CS2040S
Data Structures and Algorithms

Shortest Paths!

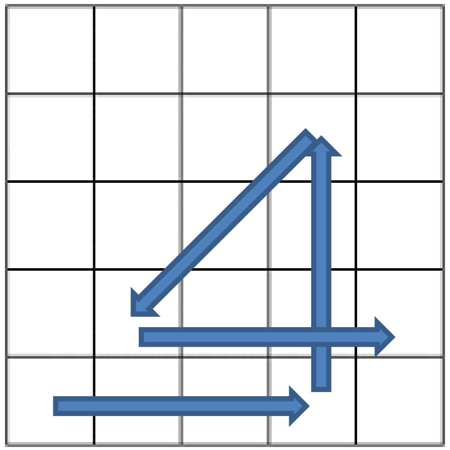
(Try writing a program to solve this!)

Puzzle of the week:

- 5 x 5 grid
- Choose a starting square
- Move: 3 cells vertically or horizontally OR
- Move: 2 cells diagonally.
- Cannot visit same cell twice.
- Cannot exit grid

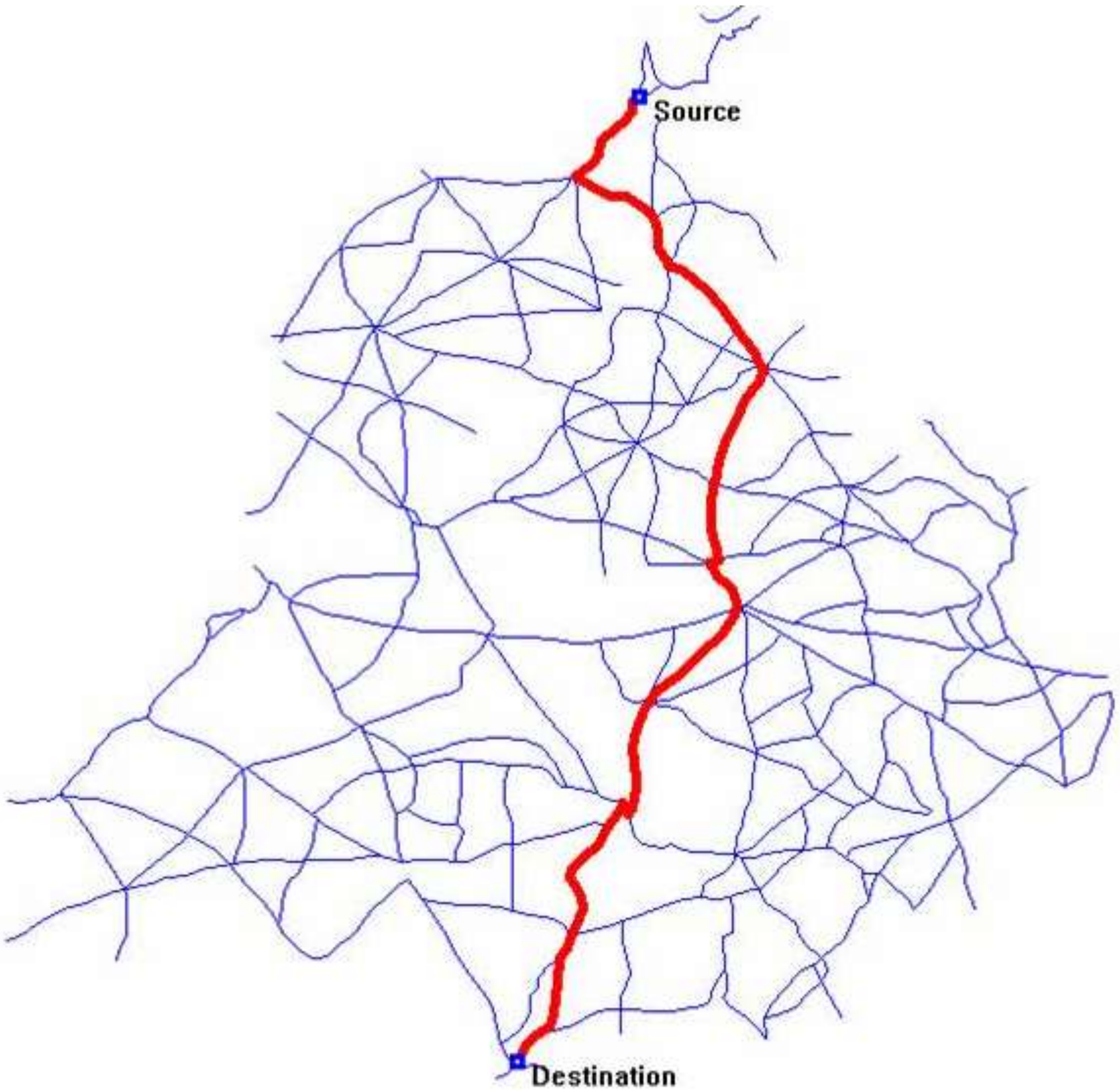
To win: visit all cells.

Example:



** What's the *worst* you can do?

SHORTEST PATHS



What is a directed graph?

Graph consists of two types of elements:

Nodes (or vertices)

- At least one.

Edges (or arcs)

- Each edge connects two nodes in the graph
- Each edge is unique.
- Each edge is **directed**.

What is a directed graph?

Graph $G = \langle V, E \rangle$

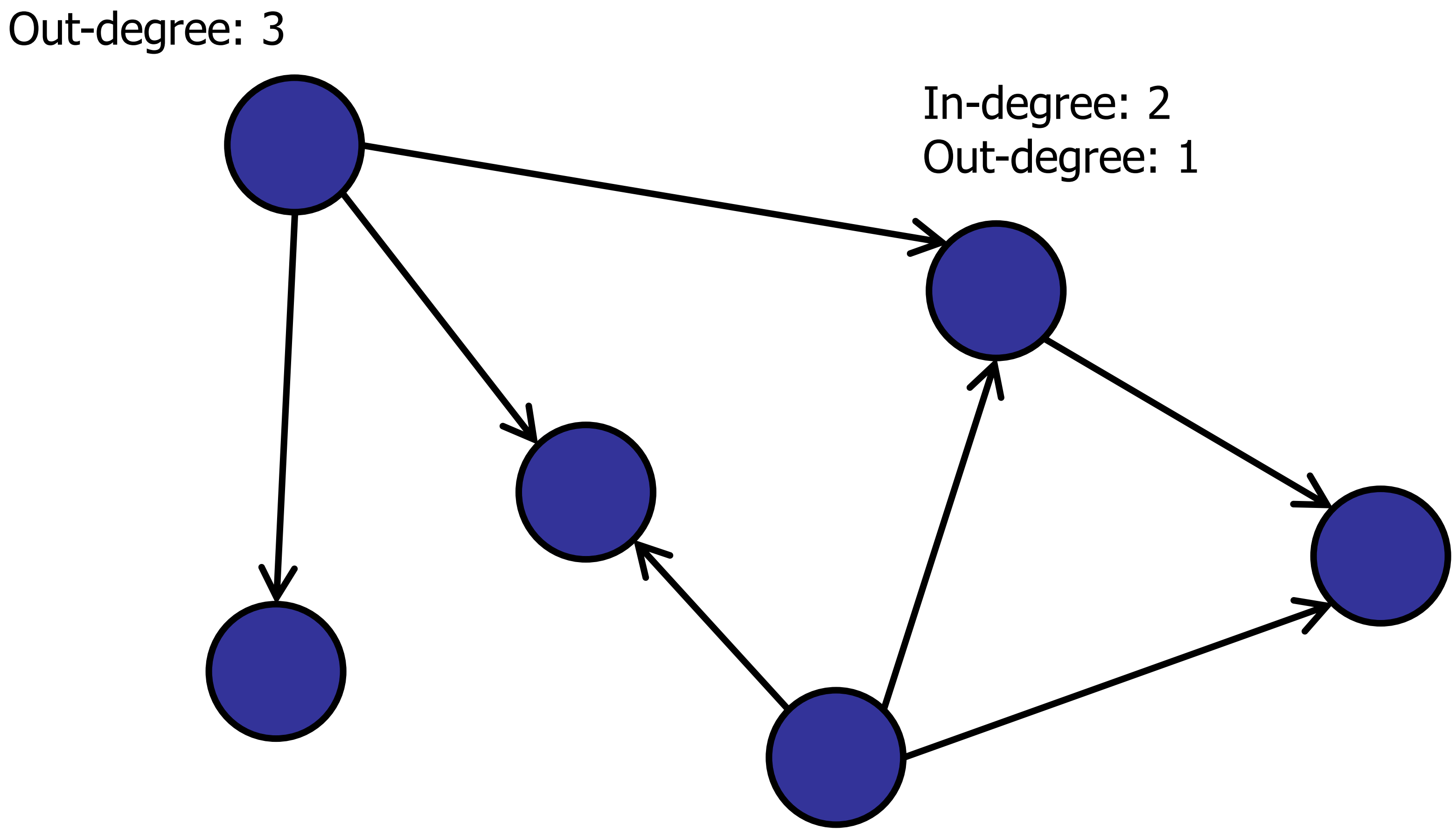
- V is a set of nodes
  - At least one: $|V| > 0$.
- E is a set of edges:
  - $E \subseteq \{ (v,w) : (v \in V), (w \in V) \}$
  - $e = (v,w)$
  - For all $e_1, e_2 \in E : e_1 \neq e_2$

Order matters!

What is a directed graph?

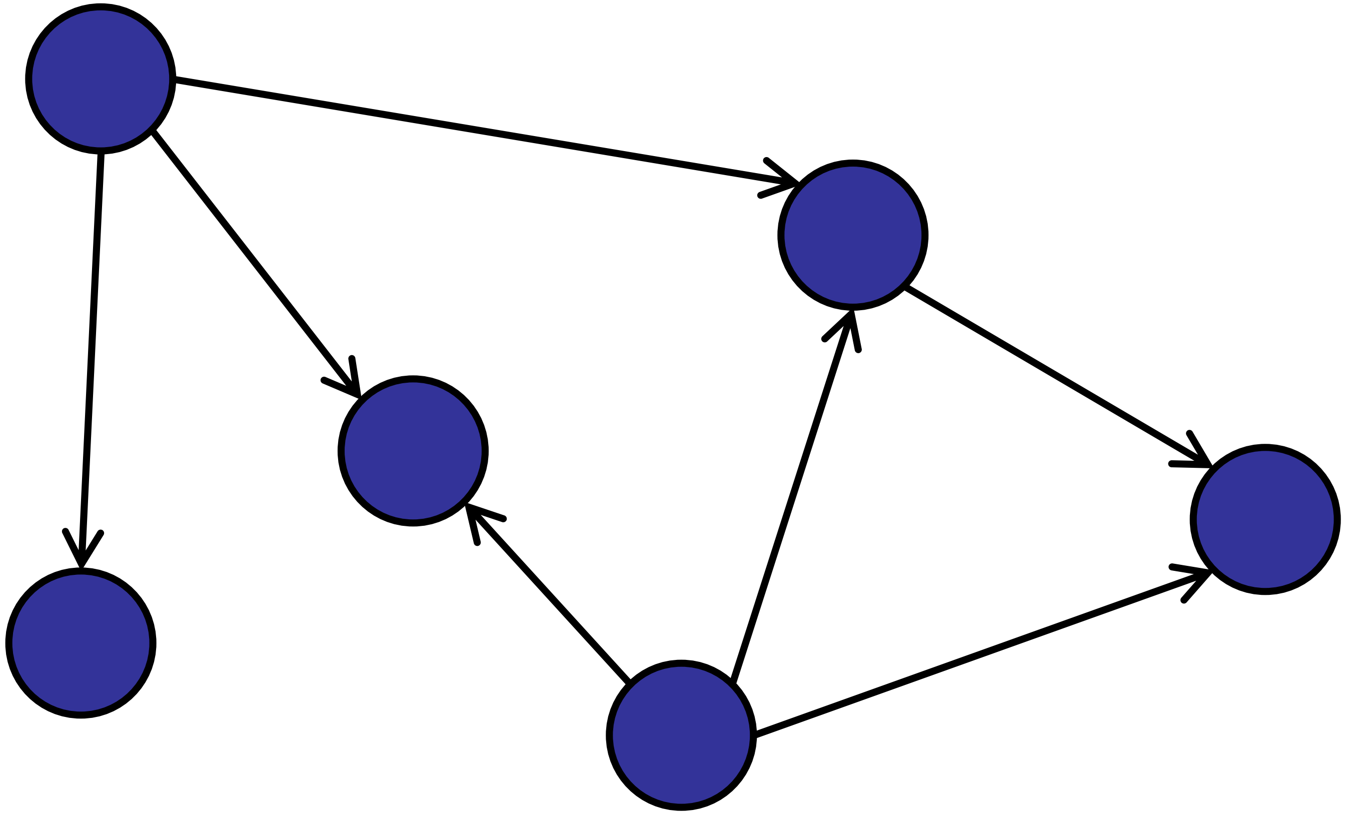
In-degree: number of incoming edges

Out-degree: number of outgoing edges



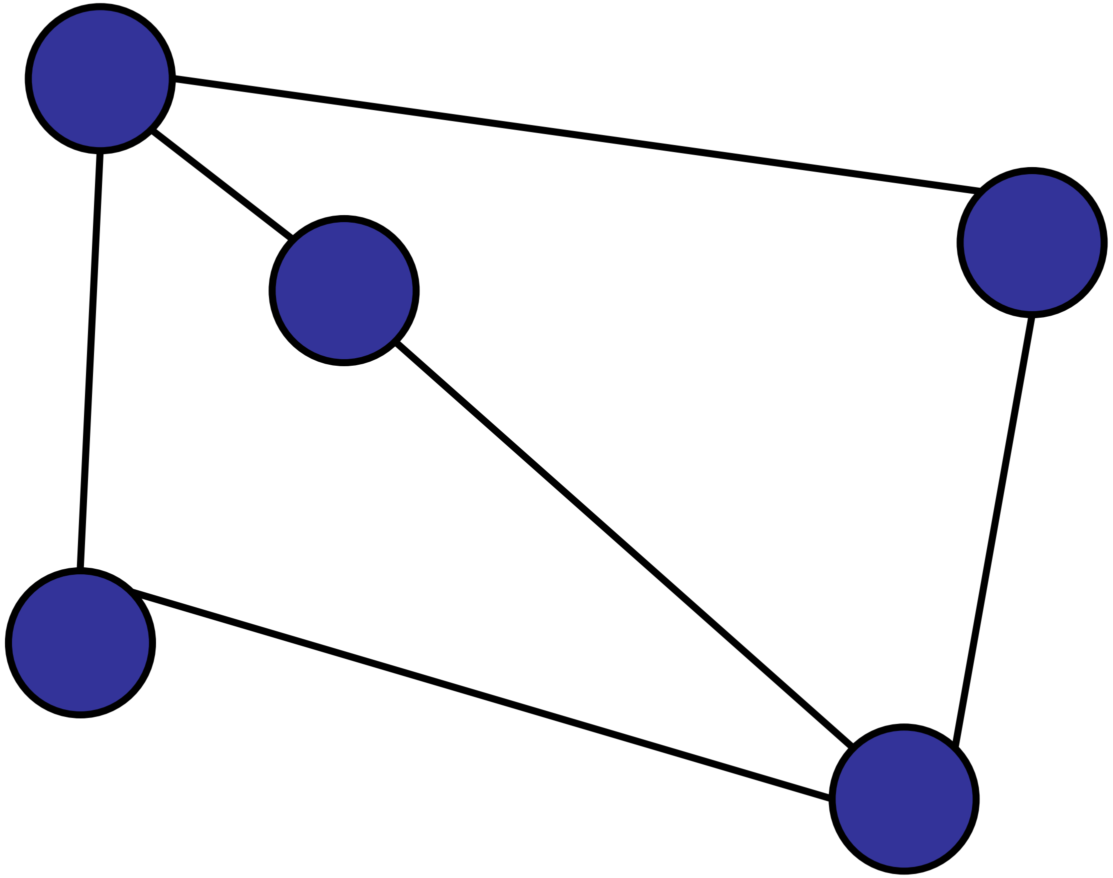
Is it a directed graph?

1. Yes
2. No.



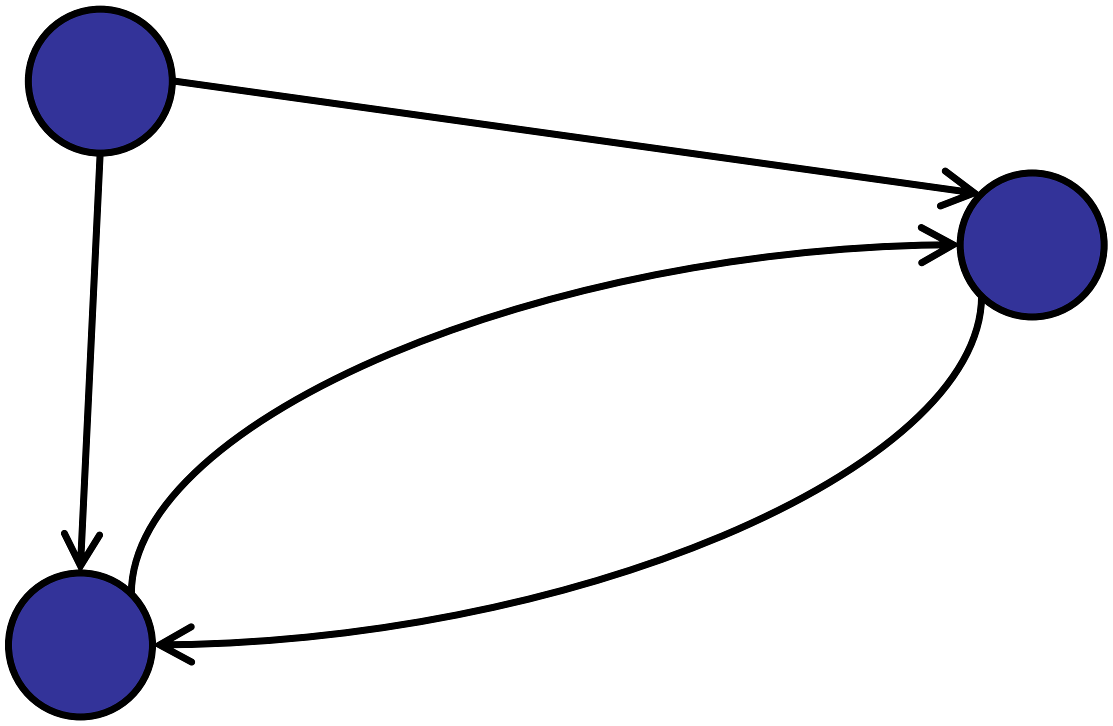
Is it a directed graph?

1. Yes
2. No.



Is it a directed graph?

1. Yes
2. No.



Representing a (Directed) Graph

Adjacency List:

- Array of nodes
- Each node maintains a list of neighbors
- Space: $O(V + E)$

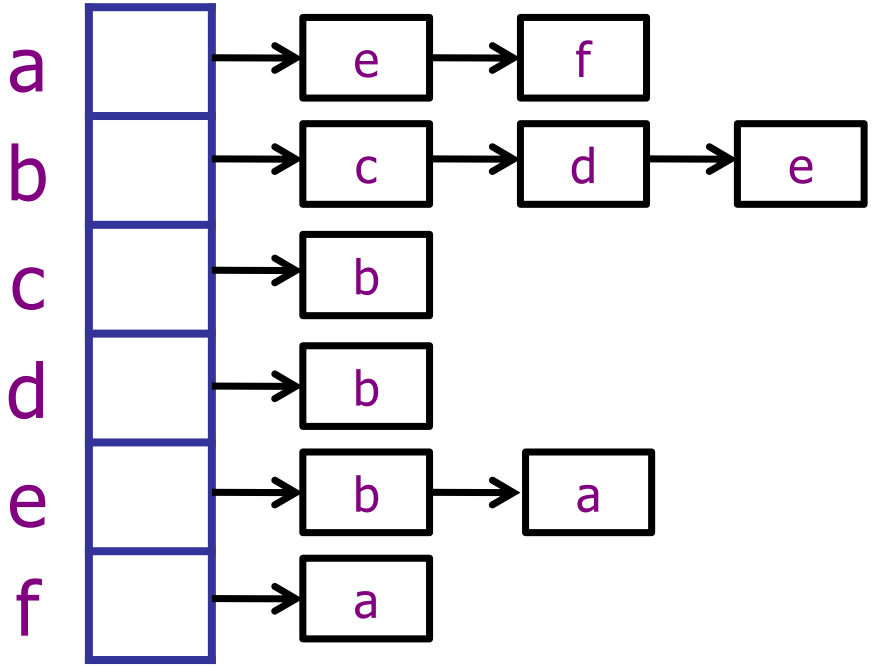
Adjacency Matrix:

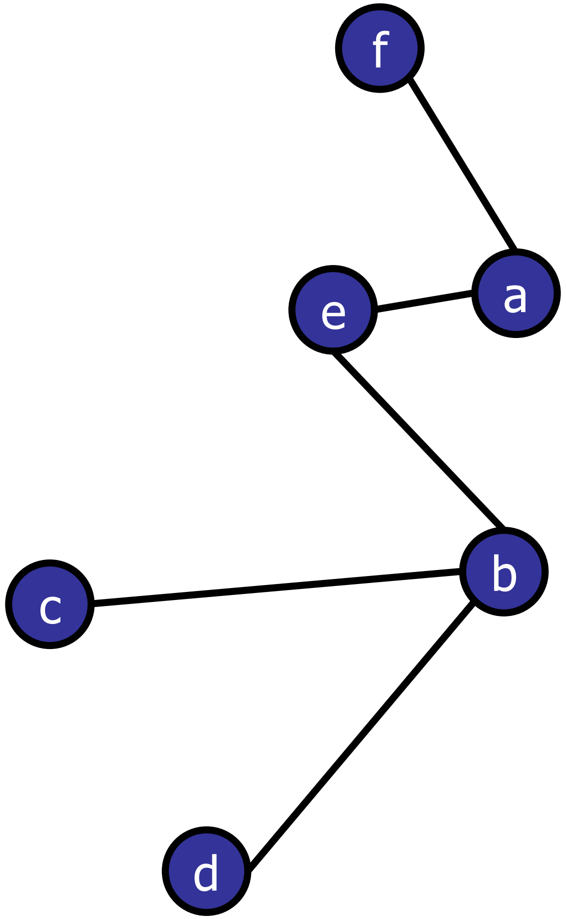
- Matrix $A[v,w]$ represents edge (v,w)
- Space: $O(V^2)$

Adjacency List

Graph consists of:

