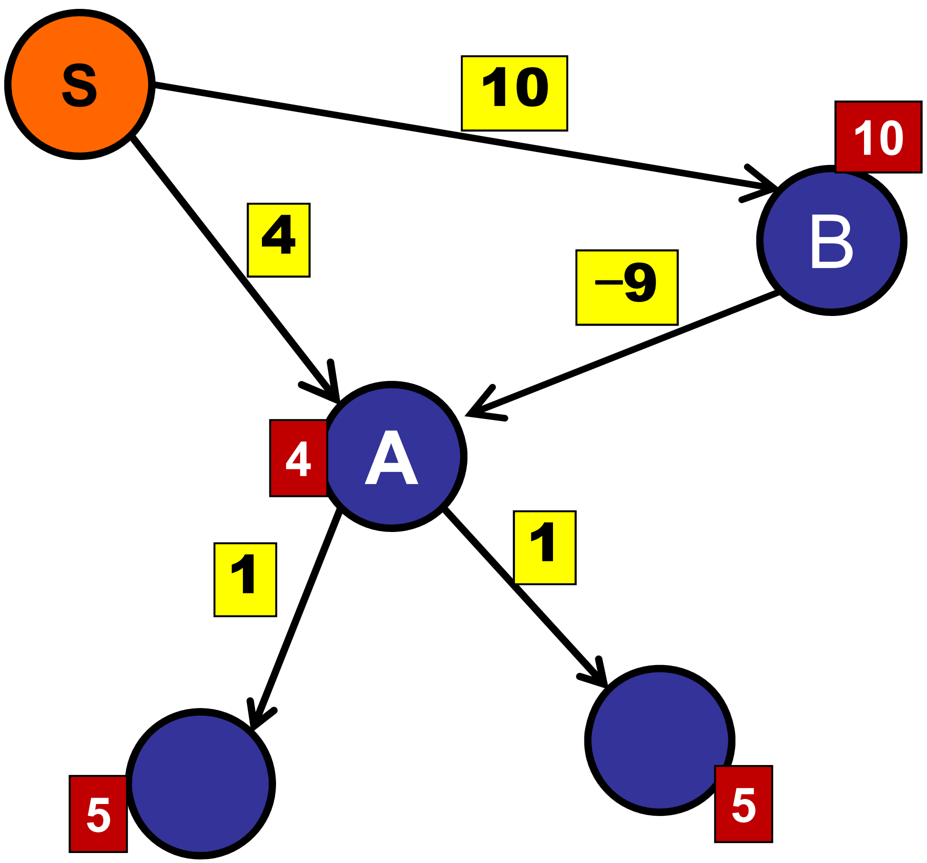
Edges with negative weights?




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

# Dijkstra's Algorithm

## Edges with negative weights?




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

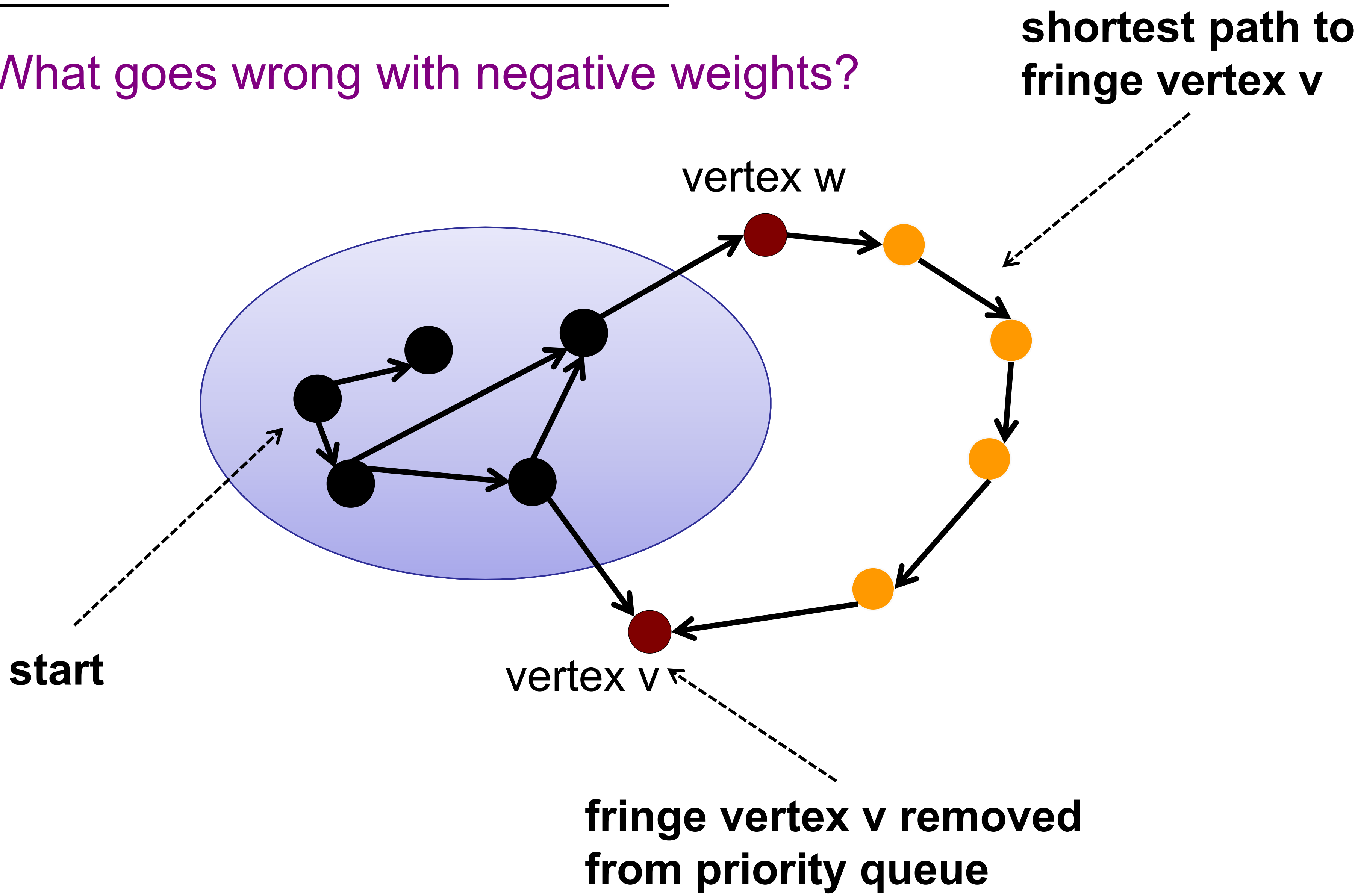
…

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

Oops: We need to update A.