- Nodes: stored in an array
- Edges: linked list per node

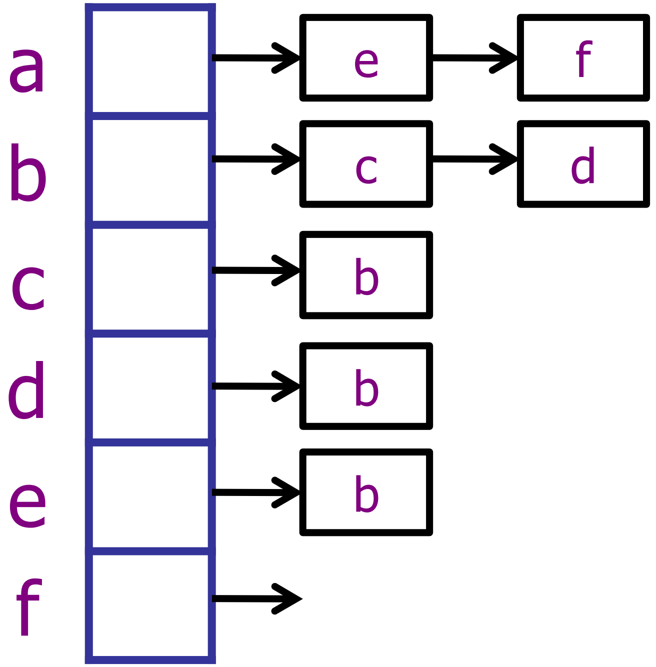


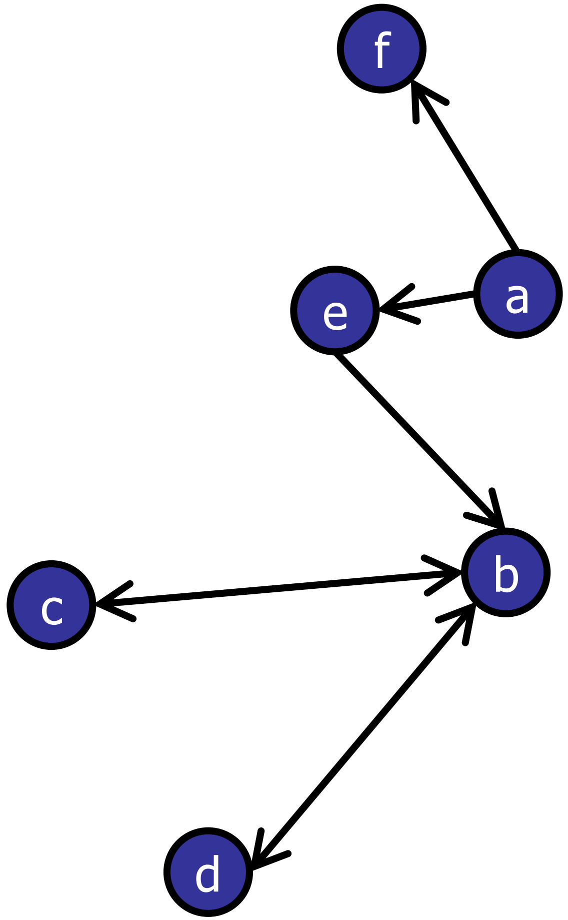


Adjacency List

Directed Graph consists of:

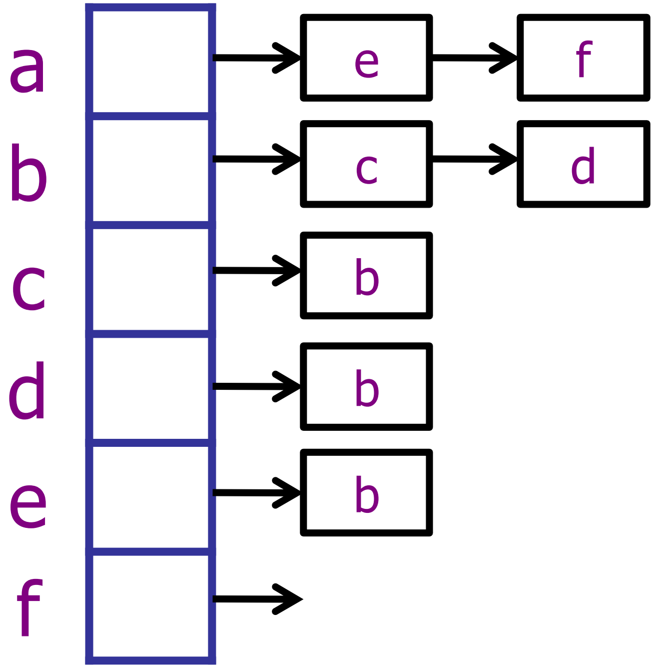
- Nodes: stored in an array
- **Outgoing** Edges: linked list per node





Adjacency List in Java

```
class NeighborList extends ArrayList<Integer> {
}

class Node {
  int key;
  NeighborList nbrs;
}

class Graph {
  Node[] nodeList;


}
```



Representing a (Directed) Graph

Adjacency List:

- Array of nodes
- Each node maintains a list of neighbors
- Space: $O(V + E)$

Adjacency Matrix:

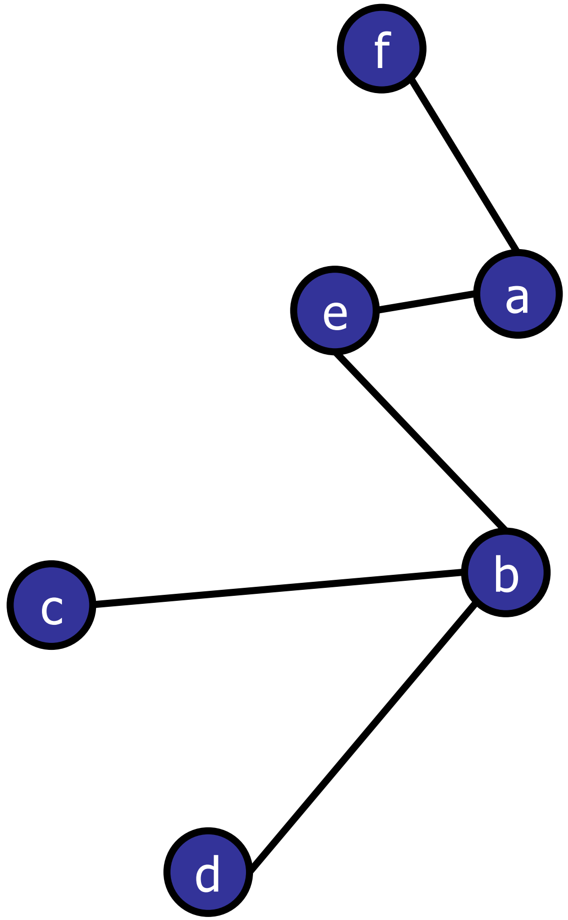
- Matrix $A[v,w]$ represents edge (v,w)
- Space: $O(V^2)$

Adjacency Matrix

Graph consists of:

- Nodes
- Edges = pairs of nodes

	a	b	c	d	e	f
a	0	0	0	0	**1**	**1**
b	0	0	**1**	**1**	**1**	0
c	0	**1**	0	0	0	0
d	0	**1**	0	0	0	0
e	**1**	**1**	0	0	0	0
f	**1**	0	0	0	0	0

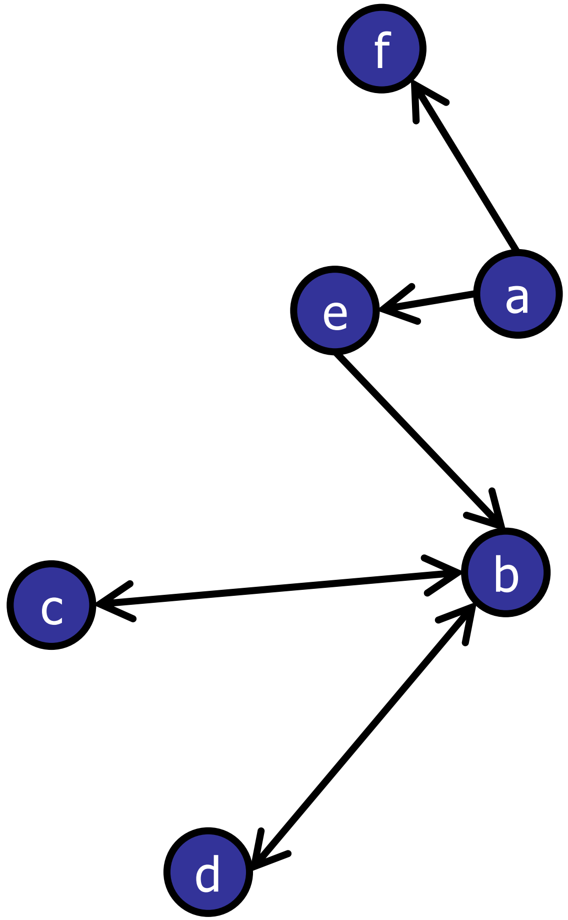


Adjacency Matrix

Directed Graph consists of:

- Nodes
- Edges = pairs of nodes

	a	b	c	d	e	f
a	0	0	0	0	**1**	**1**
b	0	0	**1**	**1**	**0**	0
c	0	**1**	0	0	0	0
d	0	**1**	0	0	0	0
e	**0**	**1**	0	0	0	0
f	**0**	0	0	0	0	0



Adjacency Matrix

Graph represented as:

$A[v][w] = 1$ iff $(v,w) \in E$

	a	b	c	d	e	f
a	0	0	0	0	**1**	**1**
b	0	0	**1**	**1**	**0**	0
c	0	**1**	0	0	0	0
d	0	**1**	0	0	0	0
e	**0**	**1**	0	0	0	0
f	**0**	0	0	0	0	0

Searching a (Directed) Graph

Breadth-First Search:

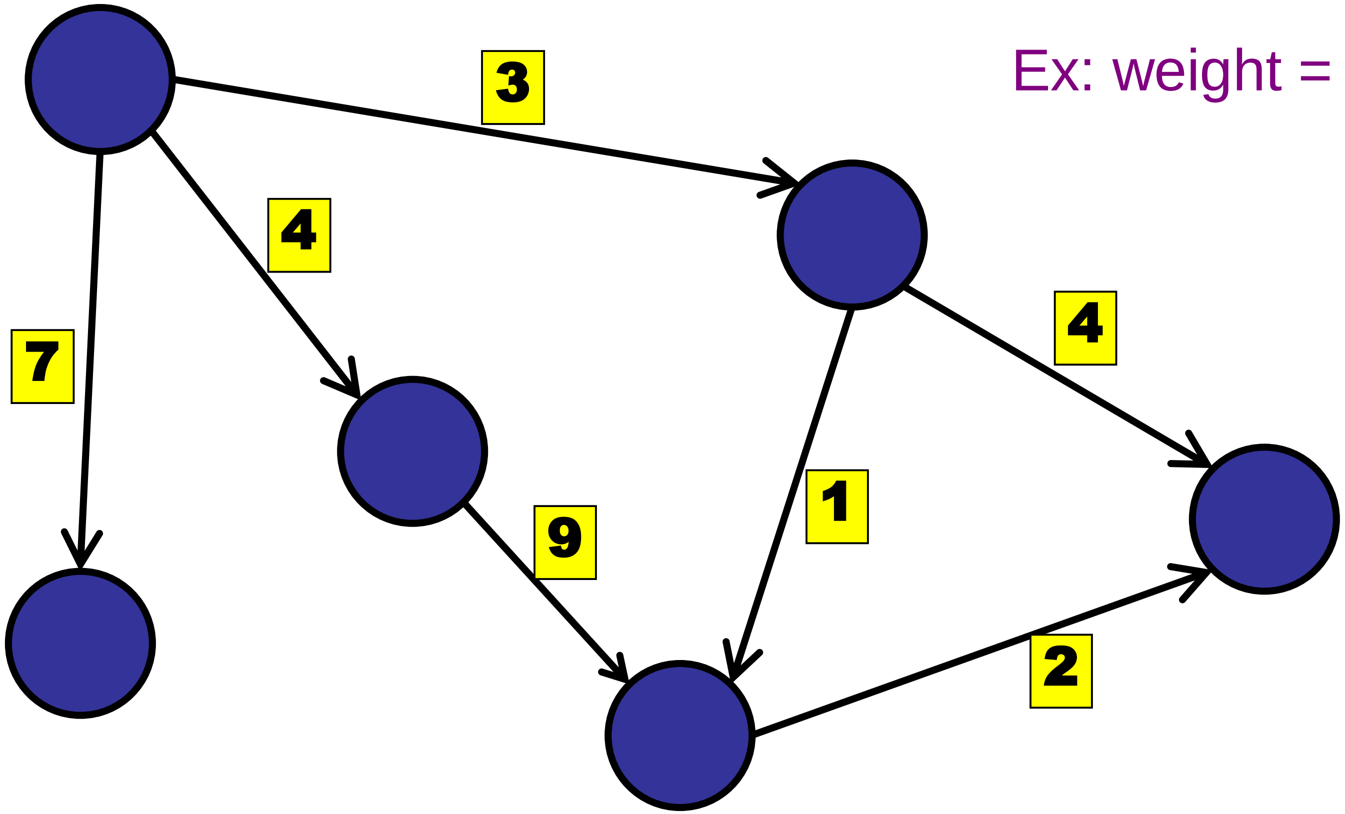
- Search level-by-level
- Follow outgoing edges
- Ignore incoming edges

Depth-First Search:

- Search recursively
- Follow outgoing edges
- Backtrack (through incoming edges)

Weighted Graphs

Edge weights: w(e) : E ➔ R

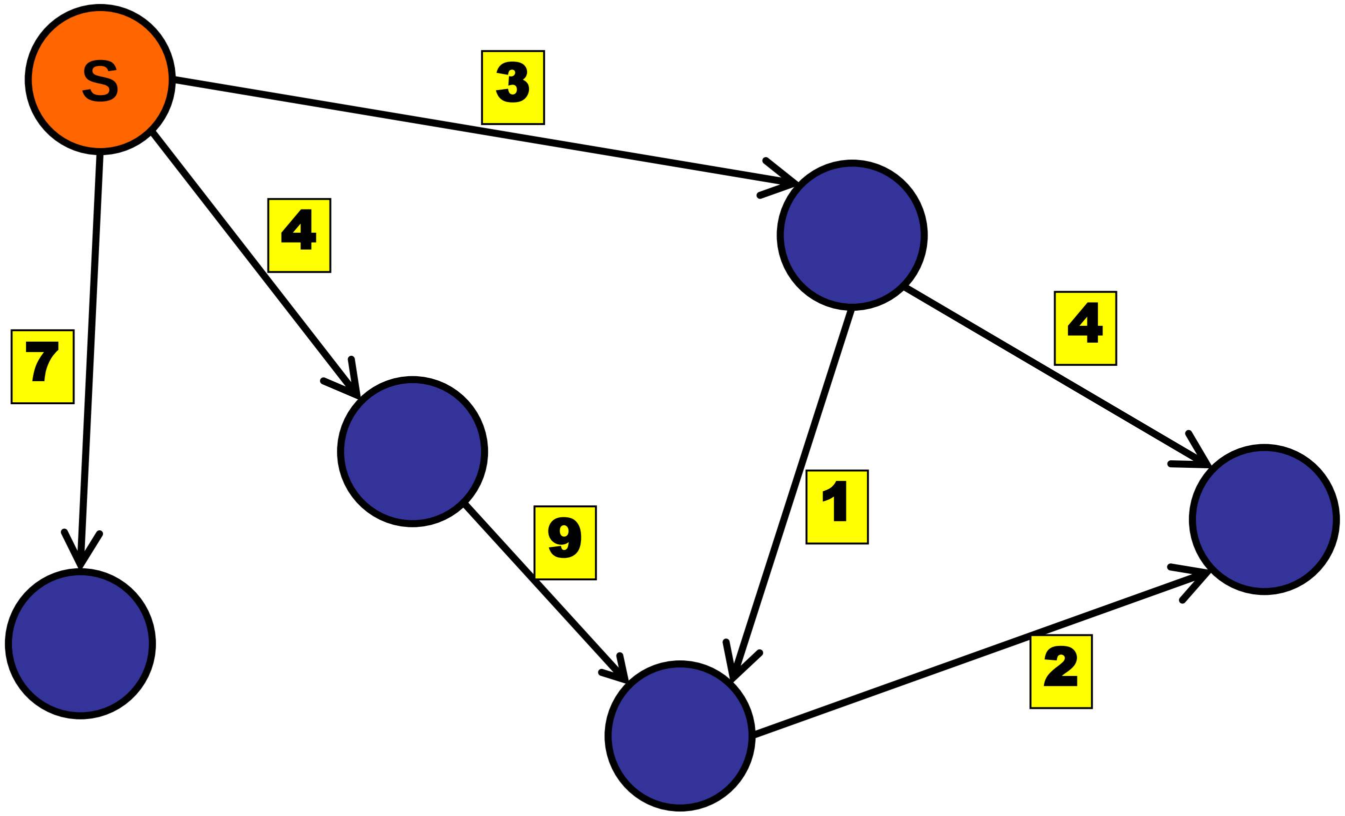


Ex: weight = distance

Adjacency list: stores weights with edge in NbrList

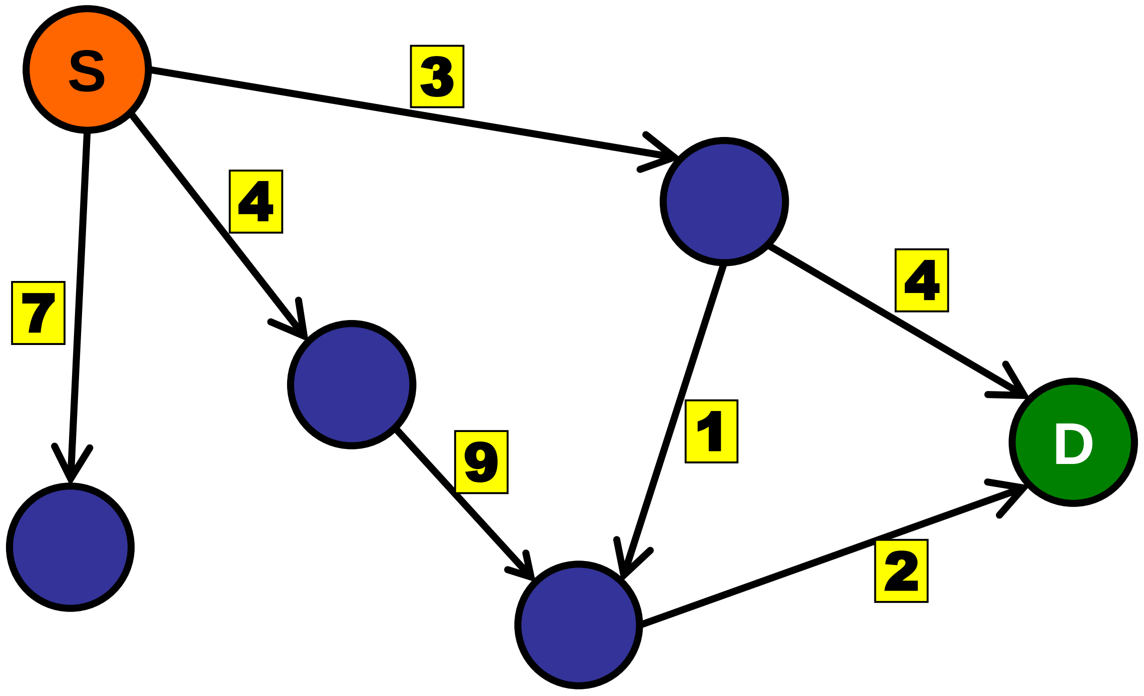
Shortest Paths

Distance from source?



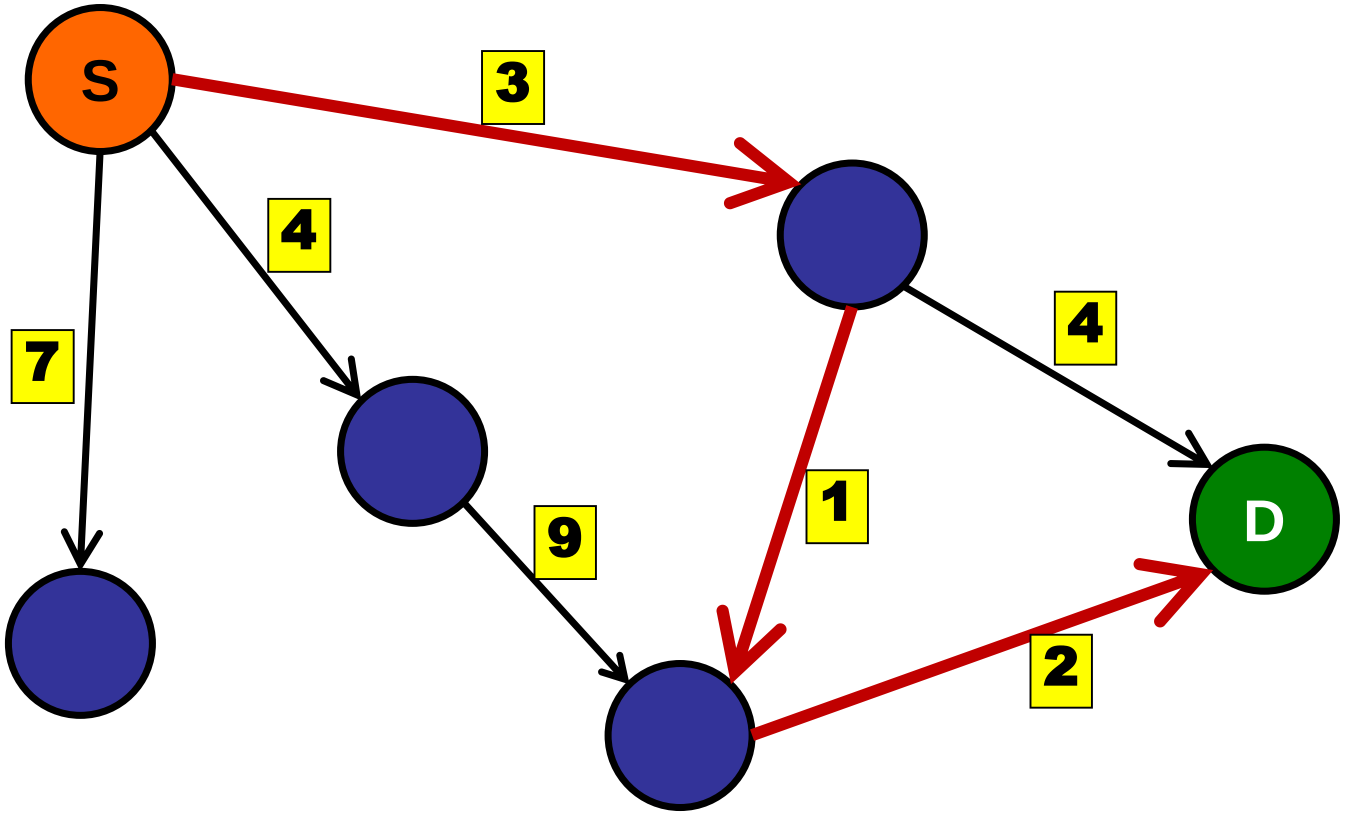
What is the distance from S to D?

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



Shortest Paths

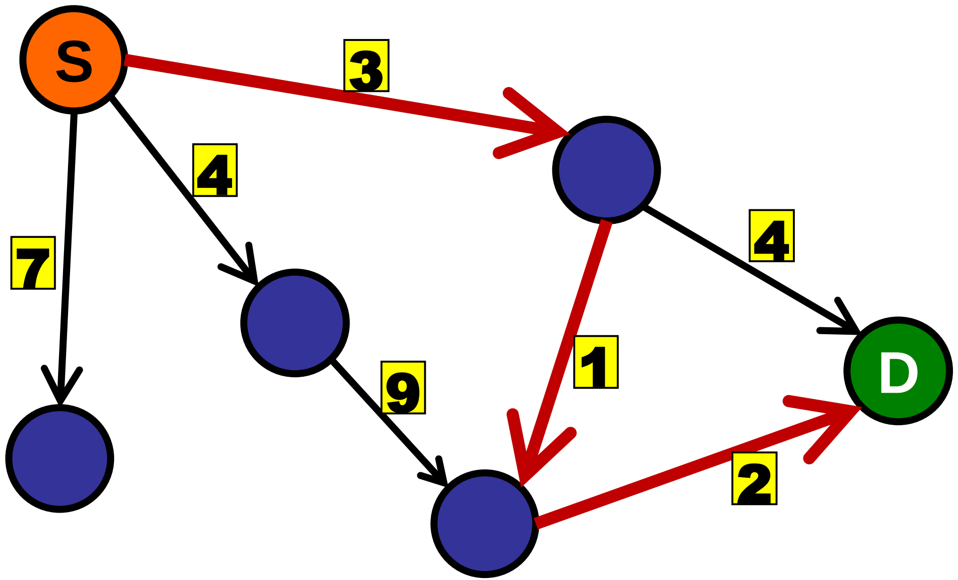
Distance from source?



Shortest Paths

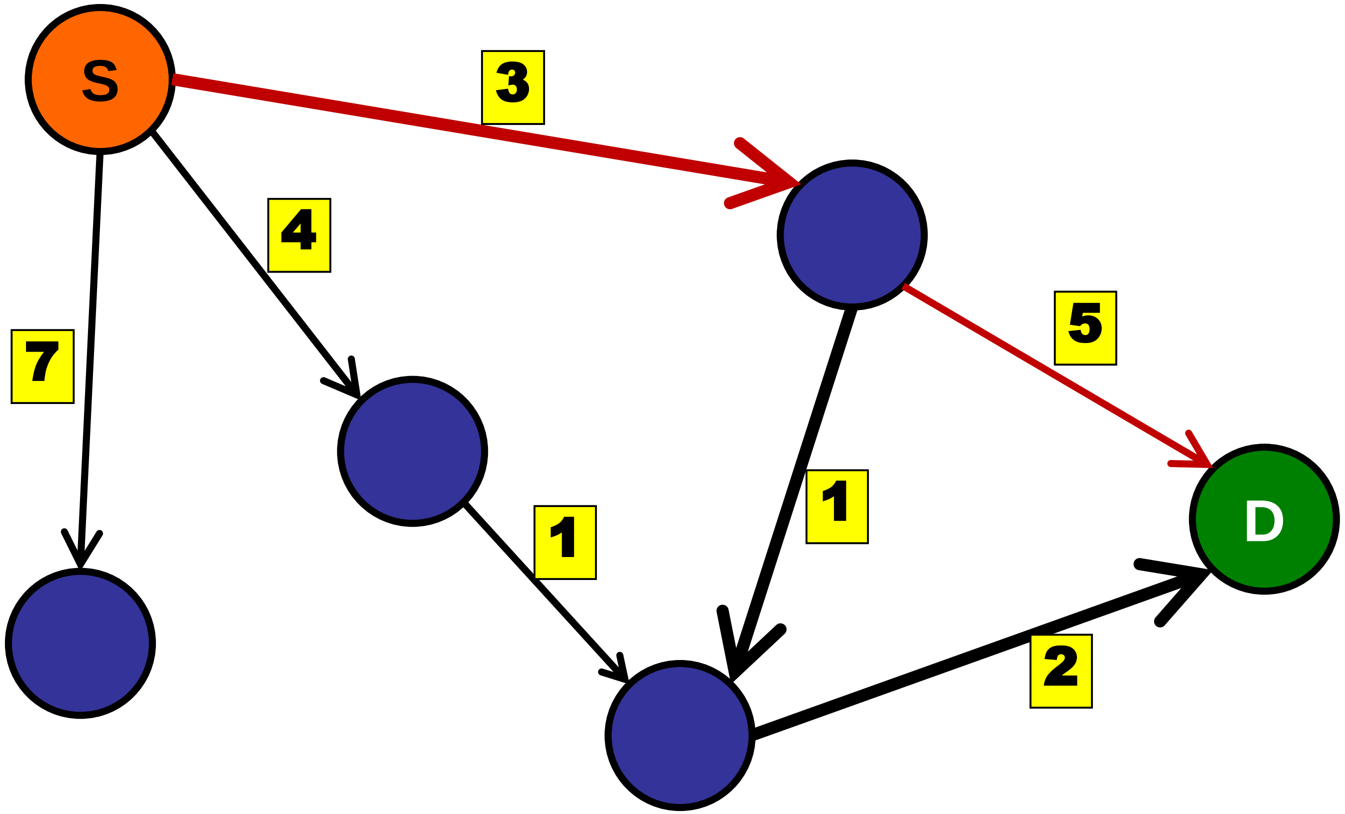
Questions:

- 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.



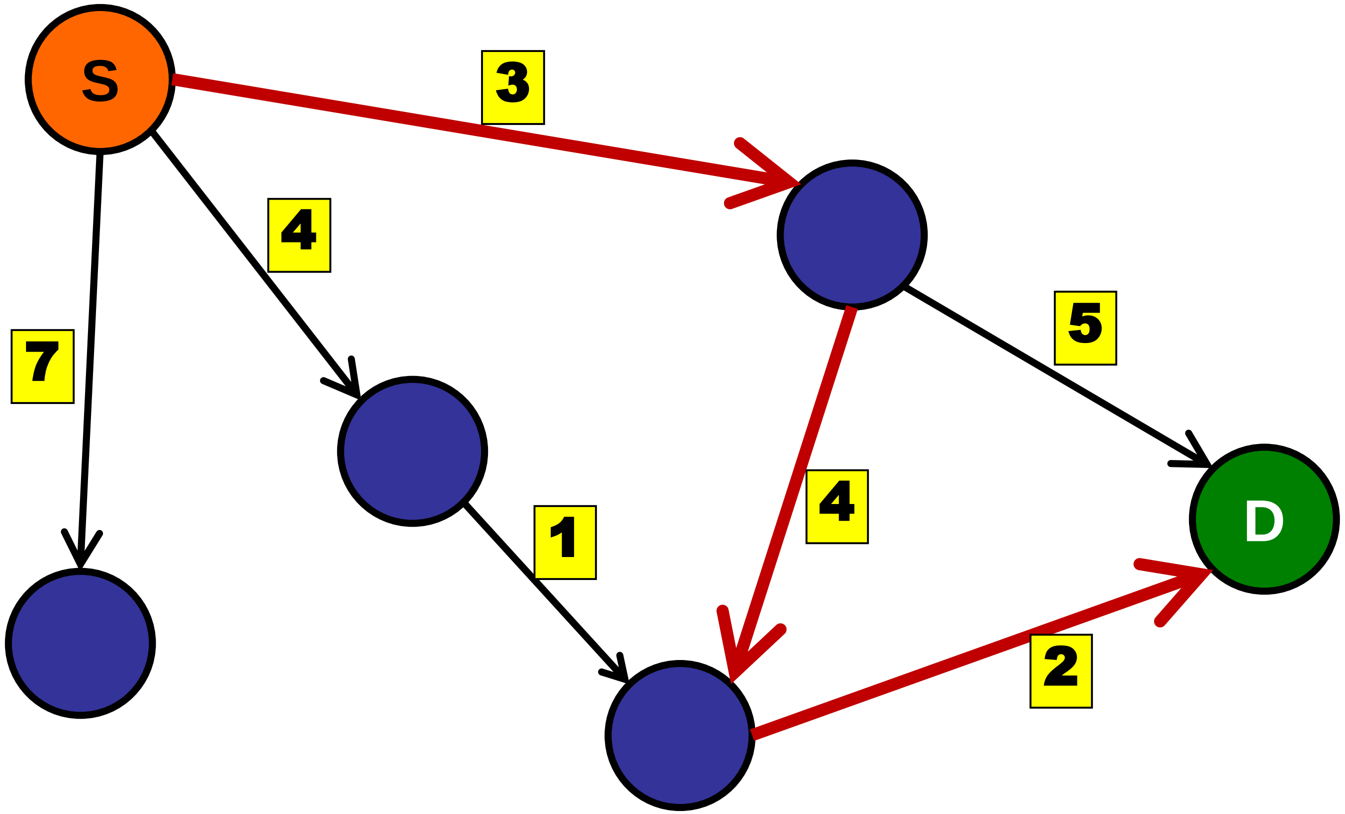
Shortest Paths

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



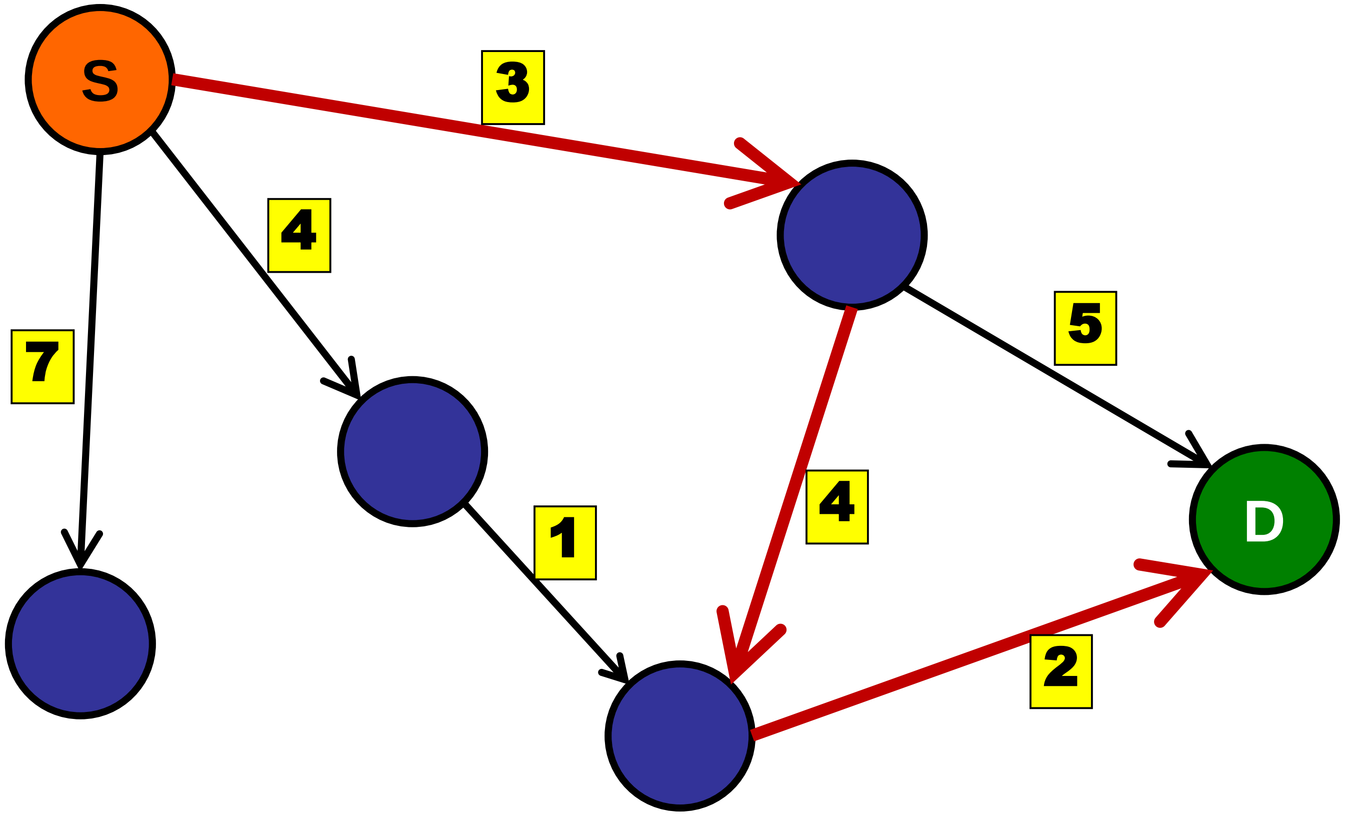
Shortest Paths

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



Shortest Paths

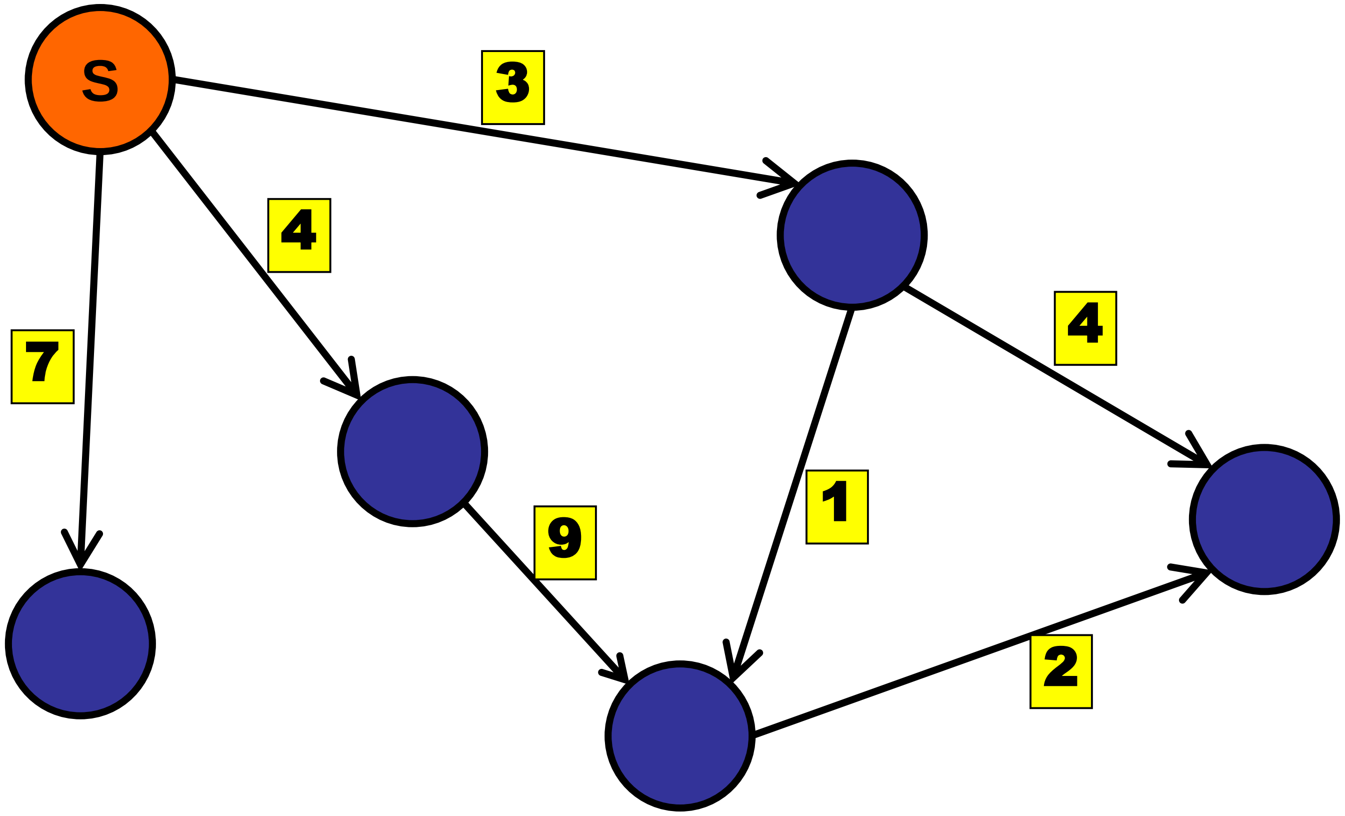
Common mistake: "Why can't I use BFS?"



BFS finds minimum number of HOPS not minimum DISTANCE.

Shortest Paths

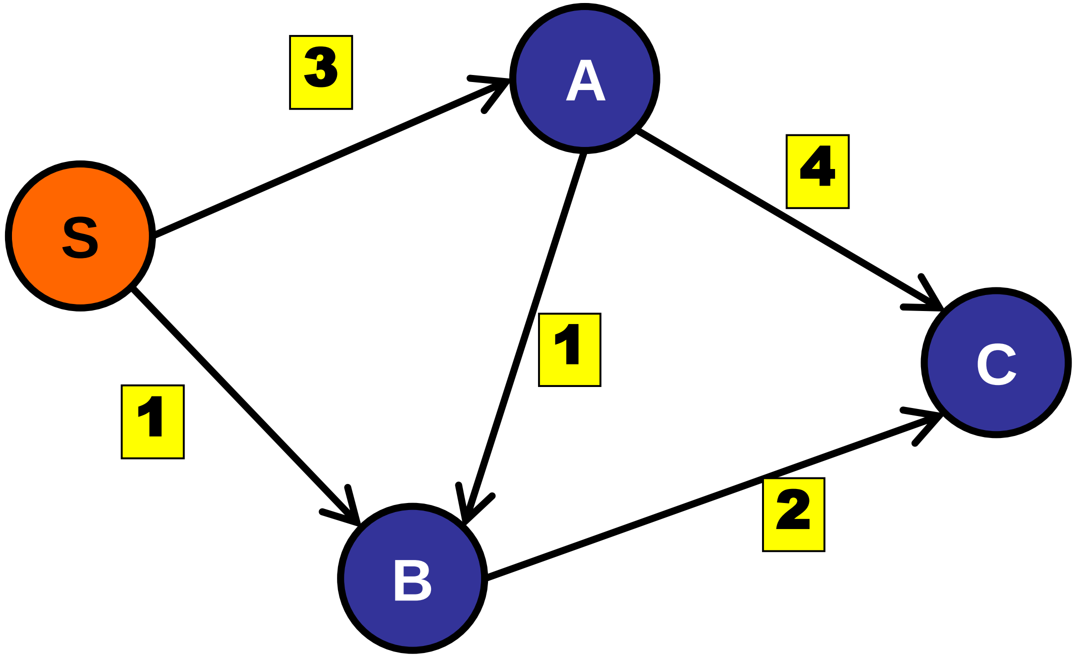
Notation: $\delta(u,v)$ = distance from u to v



Shortest Paths

Key idea: triangle inequality

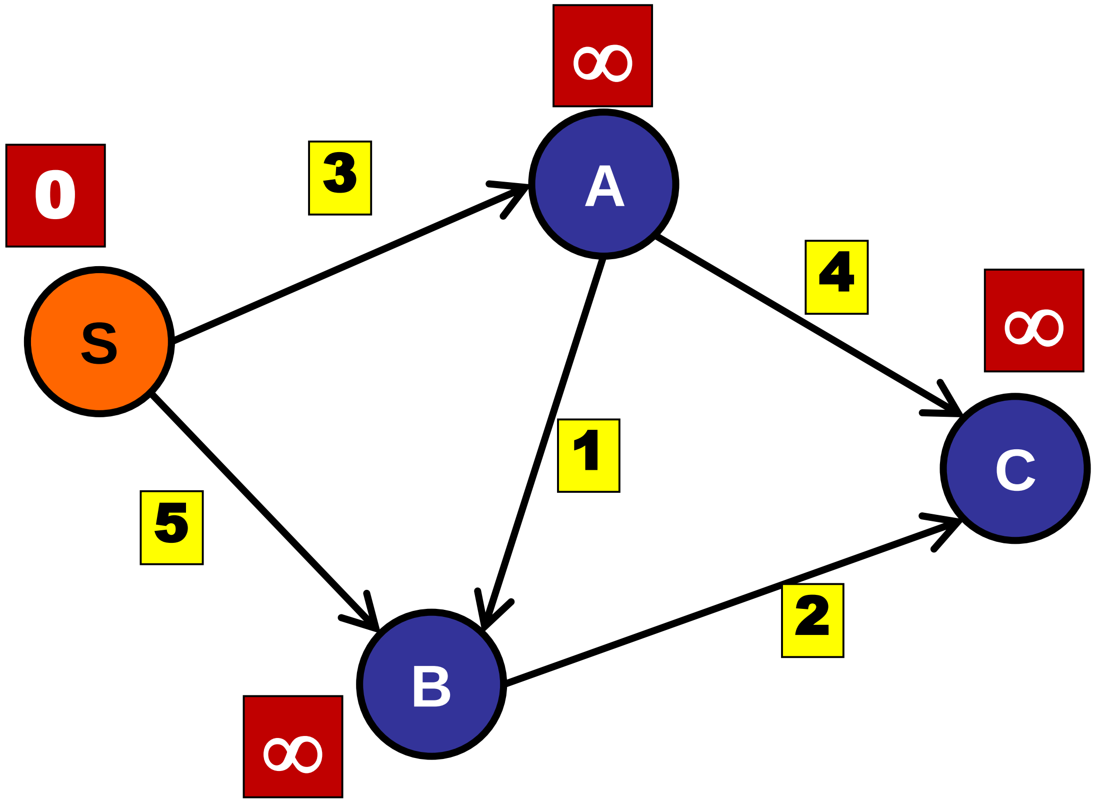
$\delta(S, C) \leq \delta(S, A) + \delta(A, C)$



Shortest Paths

Maintain estimate for each distance:

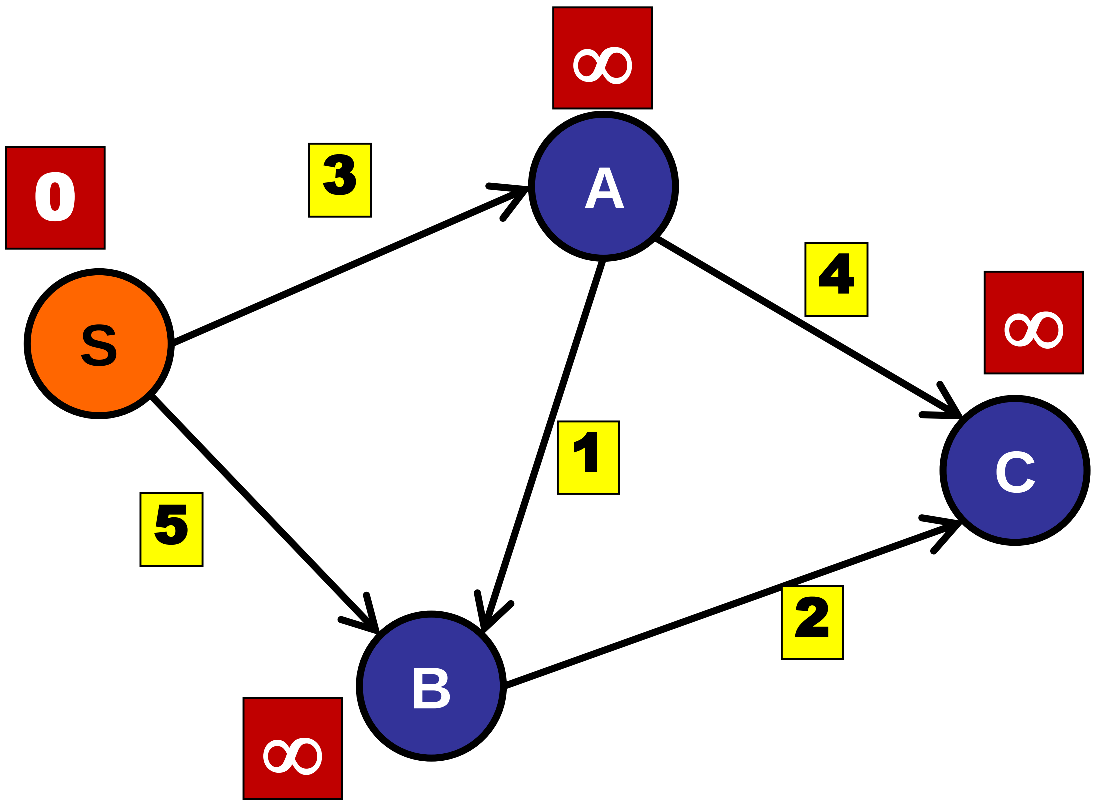
```
int[] dist = new int[V.length];

Arrays.fill(dist, INFTY);

dist[start] = 0;
```



Shortest Paths

Maintain estimate for each distance:

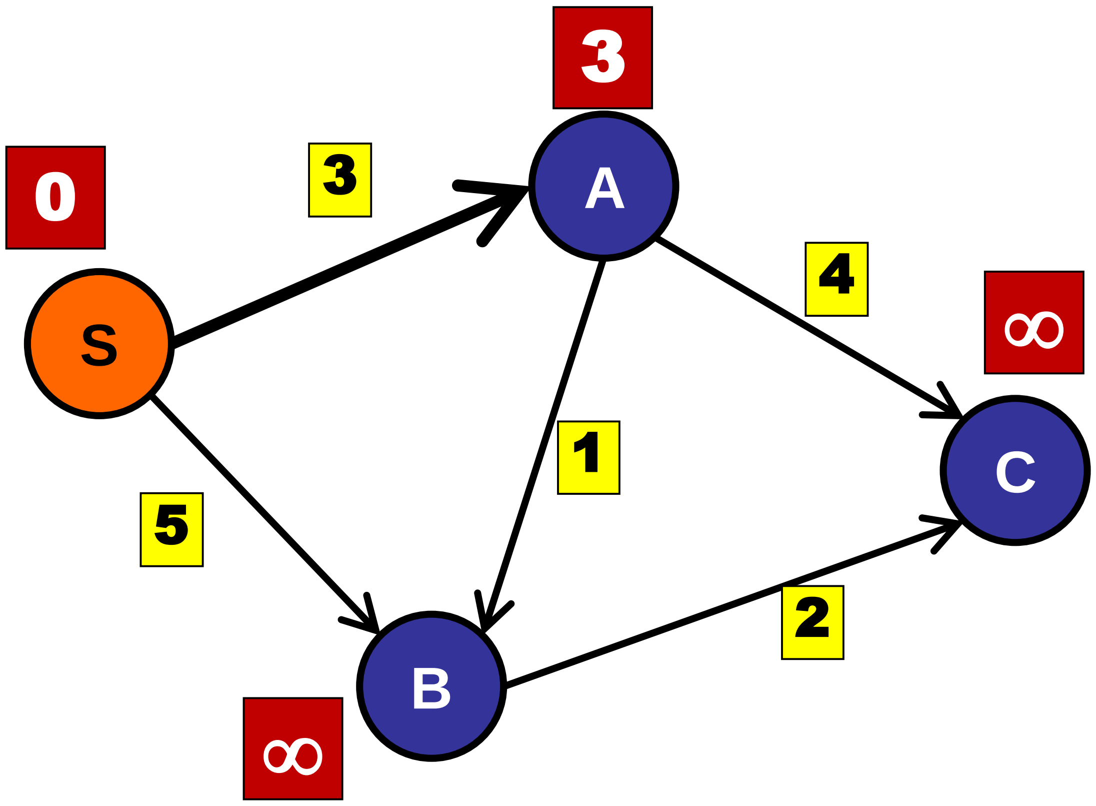
- Reduce estimate
- Invariant: estimate $\geq$ distance



Shortest Paths

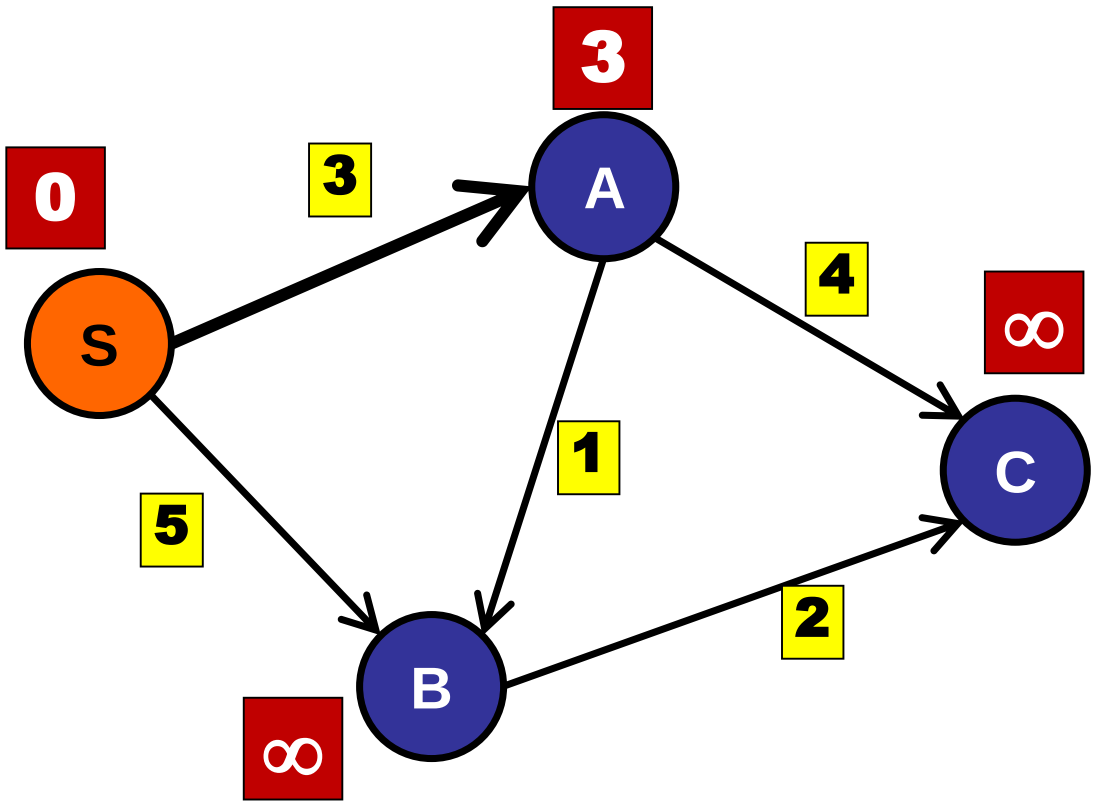
Maintain estimate for each distance:

relax(S, A)



Shortest Paths

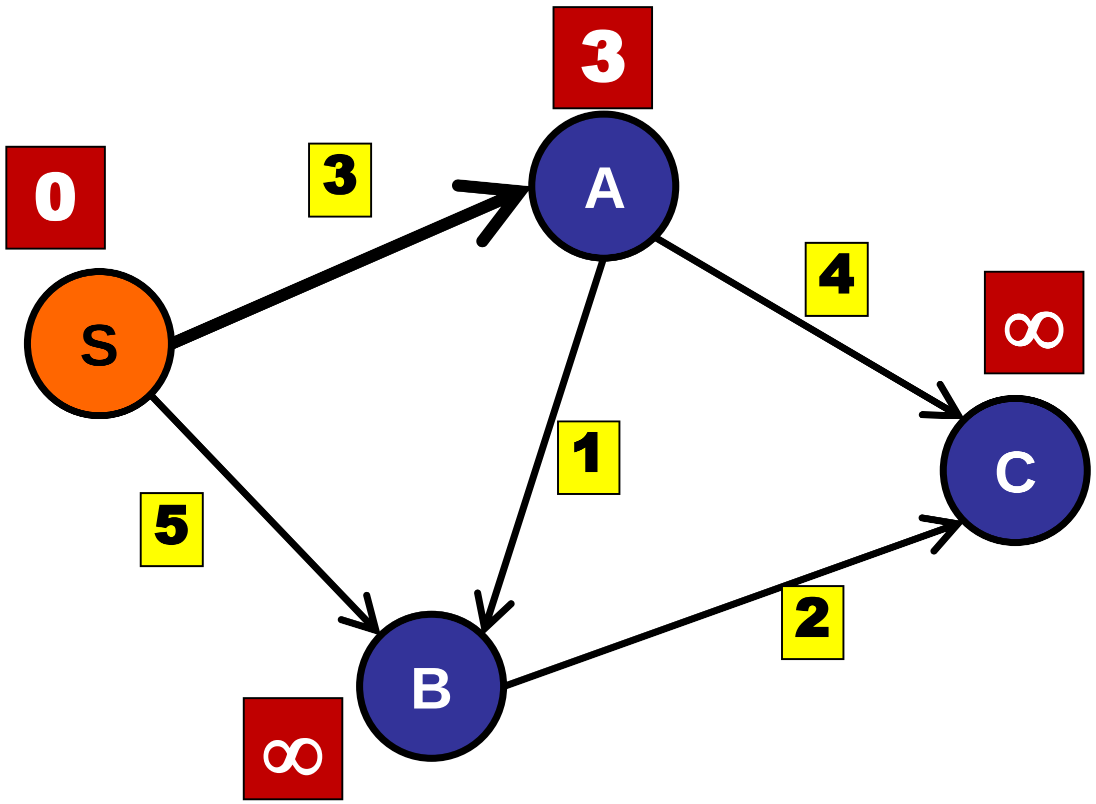
```
relax(int u, int v){
    if (dist[v] > dist[u] + weight(u,v))
        dist[v] = dist[u] + weight(u,v);
}
```



Shortest Paths

Maintain estimate for each distance:

relax(S, A)



Shortest Paths

Maintain estimate for each distance:

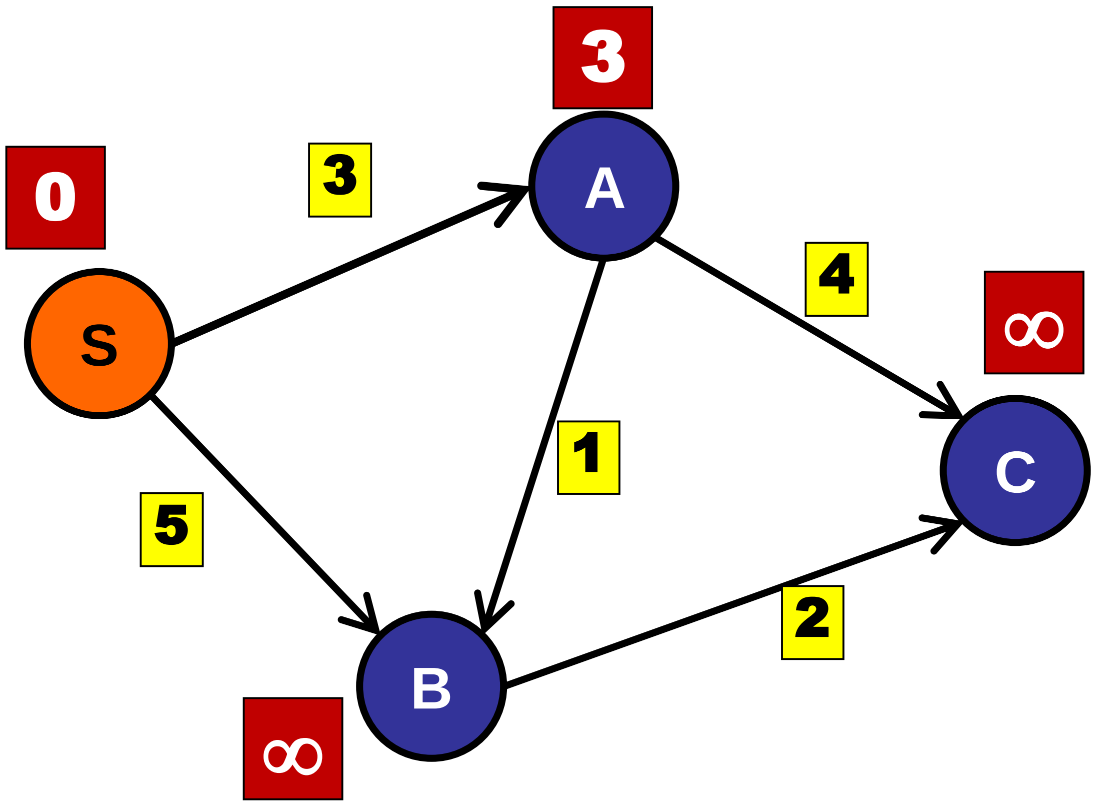
relax(A, C)

Triangle Inequality:

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

➔

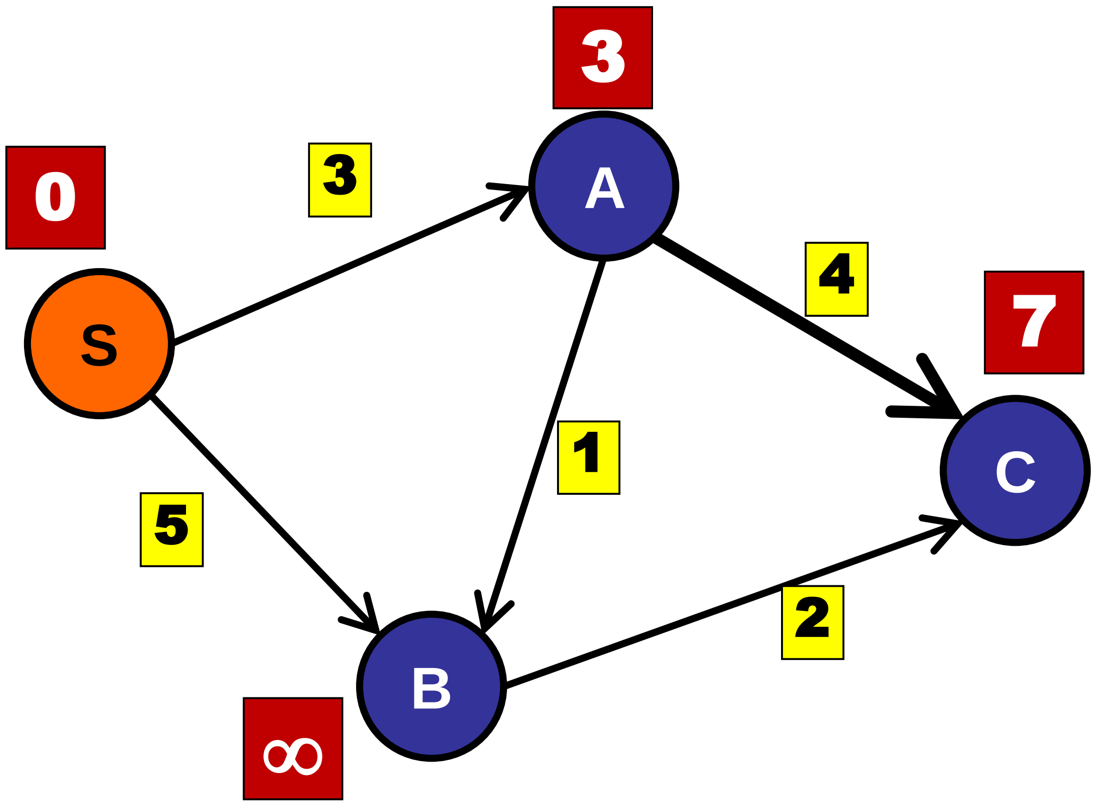
Can safely reduce estimate at C to 7.



Shortest Paths

Maintain estimate for each distance:

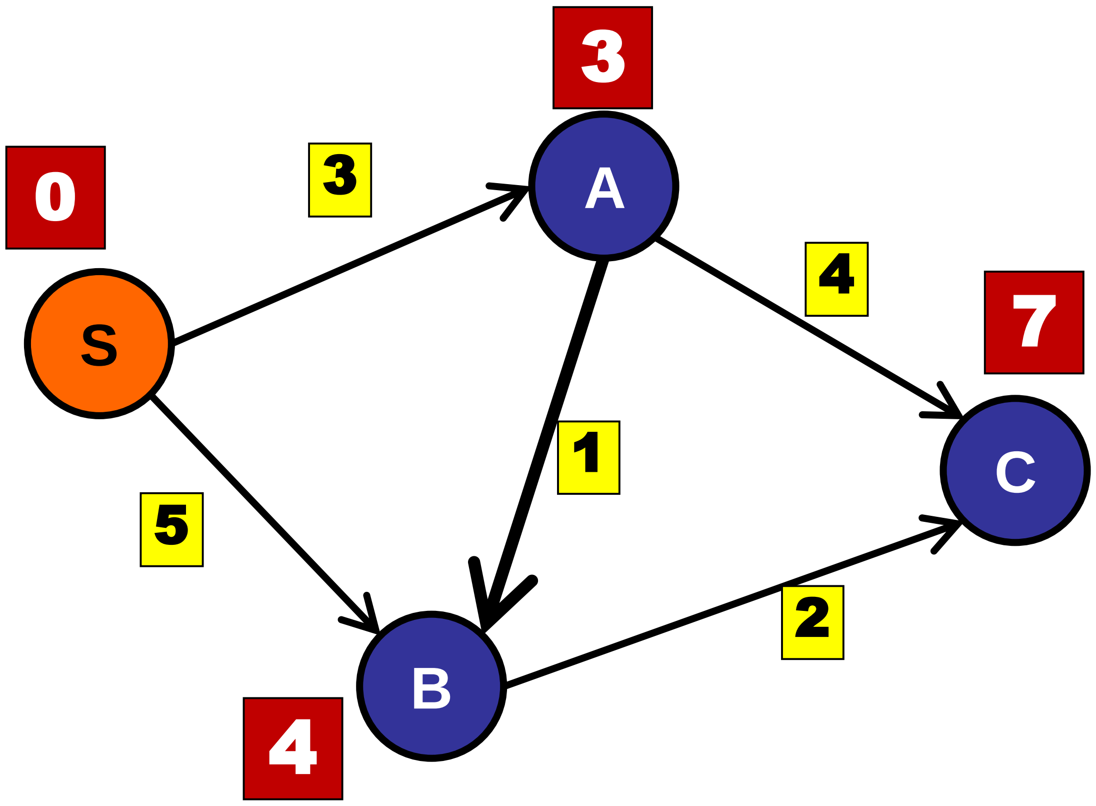
relax(A, C)



Shortest Paths

Maintain estimate for each distance:

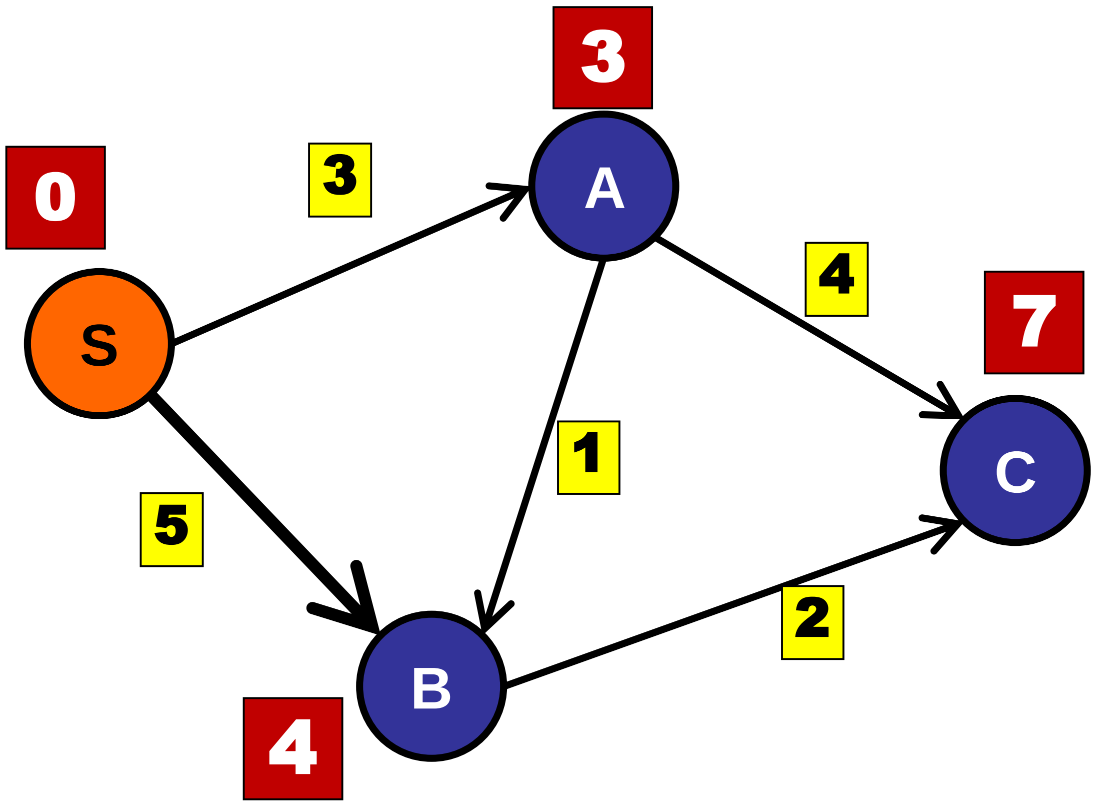
relax(A, B)



Shortest Paths

Maintain estimate for each distance:

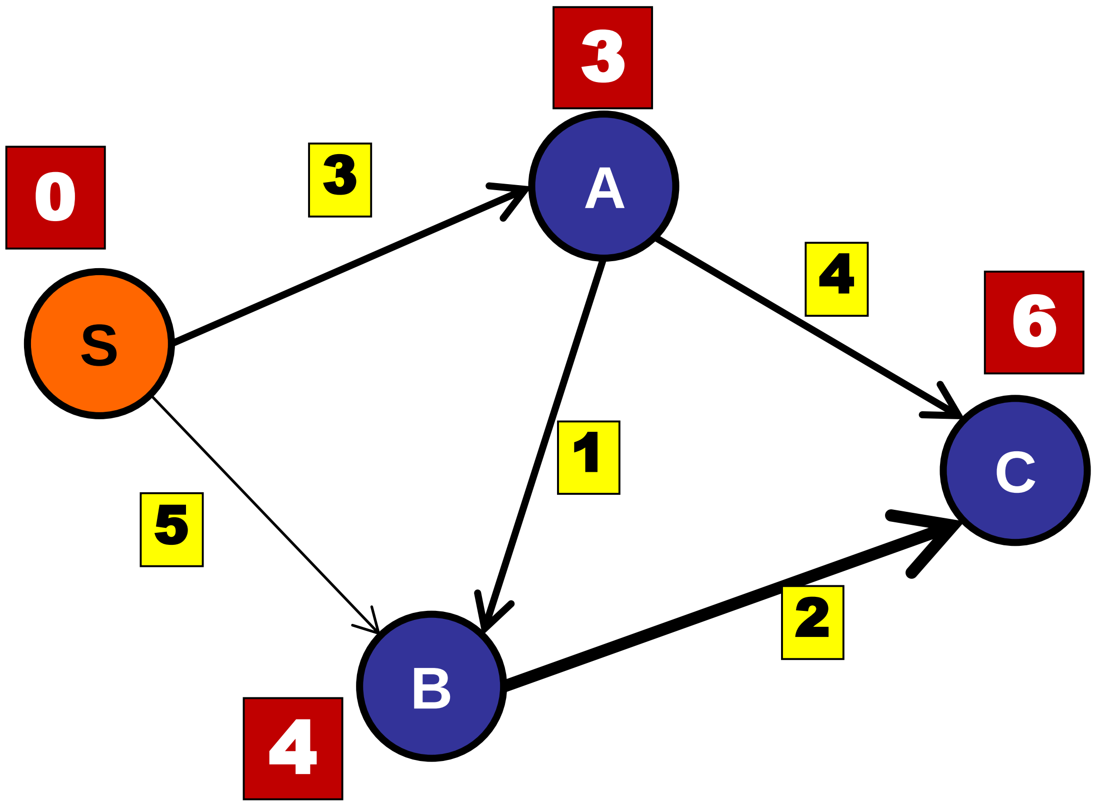
relax(S, B)



Shortest Paths

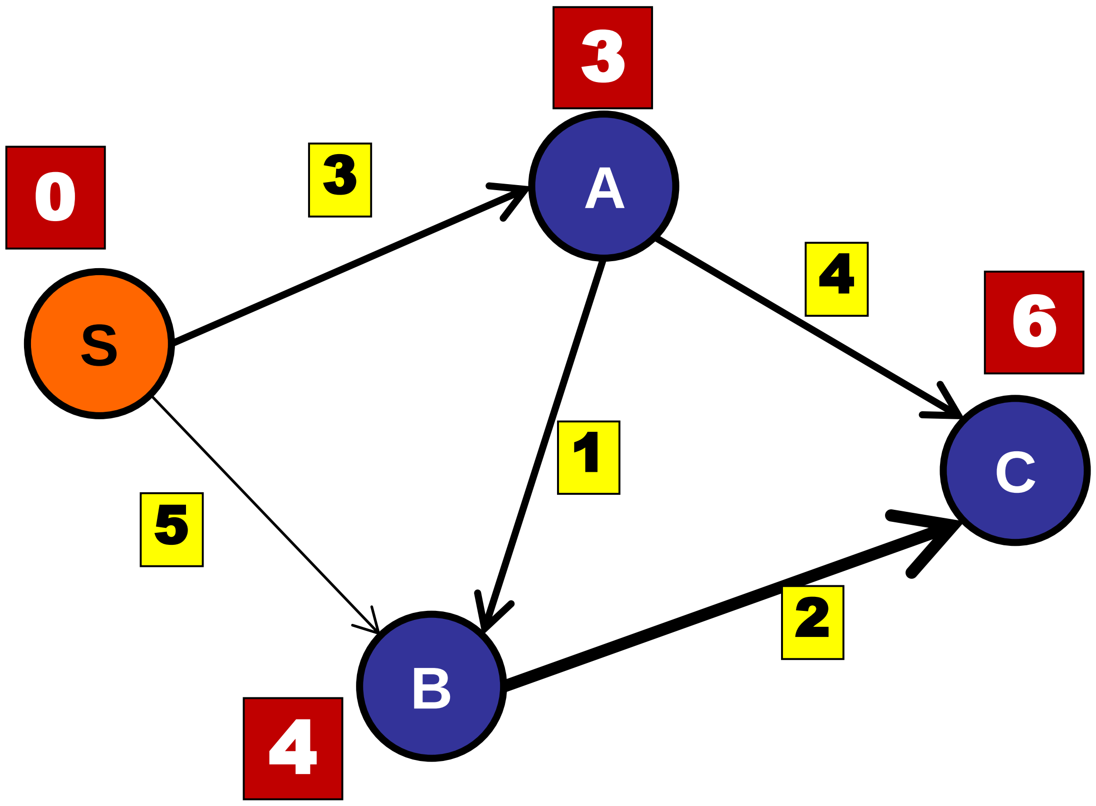
Maintain estimate for each distance:

relax(B, C)