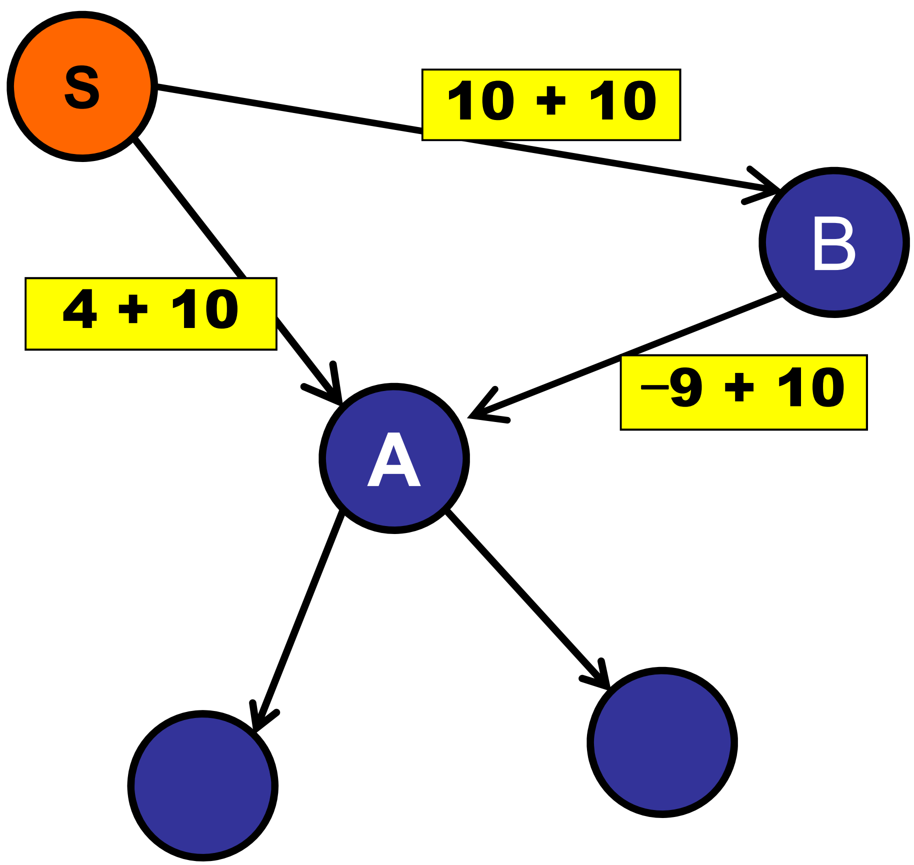
# Dijkstra's Algorithm

What goes wrong with negative weights?



# Dijkstra’s Algorithm

Can we reweight? e.g.: weight += 10

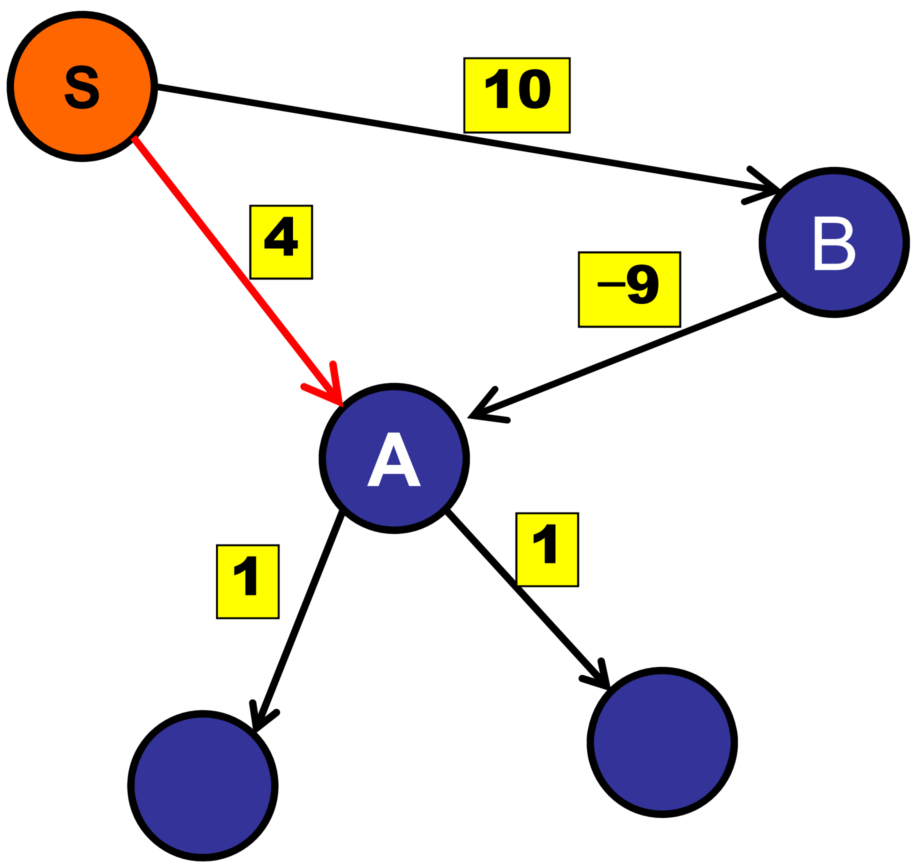


# Can we reweight the graph?

1. Yes.
2. Only if there are no negative weight cycles.
3. No.

# Dijkstra's Algorithm

Can we reweight?