Shortest Paths

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

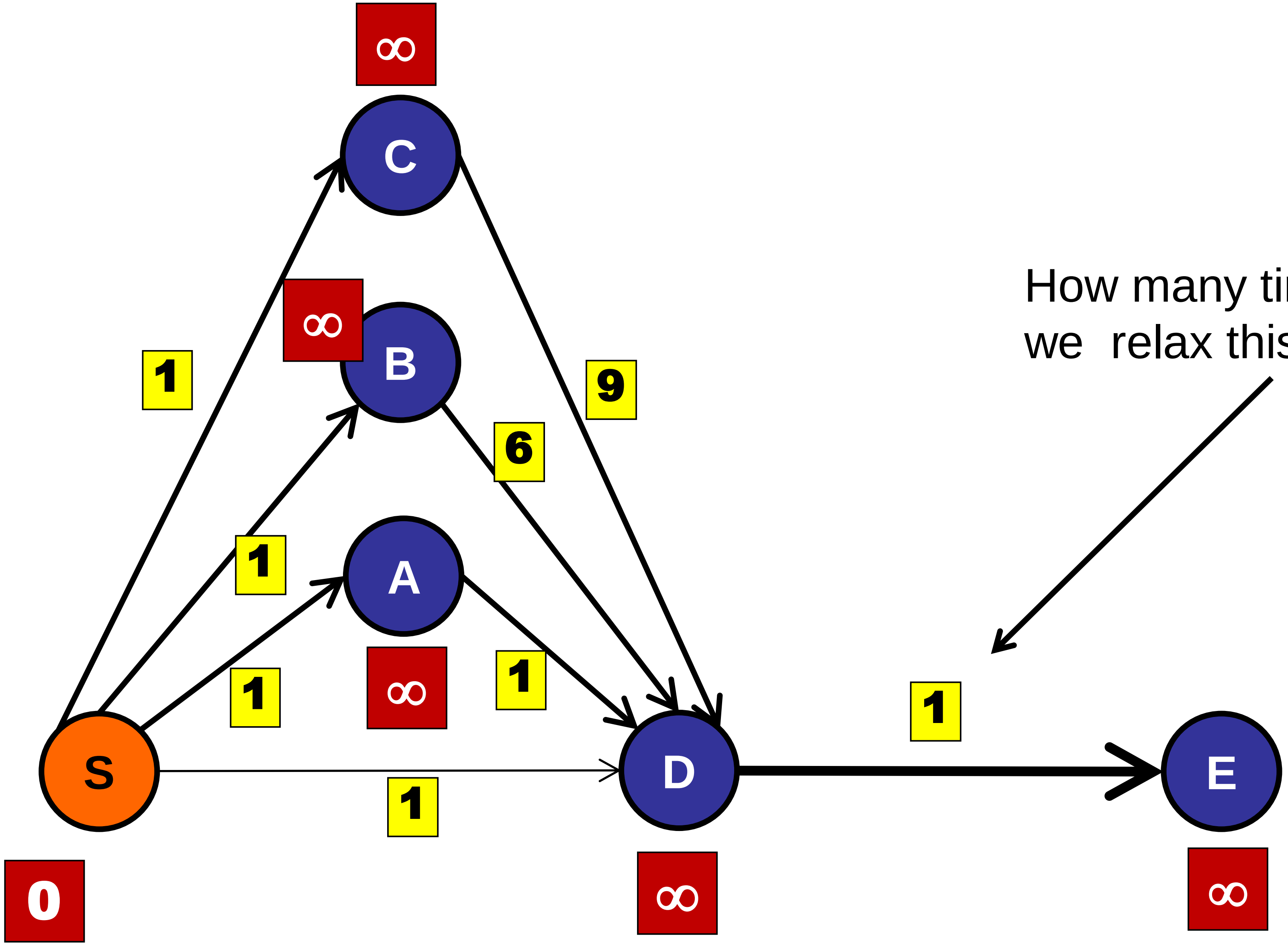


Does this algorithm work:

for every edge e: relax(e)

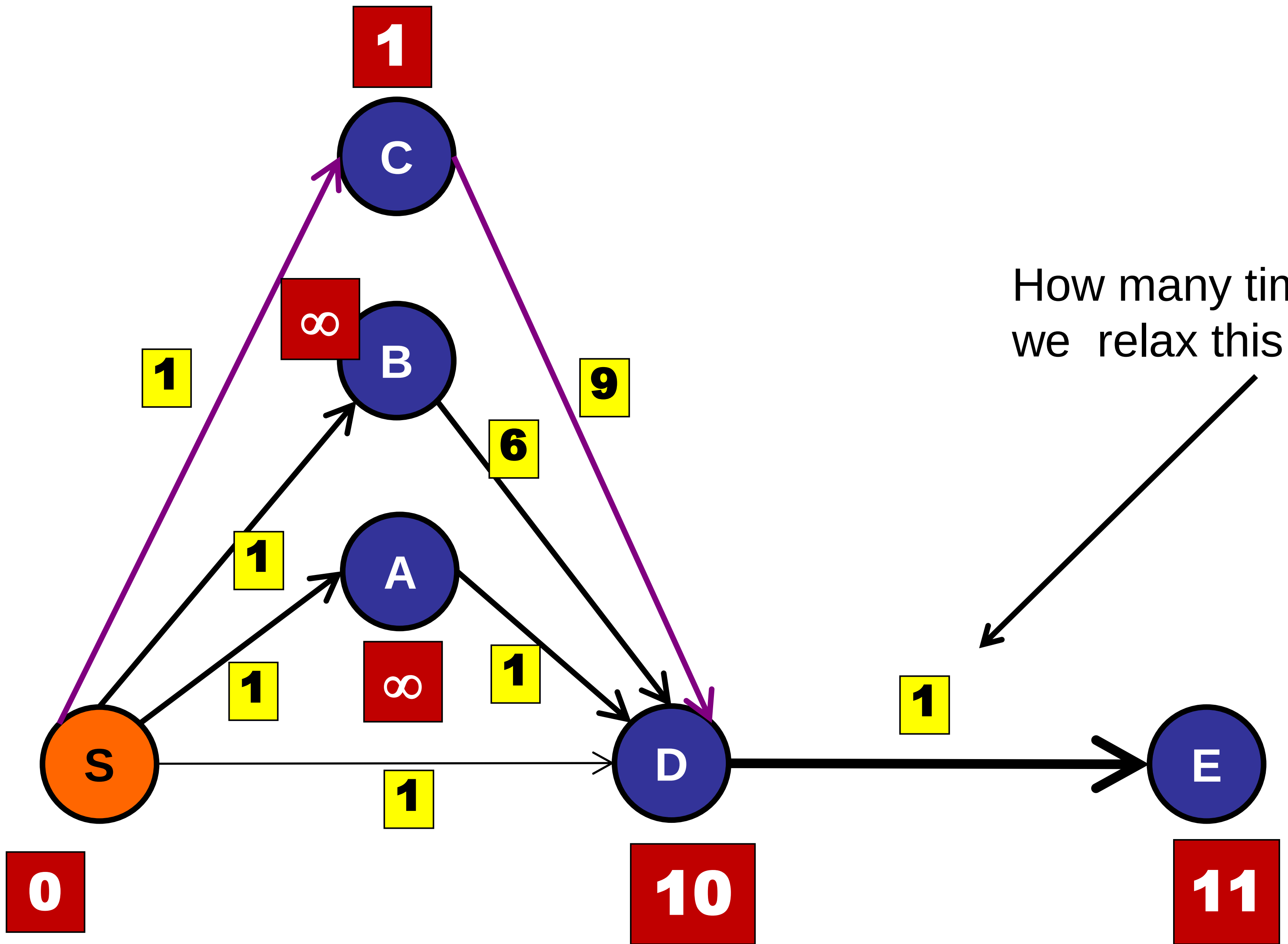
1. Yes
2. Sometimes
3. No

Shortest Paths



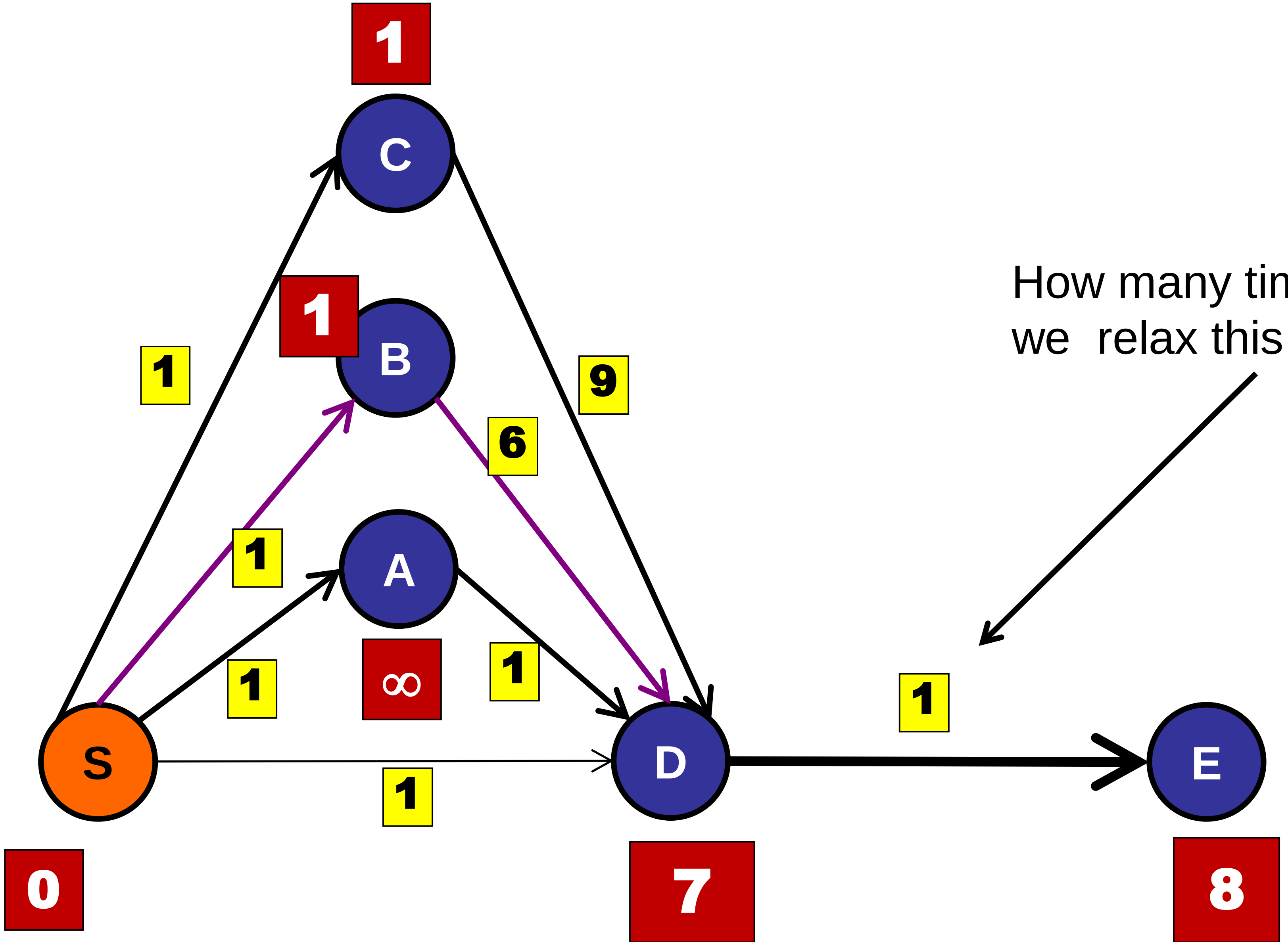
How many times might we relax this edge?

Shortest Paths



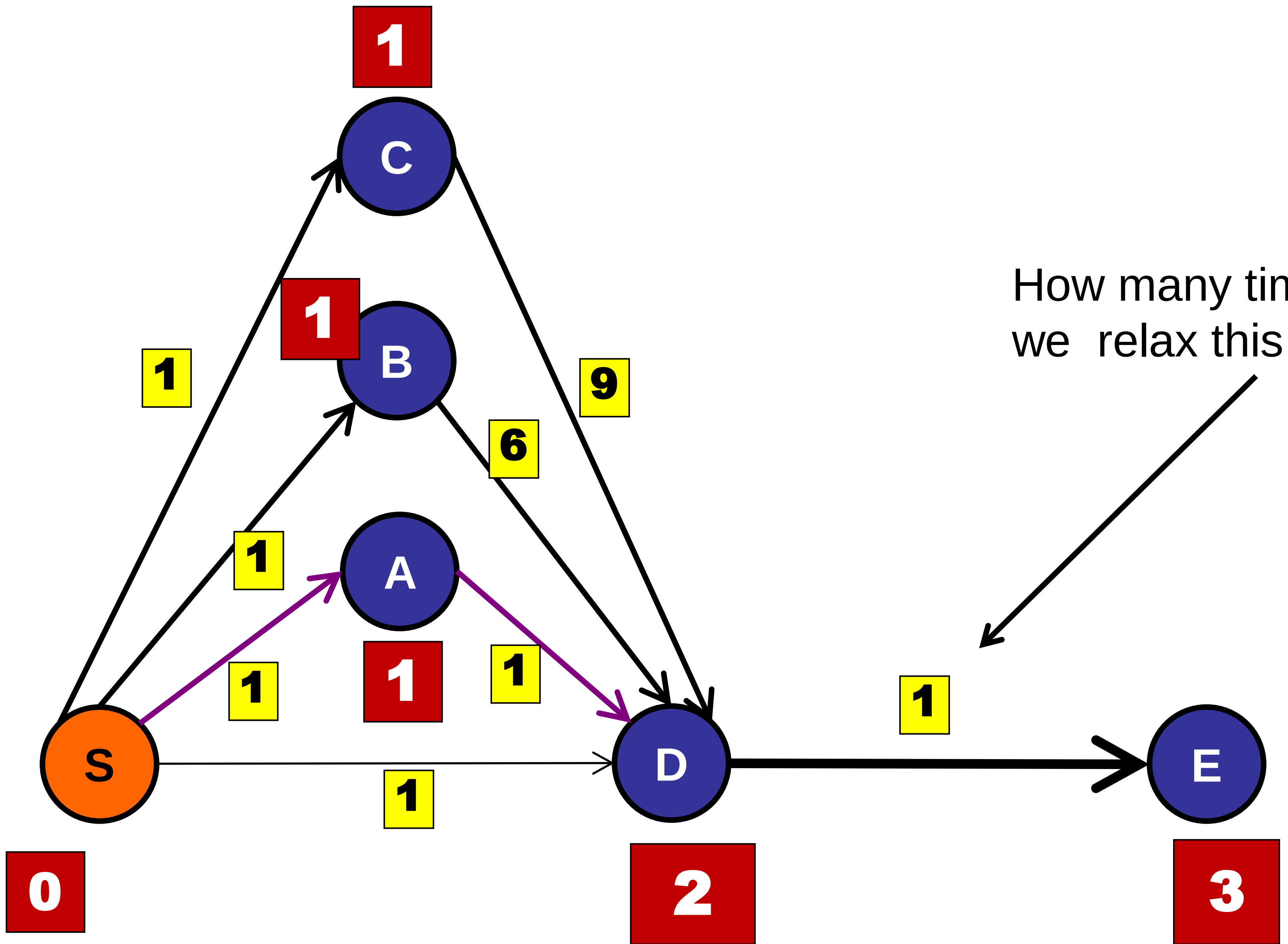
How many times might we relax this edge?

Shortest Paths



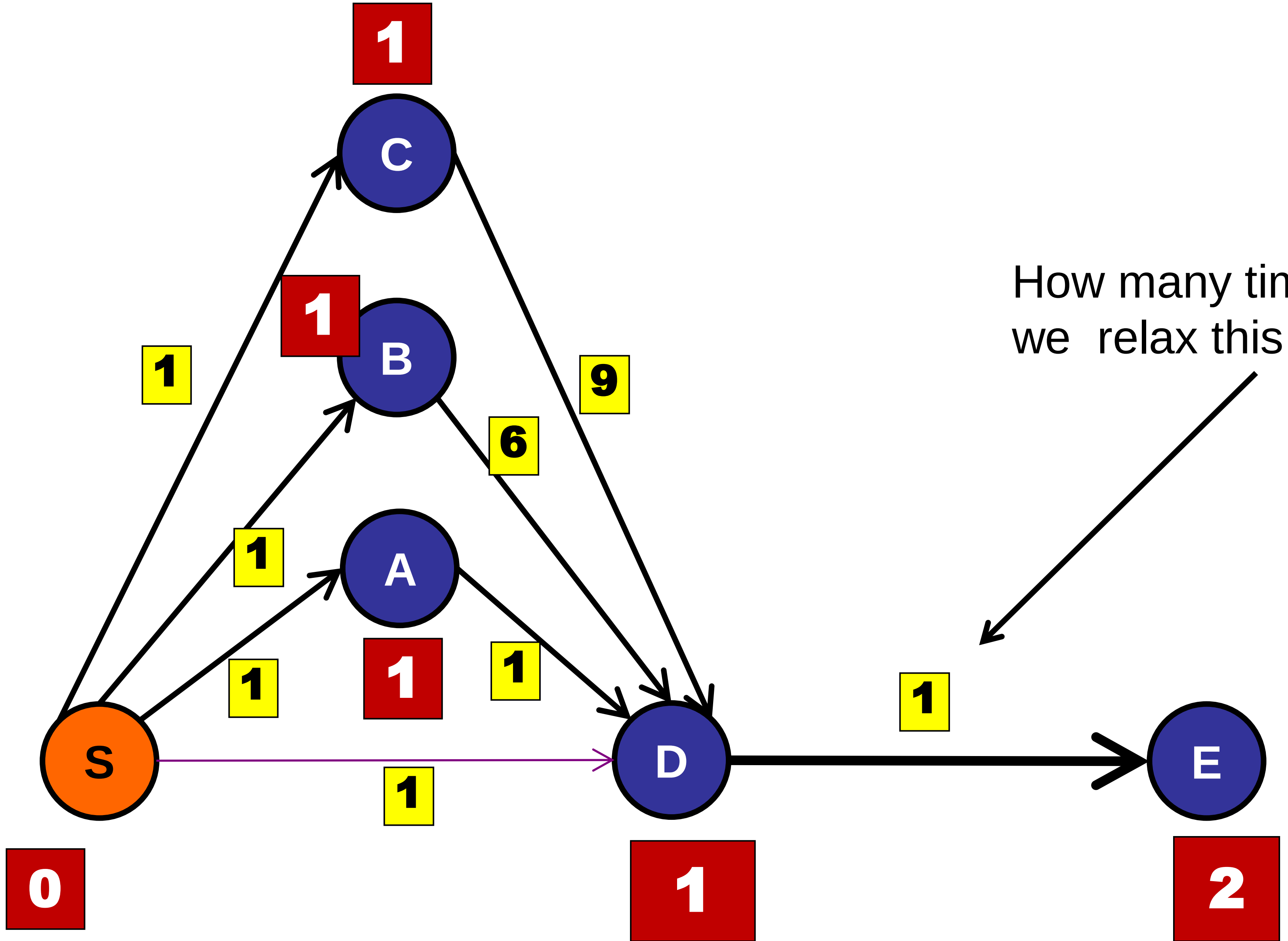
How many times might we relax this edge?

Shortest Paths



How many times might we relax this edge?

Shortest Paths



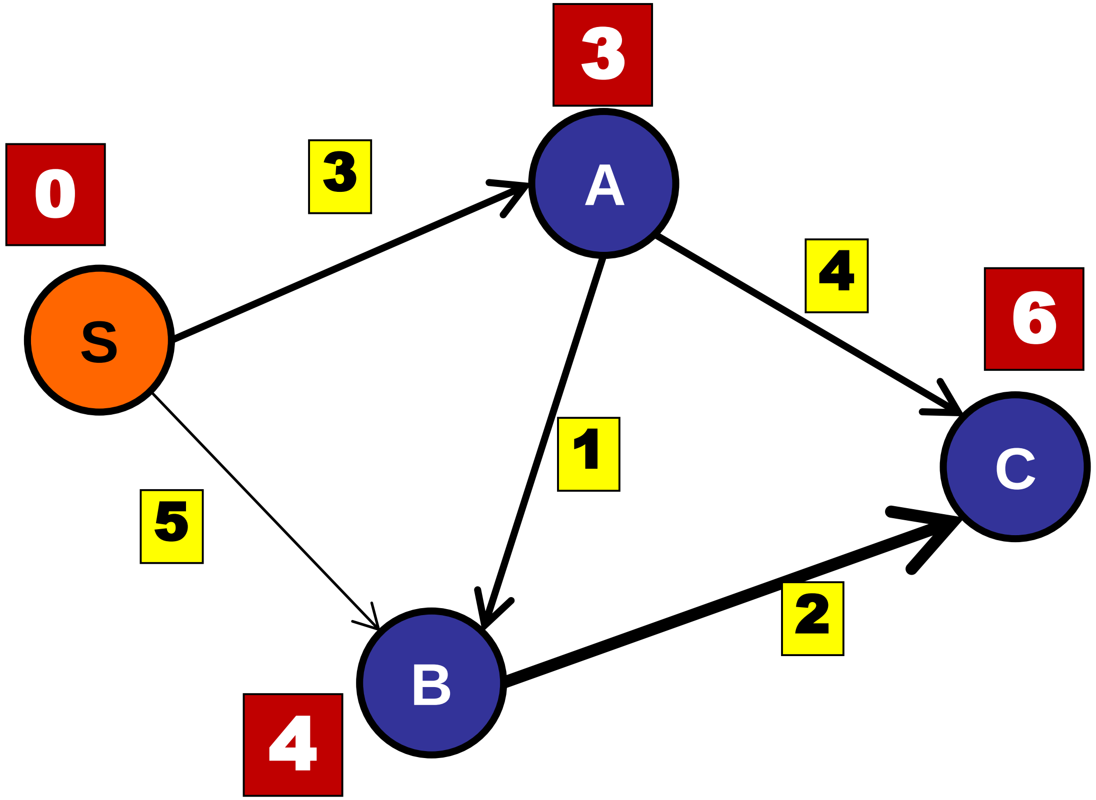
How many times might we relax this edge?

Bellman-Ford

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

Richard Bellman

Lester R. Ford, Jr

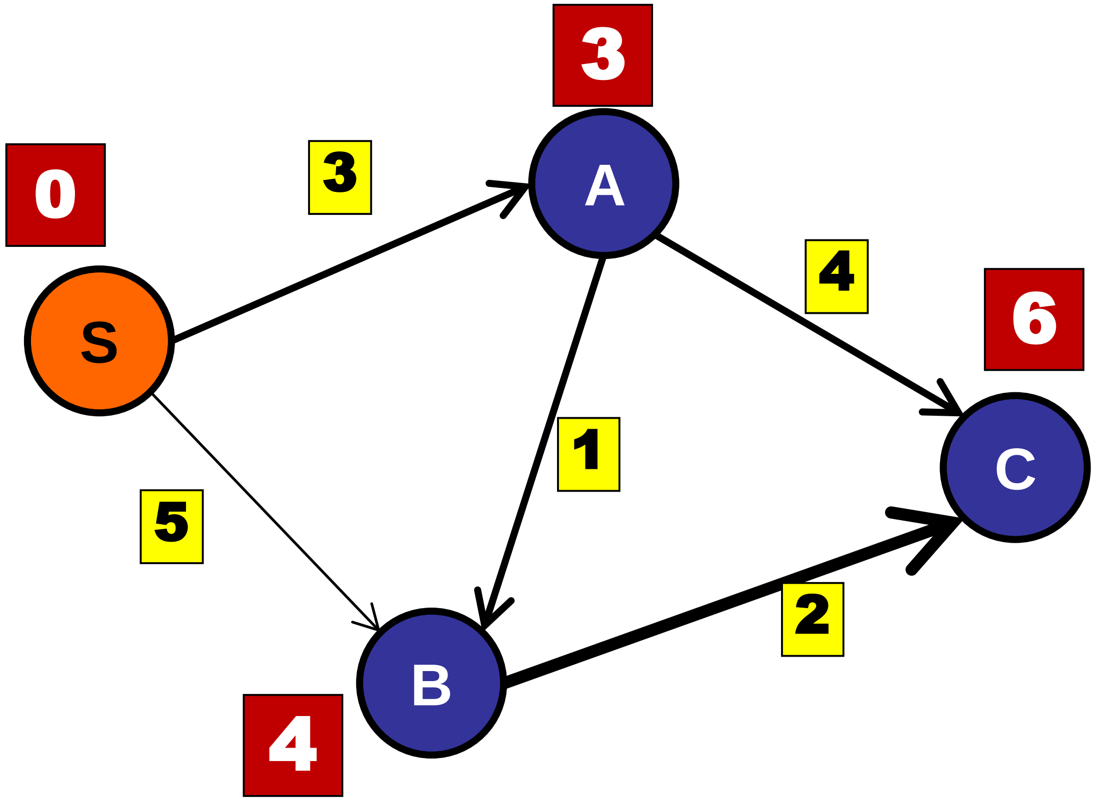


When can you terminate early?