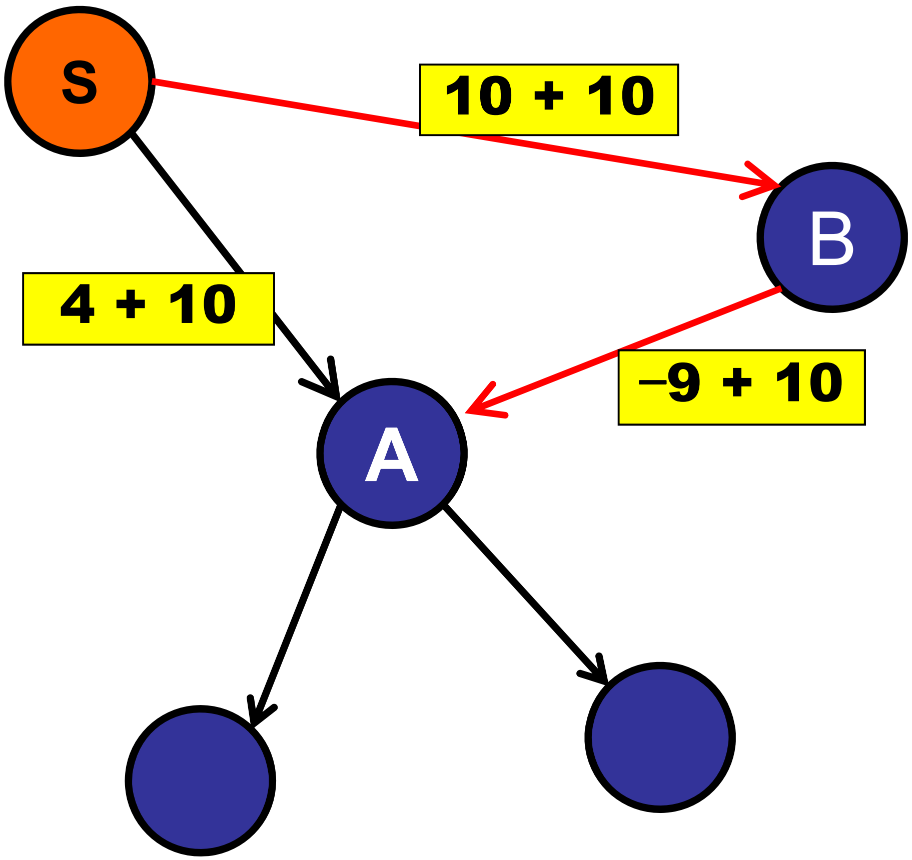



Path S-B-A: 1

Path S-A: 4

# Dijkstra's Algorithm

Can we reweight?




Path S-B-A: 21

Path S-A: 14

# Dijkstra Summary

Basic idea:

- Maintain distance estimates.
- Repeat:
  - Find unfinished vertex with smallest estimate.
  - Relax all outgoing edges.
  - Mark vertex finished.
- O(E log V) time (with AVL tree Priority Queue).
- No negative weight edges!

# Dijkstra Comparison

Same algorithm:

- Maintain a set of explored vertices.
- Add vertices to the explored set by following edges that go from a vertex in the explored set to a vertex outside the explored set.

  - BFS: Take edge from vertex that was discovered **least** recently.
  - DFS: Take edge from vertex that was discovered **most** recently.
  - Dijkstra's: Take edge from vertex that is **closest** to source.

# Dijkstra Comparison

Same algorithm:

- Maintain a set of explored vertices.
- Add vertices to the explored set by following edges that go from a vertex in the explored set to a vertex outside the explored set.

  - BFS: Use queue.
  - DFS: Use stack.
  - Dijkstra's: Use priority queue.

# Today

## Single-Source Shortest Paths

- Weighted, Directed Graphs
- Bellman-Ford: simple, general
- Dijkstra: faster, only non-negative weights