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)
```

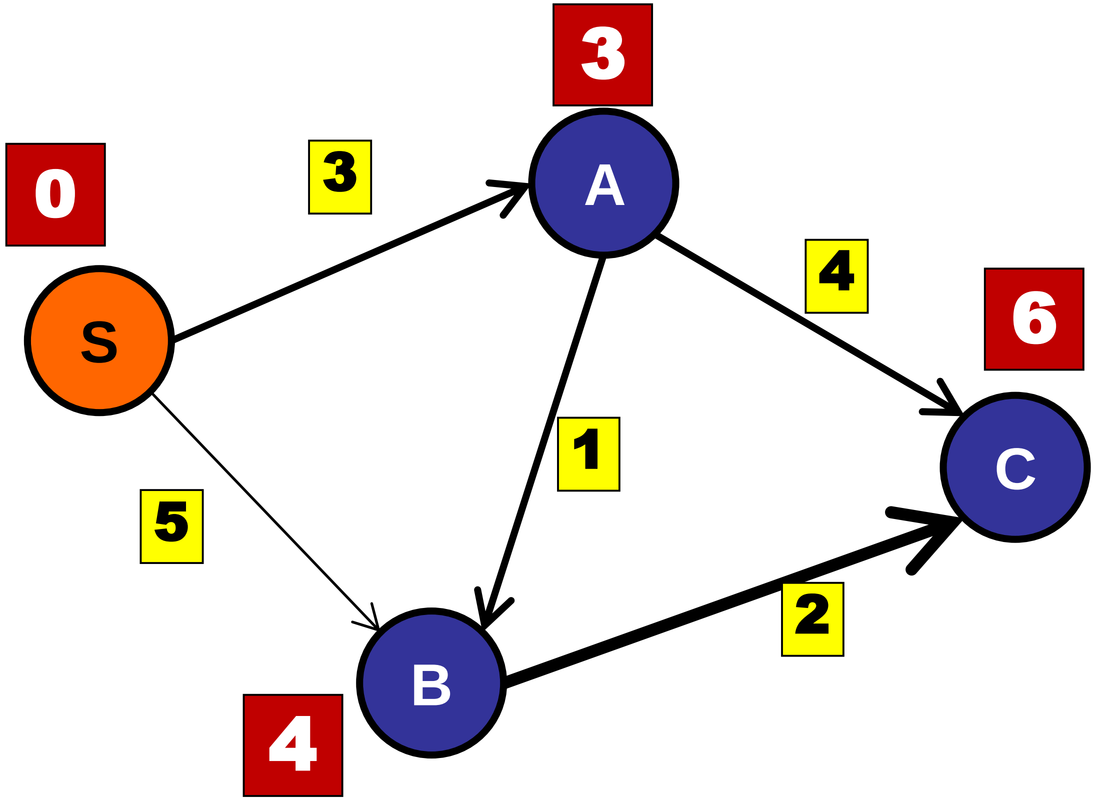


What is the running time of Bellman-Ford?

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

Properties:

- O(EV) running time (in the worst-case)
- Can stop after one entire iteration with no changes to the estimates.

Invariant:

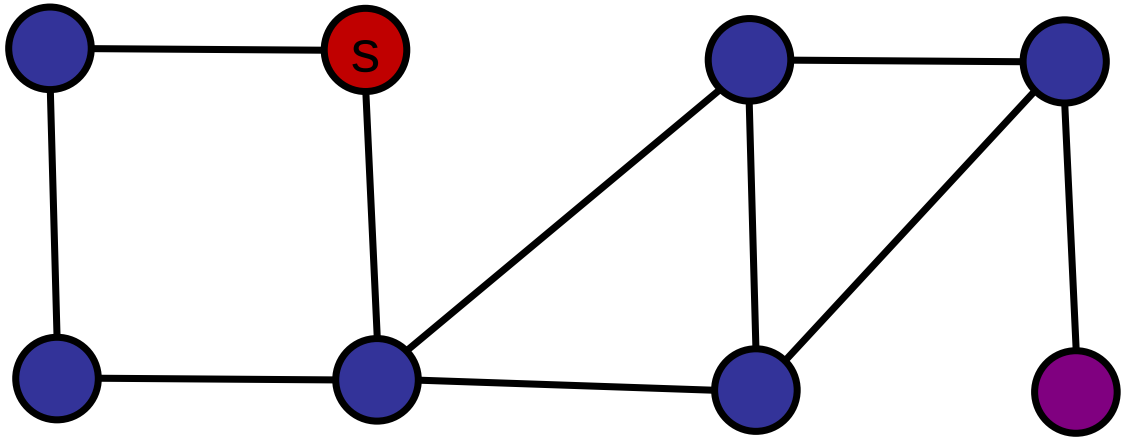
- Let T be a *shortest path tree* of graph G rooted at source s.
- After iteration j, if node u is j hops from s on tree T, then est[u] = distance(s, u).

Bellman-Ford

Why does this work?

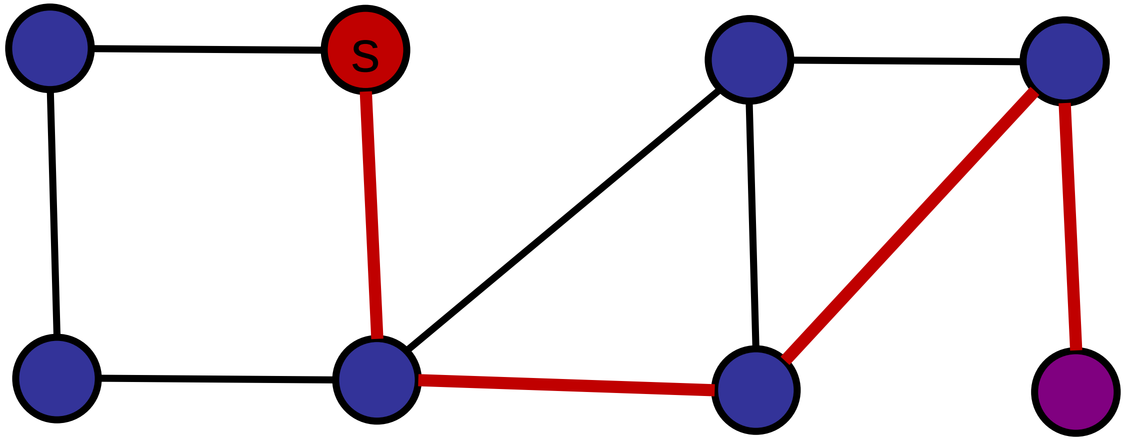
Bellman-Ford

Why does this work?



Bellman-Ford

Why does this work?

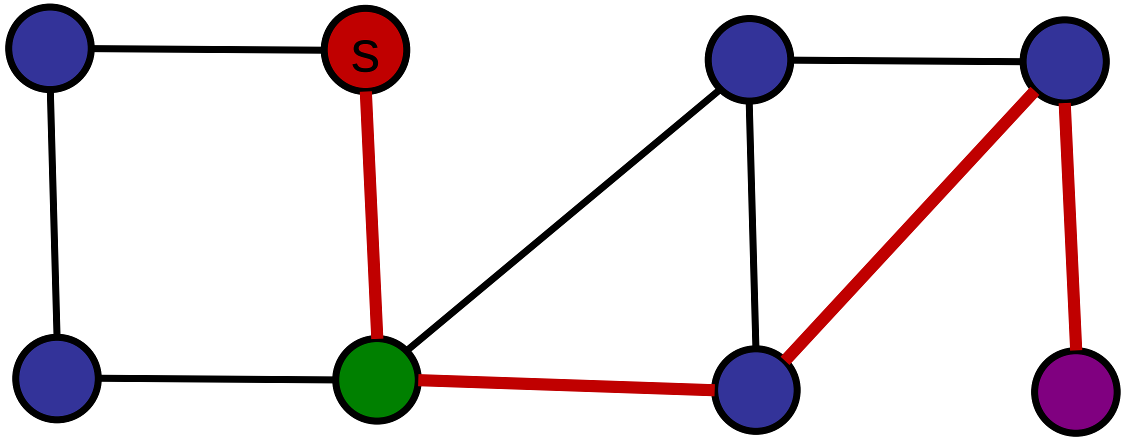


Look at minimum weight path from S to D.

(Path is simple: no loops.)

Bellman-Ford

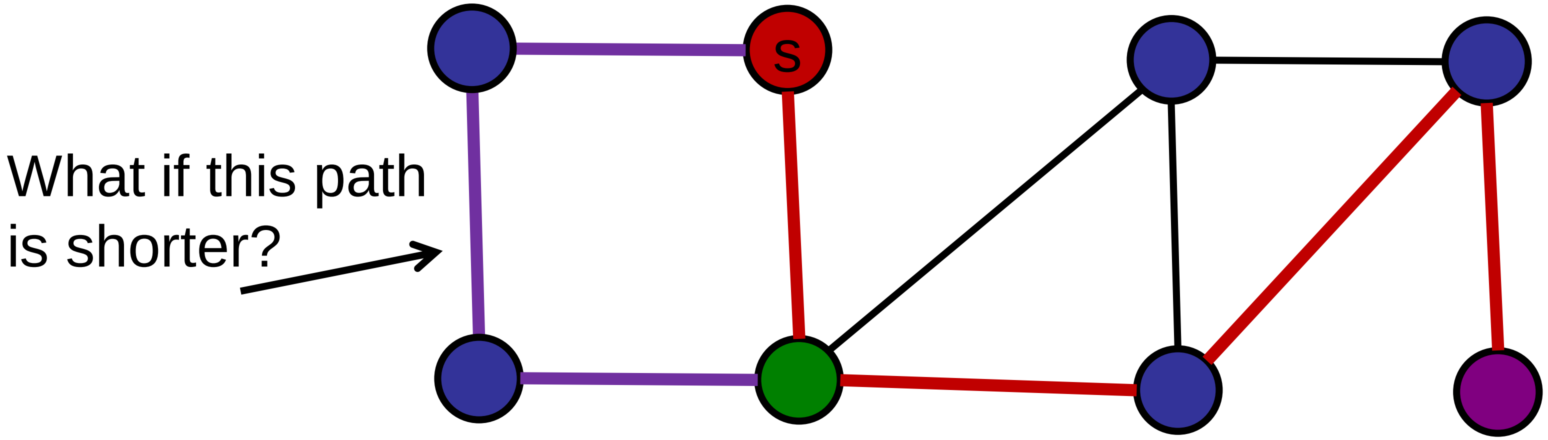
Why does this work?



After 1 iteration, 1 hop estimate is correct.

Bellman-Ford

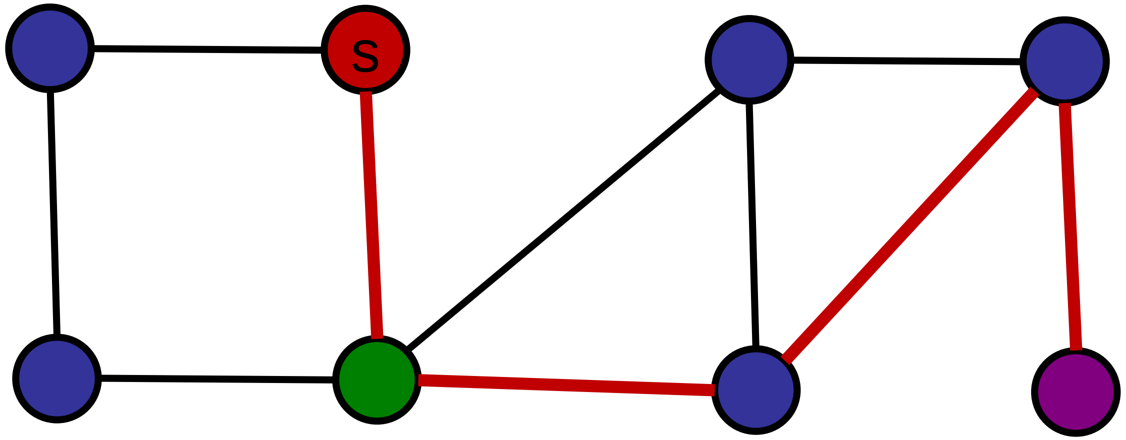
Why does this work?



After 1 iteration, 1 hop estimate is correct.

Bellman-Ford

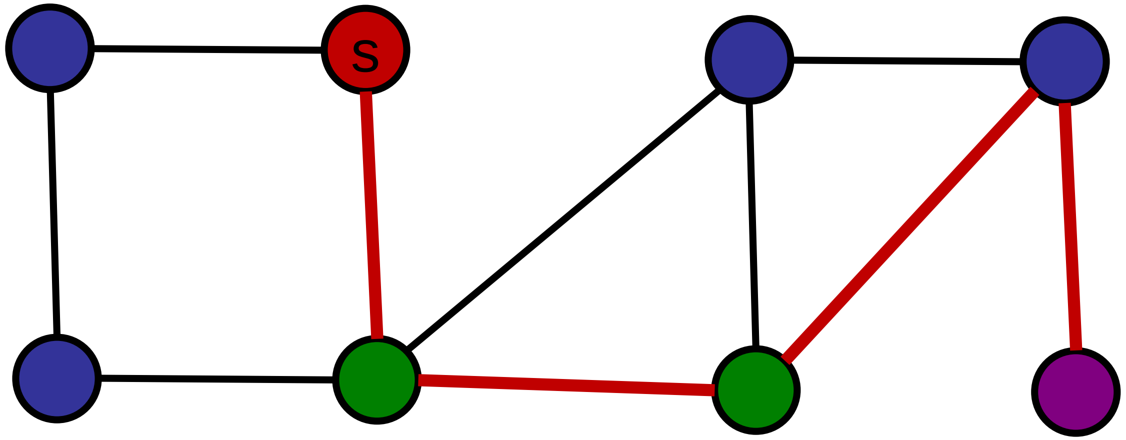
Why does this work?



After 1 iteration, 1 hop estimate is correct.

Bellman-Ford

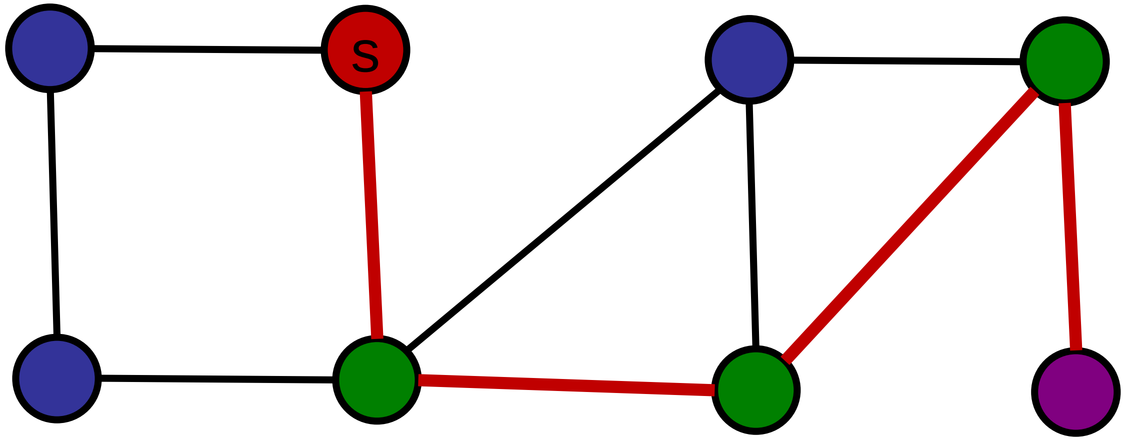
Why does this work?



After 2 iterations, 2 hop estimate is correct.

Bellman-Ford

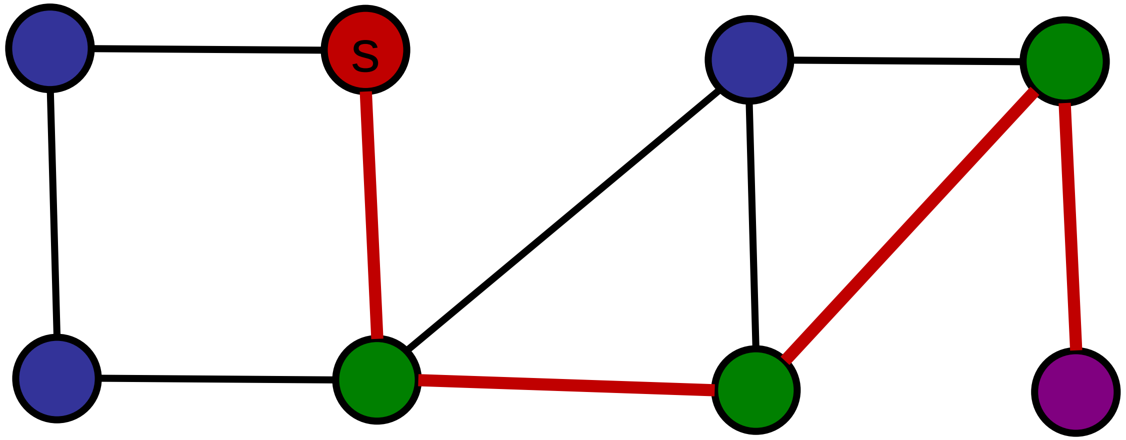
Why does this work?



After 3 iterations, 3 hop estimate is correct.

Bellman-Ford

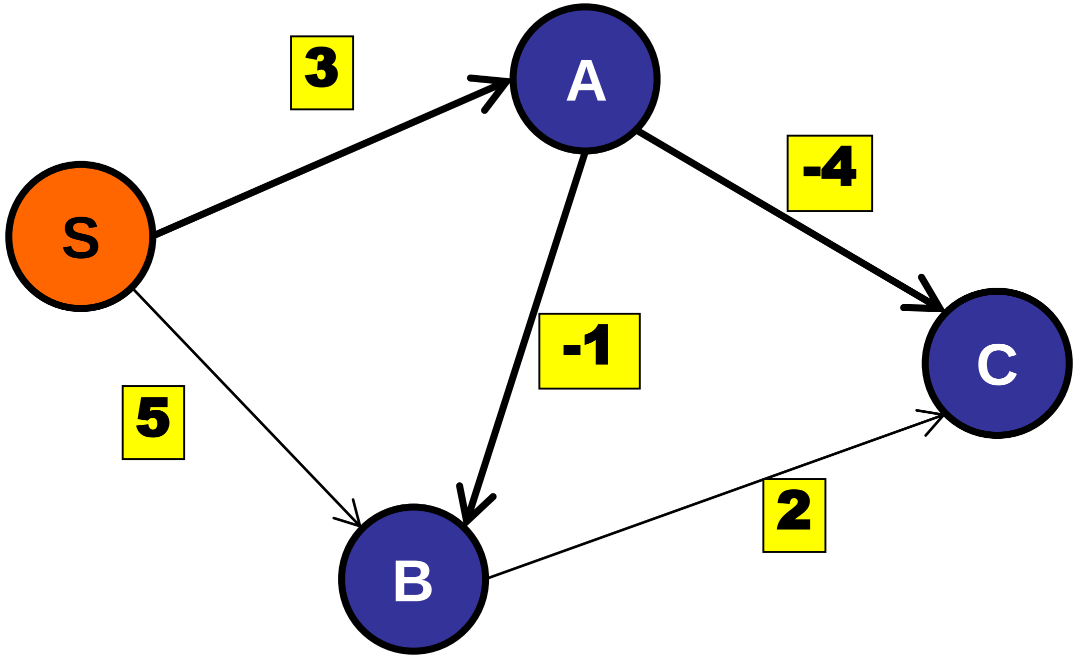
Why does this work?



After 4 iterations, D estimate is correct.

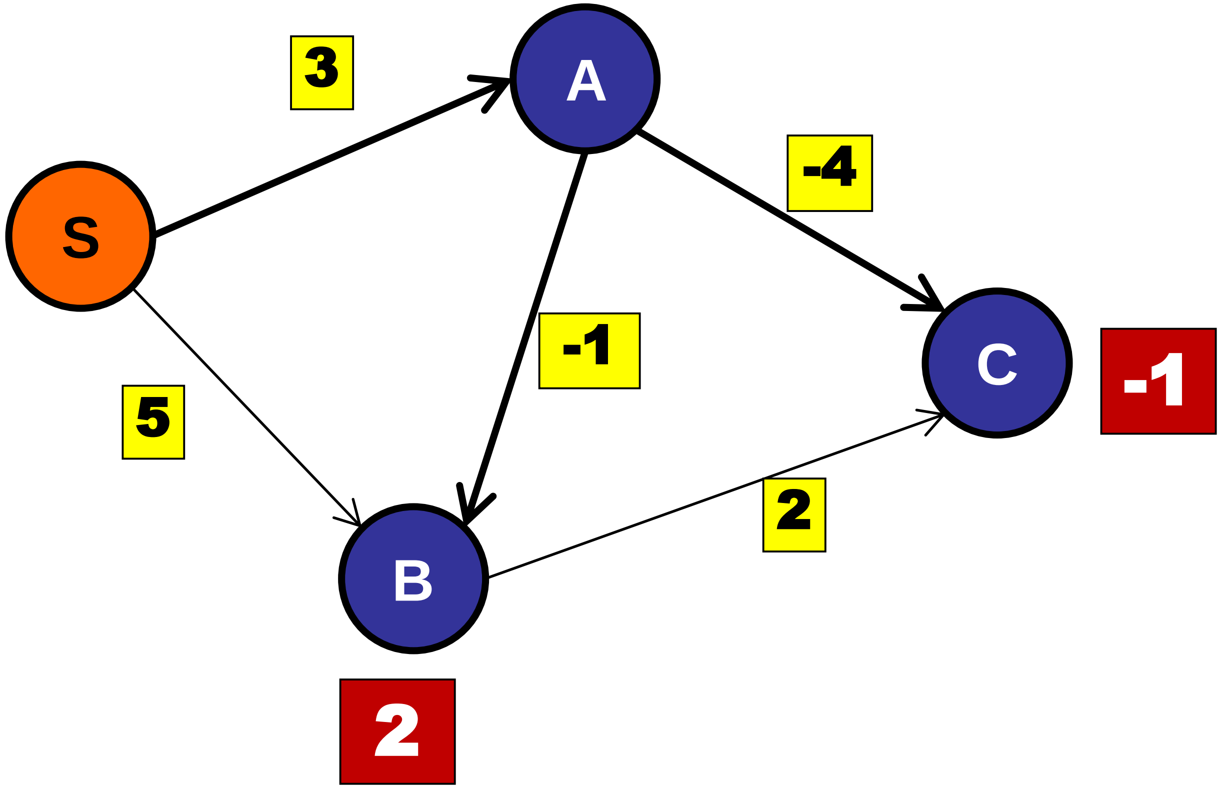
Bellman-Ford

What if edges have negative weight?



Bellman-Ford

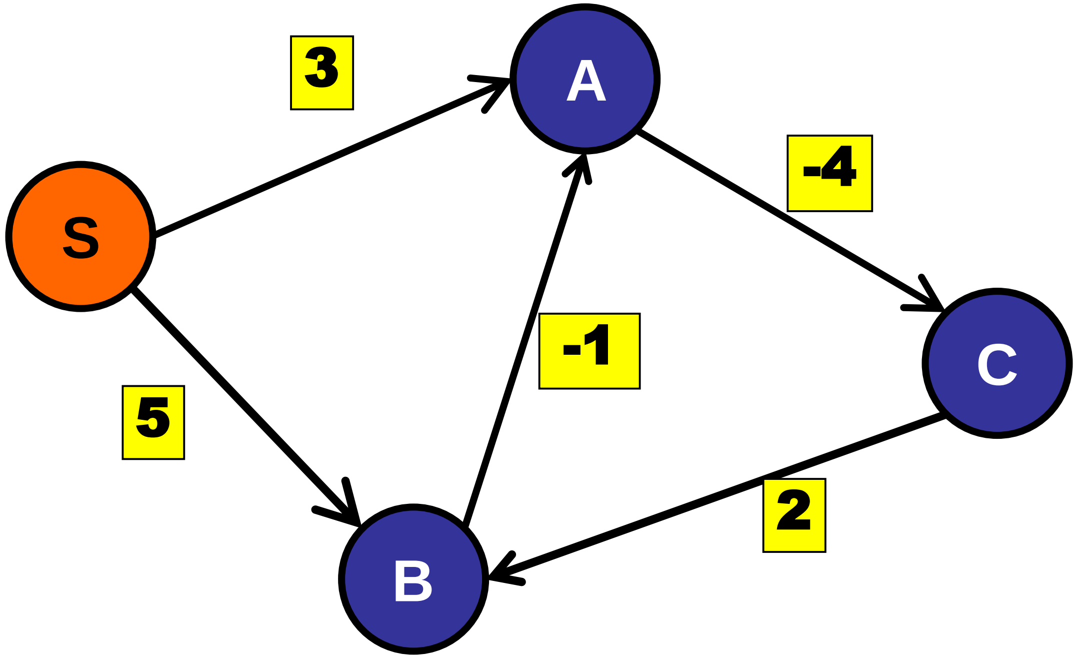
What if edges have negative weight?



No problem!

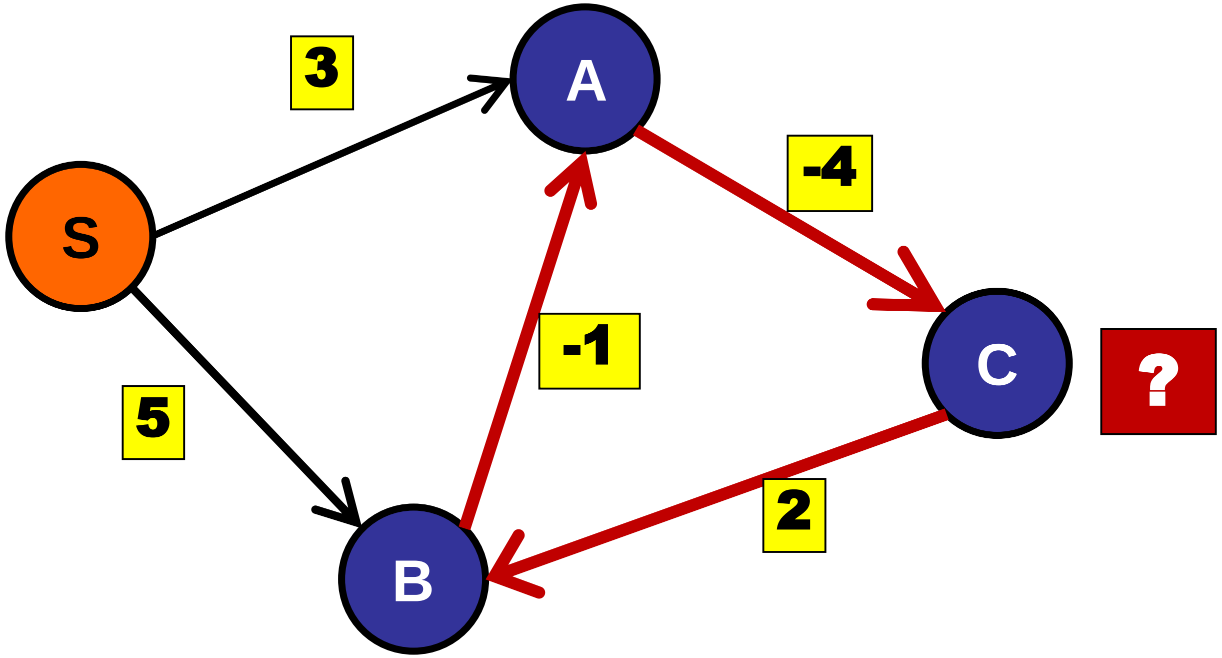
Bellman-Ford

What if edges have negative weight?



Bellman-Ford

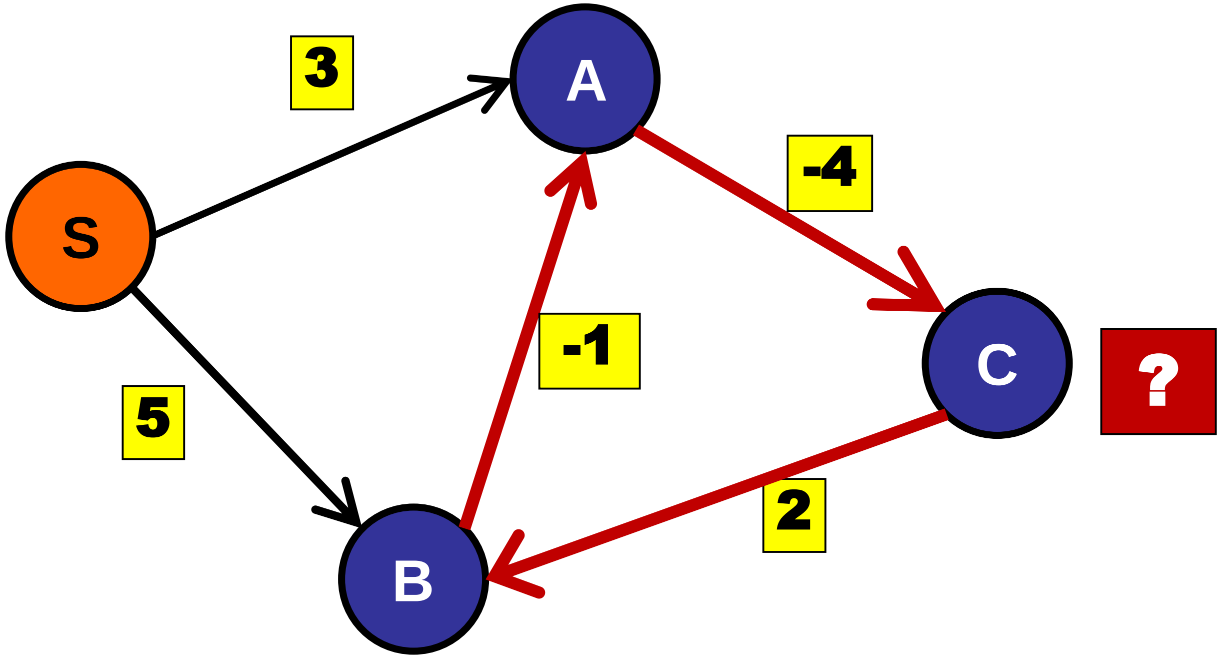
What if edges have negative weight?



d(S,C) is infinitely negative!

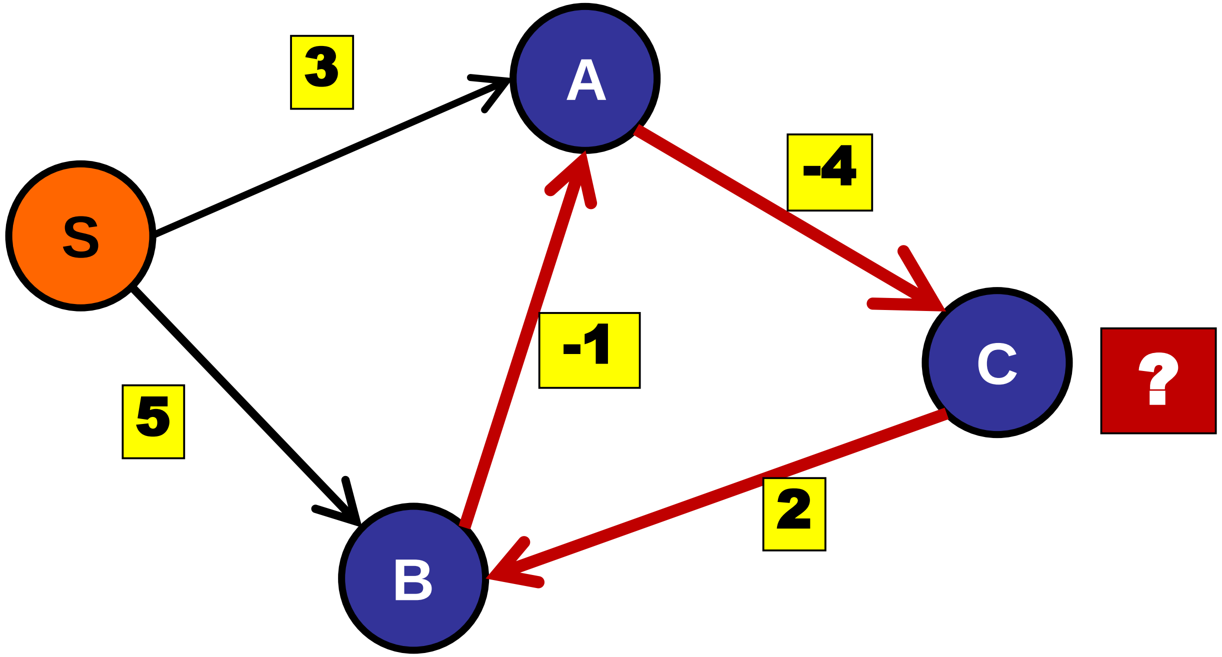
Negative weight cycles

How to detect negative weight cycles?



Negative weight cycles

How to detect negative weight cycles?

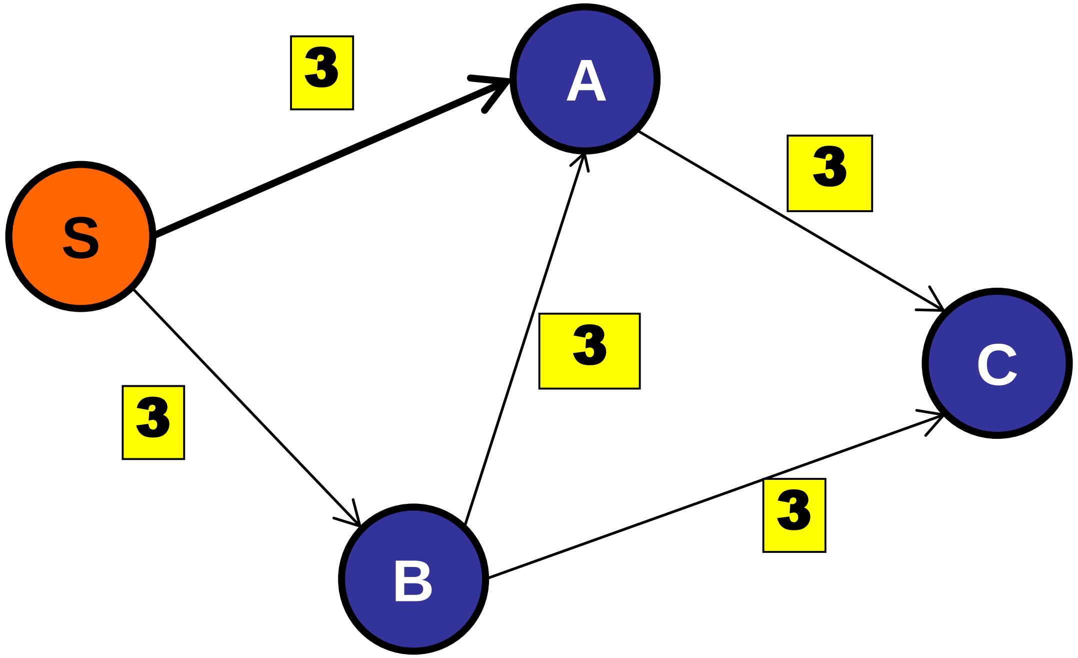


Run Bellman-Ford for |V|+1 iterations.

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

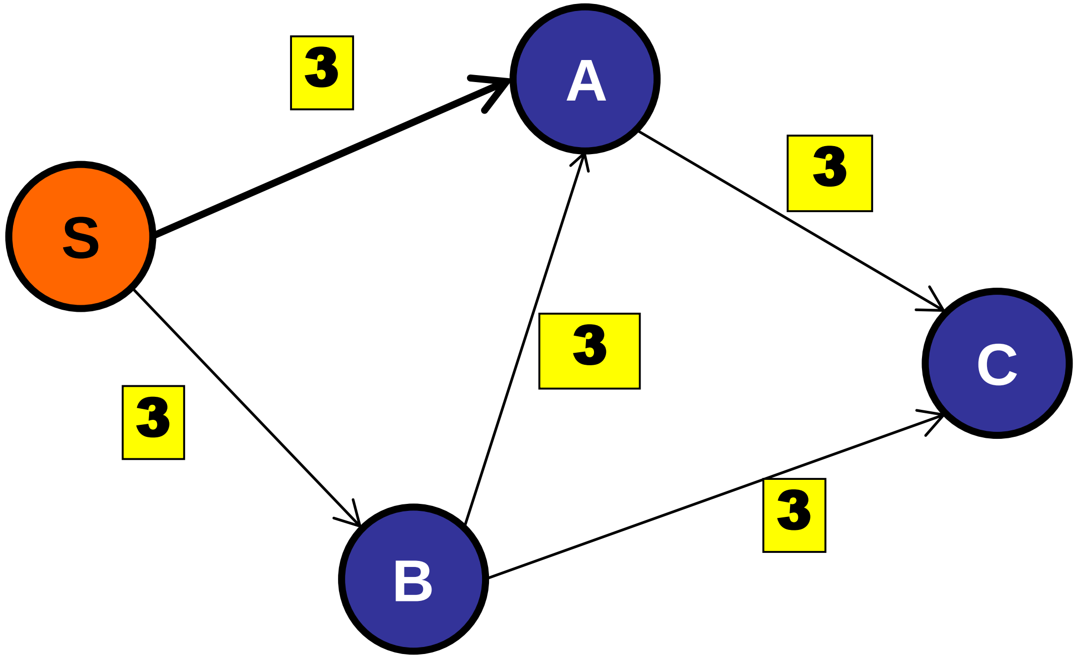
Bellman-Ford

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).

Relax edges in the right order

If there are no negative weight cycles, is there *always* a "right" order to relax the edges?

ARCHIPELAGO

is open

If so, prove it.

If not, give a counter-example.

** a "right" order is one where each edge is relaxed only once.

Faster algorithms?

Key idea:

Relax the edges in the "right" order.

Only relax each edge once:

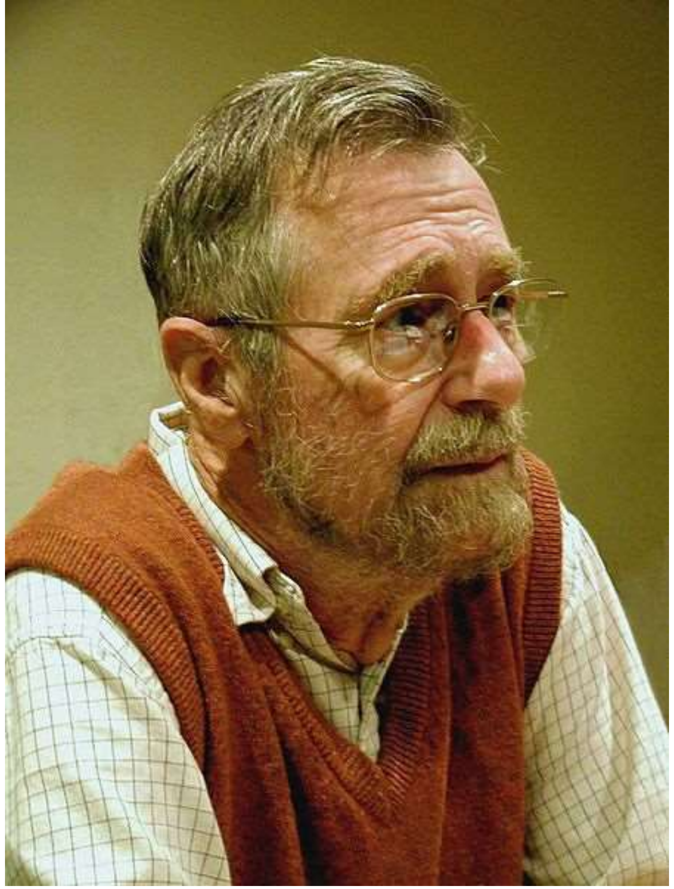
- O(E) cost (for relaxation step).

not a useful algorithm!

A right order always exists (if no neg. wt. cycles):

- Find shortest path tree.
- Relax tree edges in breadth-first order.
- Relax non-tree edges in any order.

Faster algorithms?



Key idea:

Relax the edges in the “right” order.

Only relax each edge once:

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

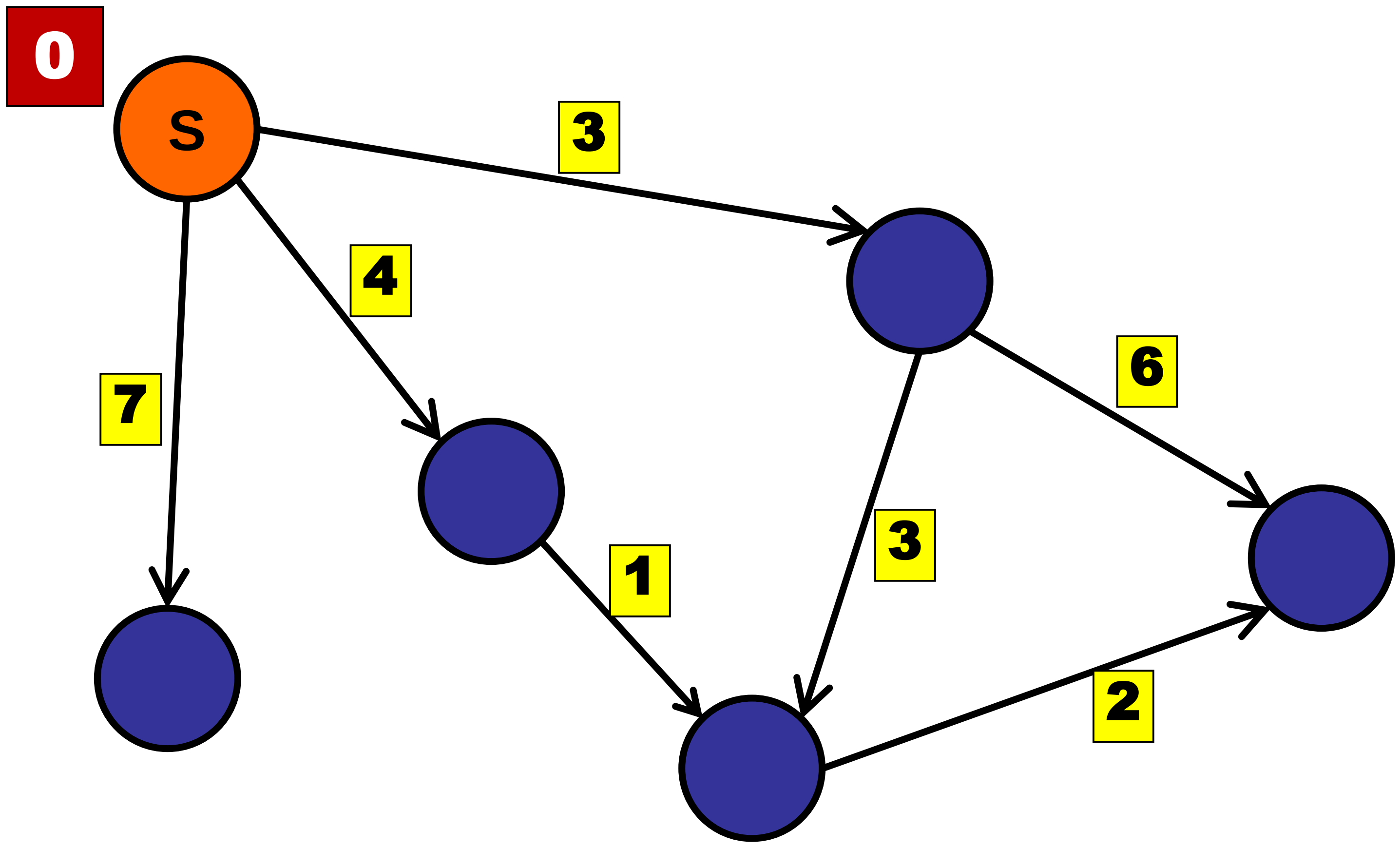
Necessary assumption:

All edges weights >= 0.

Extending a path does not make it shorter!

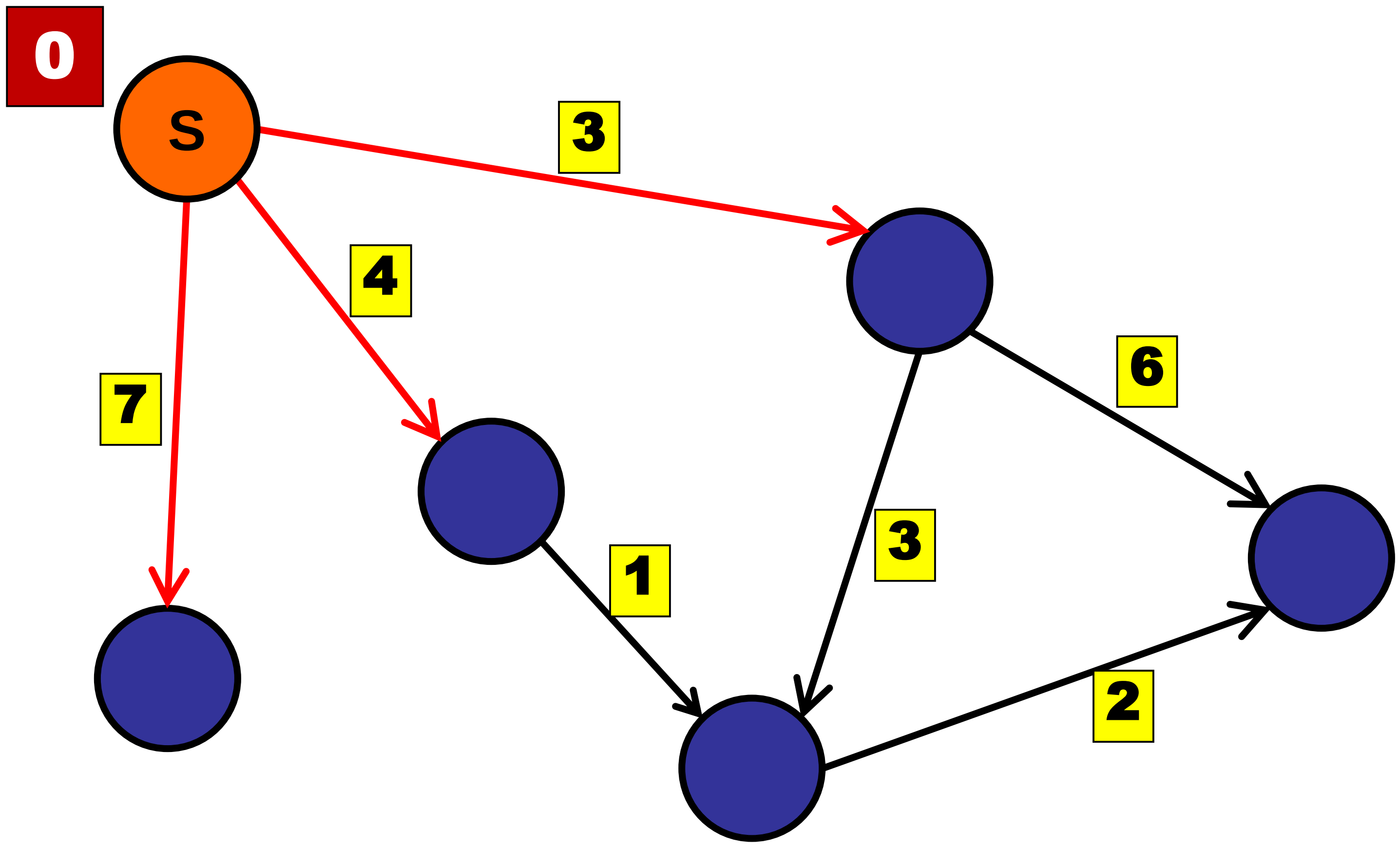
Dijkstra’s Algorithm (First Try)

Relax shortest edge first



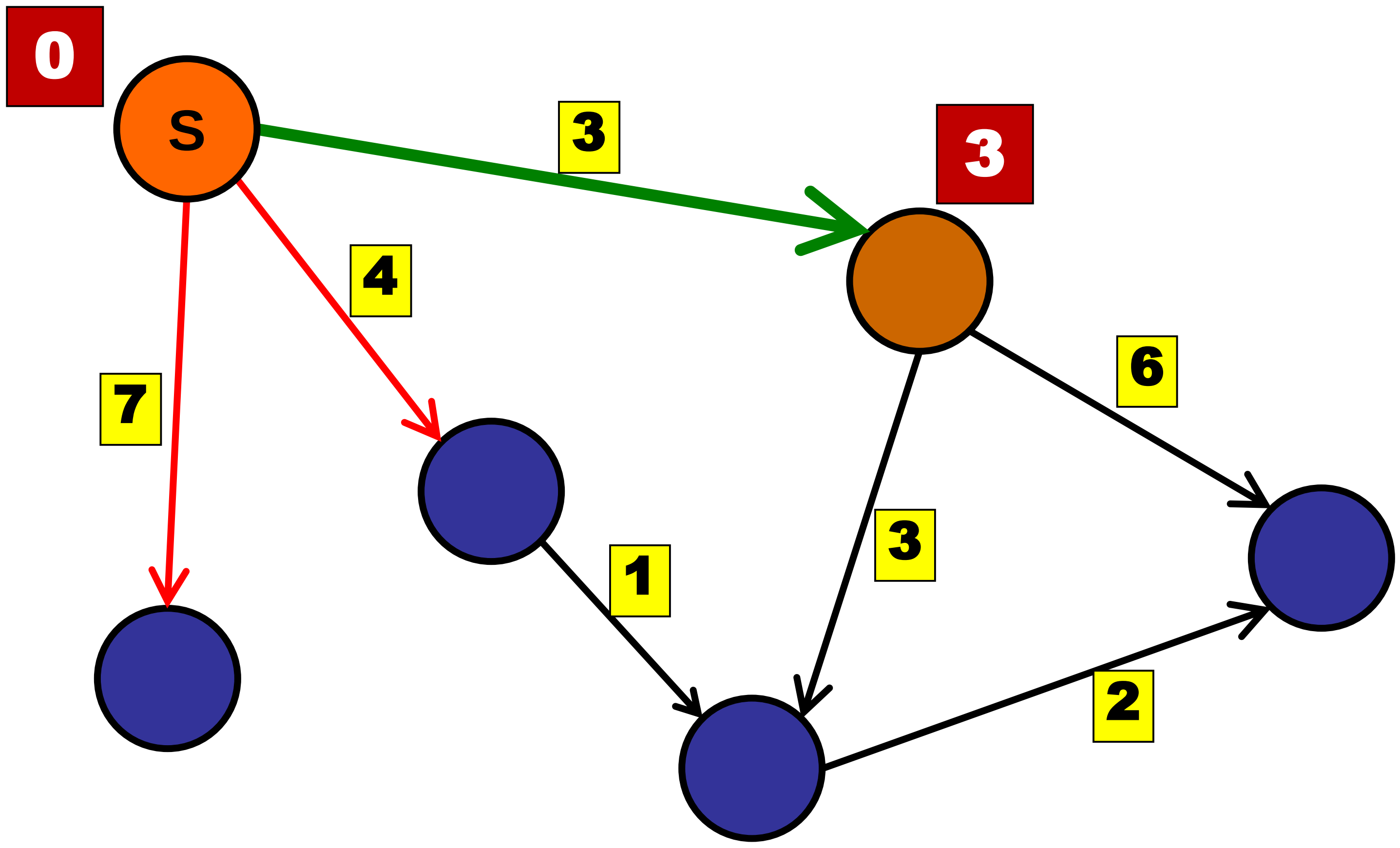
Dijkstra’s Algorithm (First Try)

Relax shortest edge first



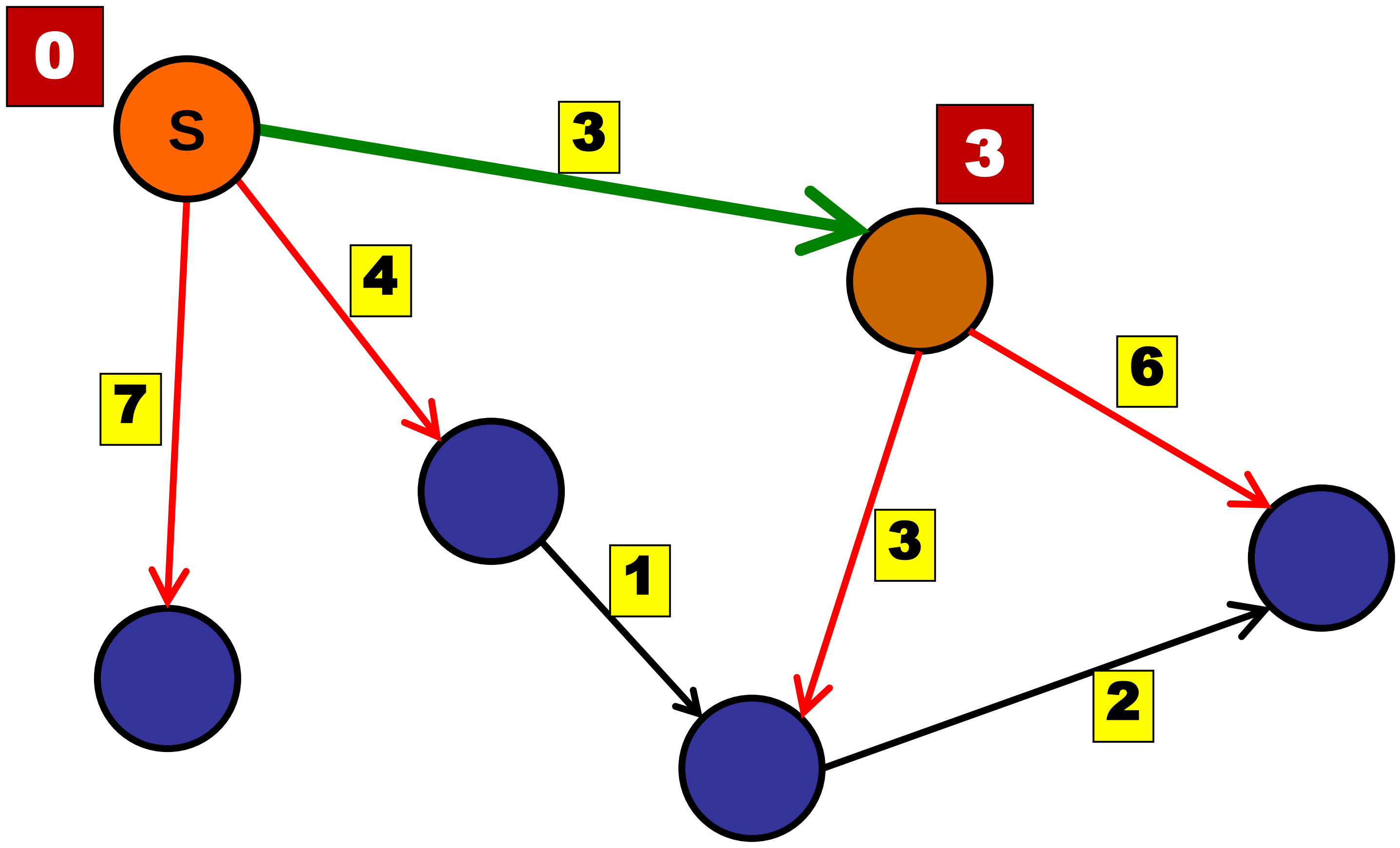
Dijkstra’s Algorithm (First Try)

Relax shortest edge first



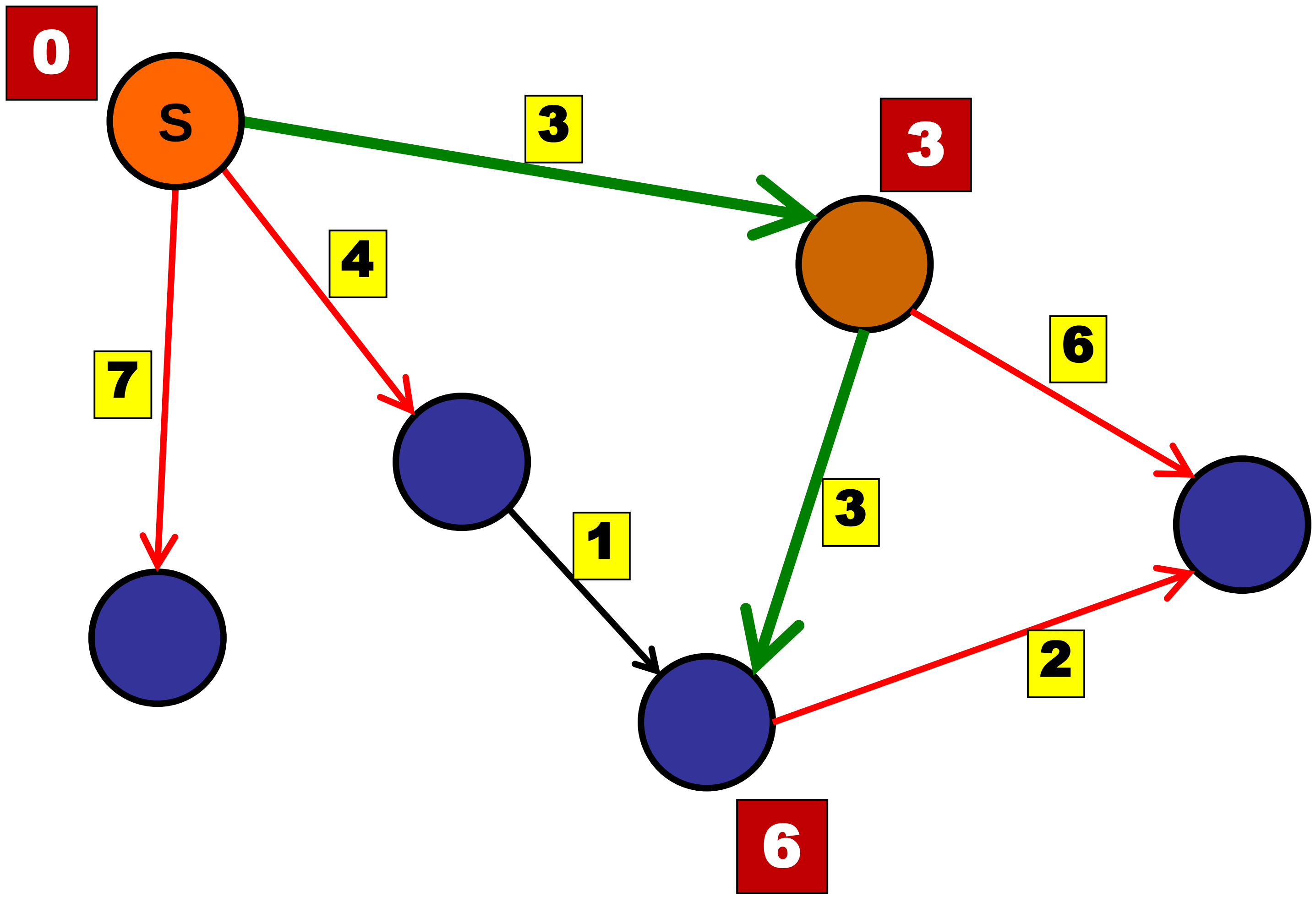
Dijkstra's Algorithm (First Try)

Relax shortest edge first



Dijkstra's Algorithm (First Try)

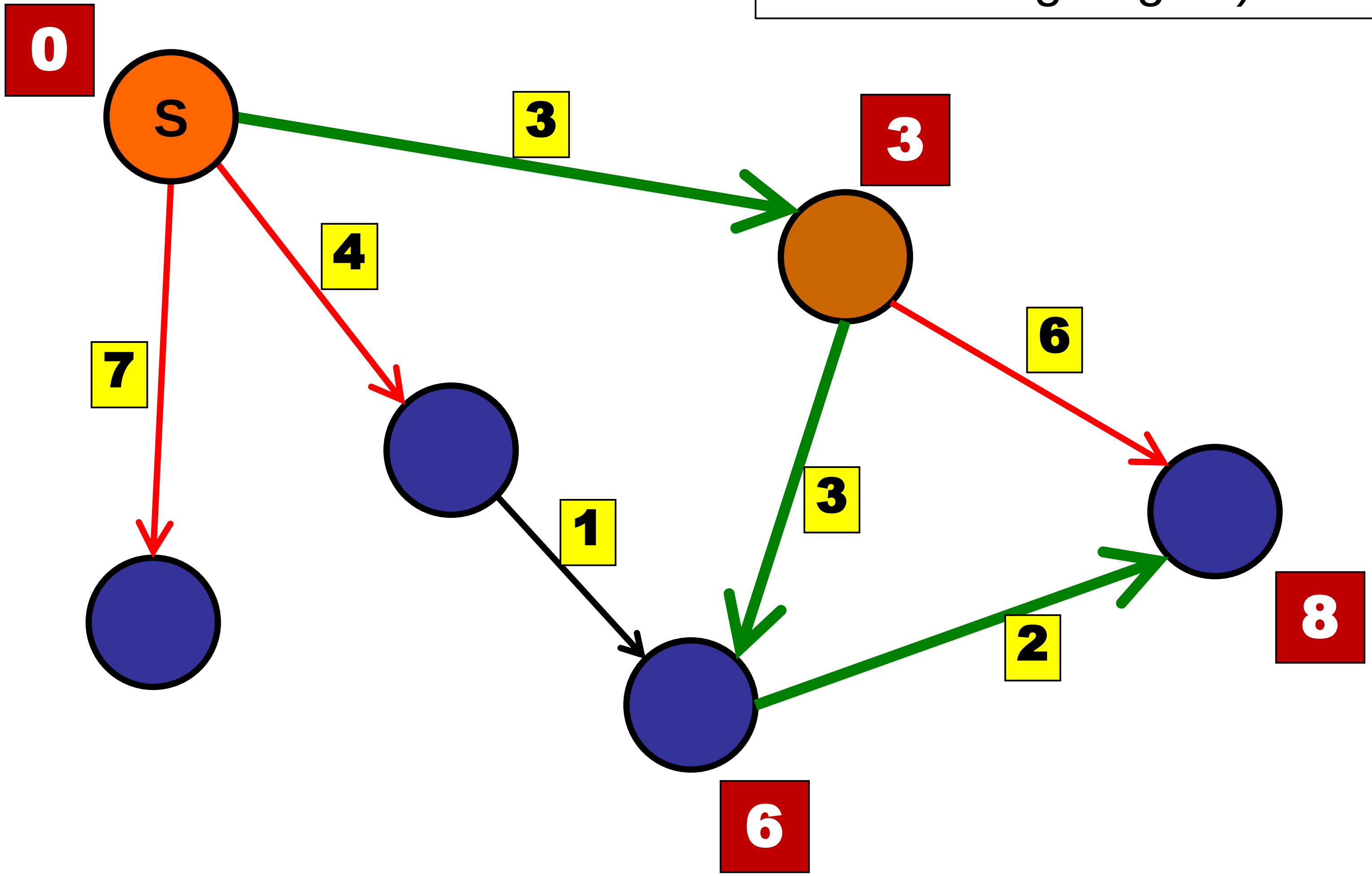
Relax shortest edge first



Dijkstra’s Algorithm (Failed Try)

Oops….

Only relax an edge once its estimate is correct (and will never change again)!

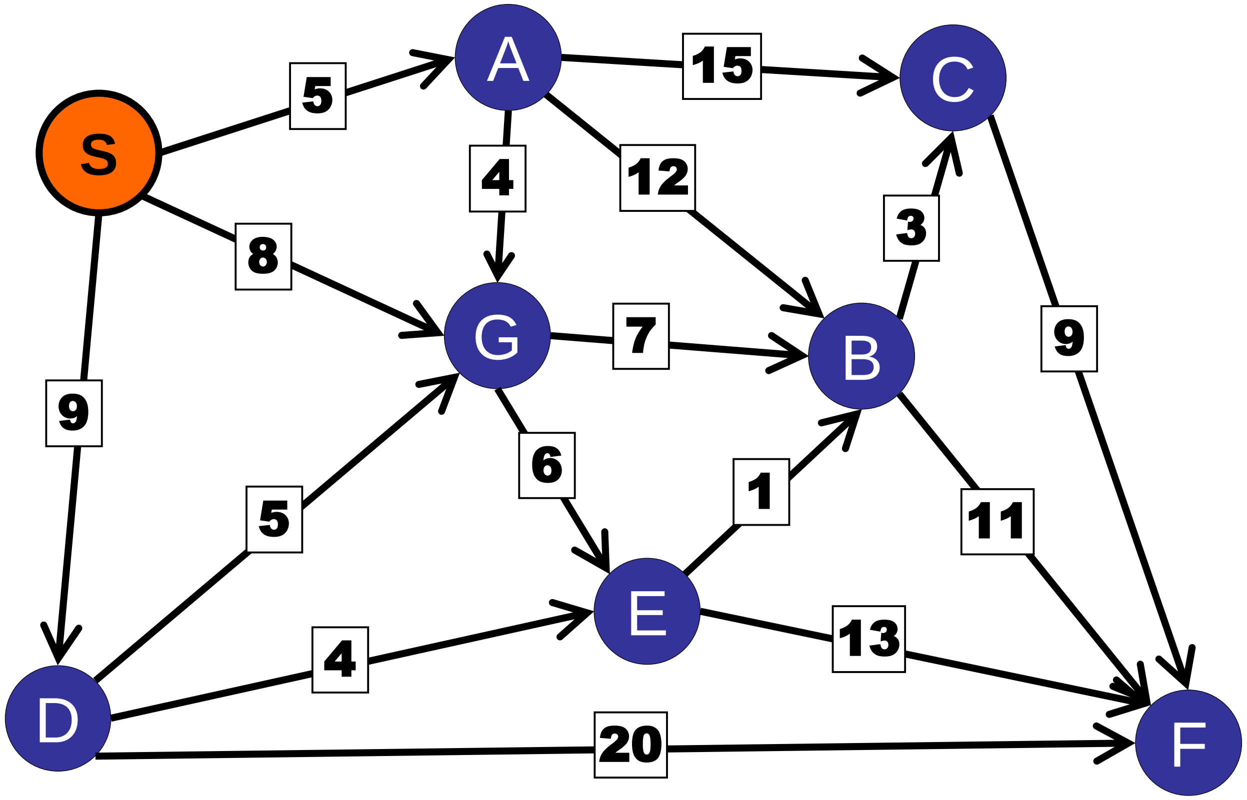


Dijkstra's Algorithm

Basic idea:

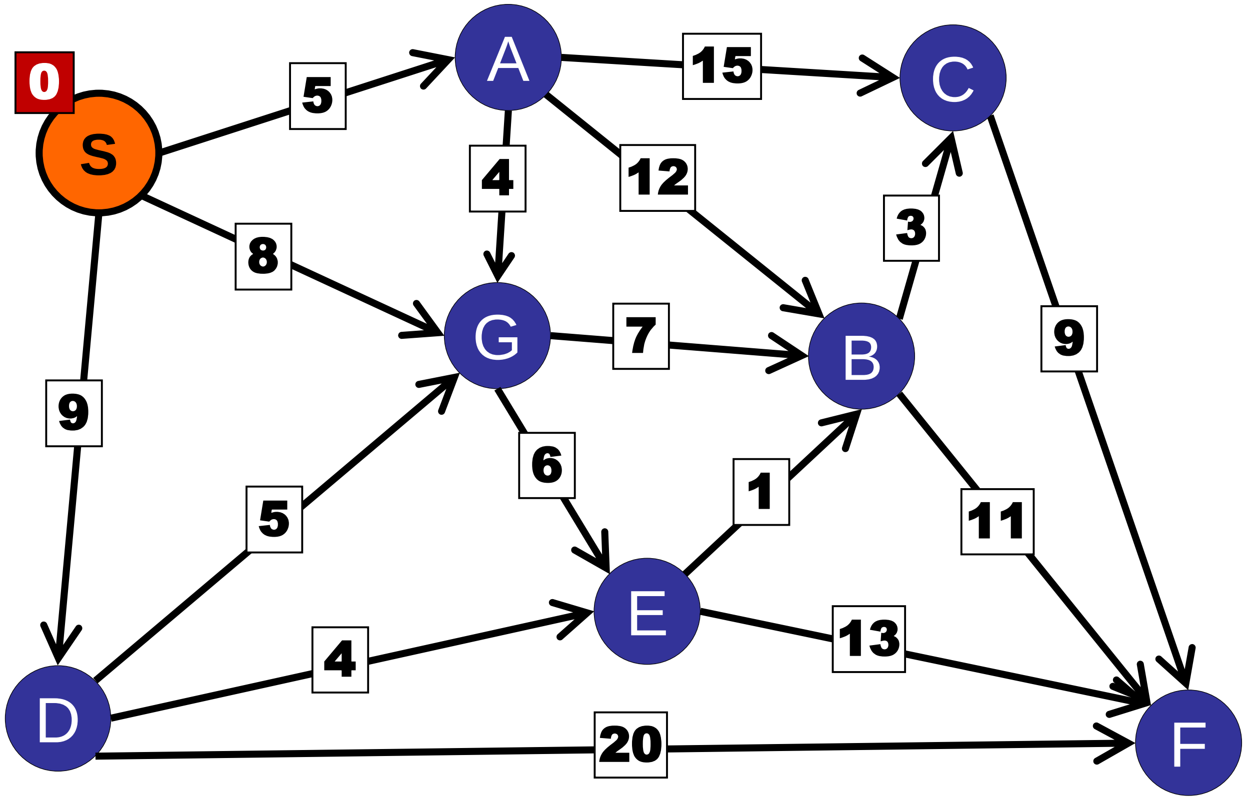
- Maintain distance estimate for every node.
- Begin with empty shortest-path-tree.
- Repeat:
  - Consider **node** with minimum estimate.
  - (We will show that this node has a good estimate.)
  - Add node to shortest-path-tree.
  - Relax all outgoing edges.

Shortest Paths



Dijkstra's Algorithm

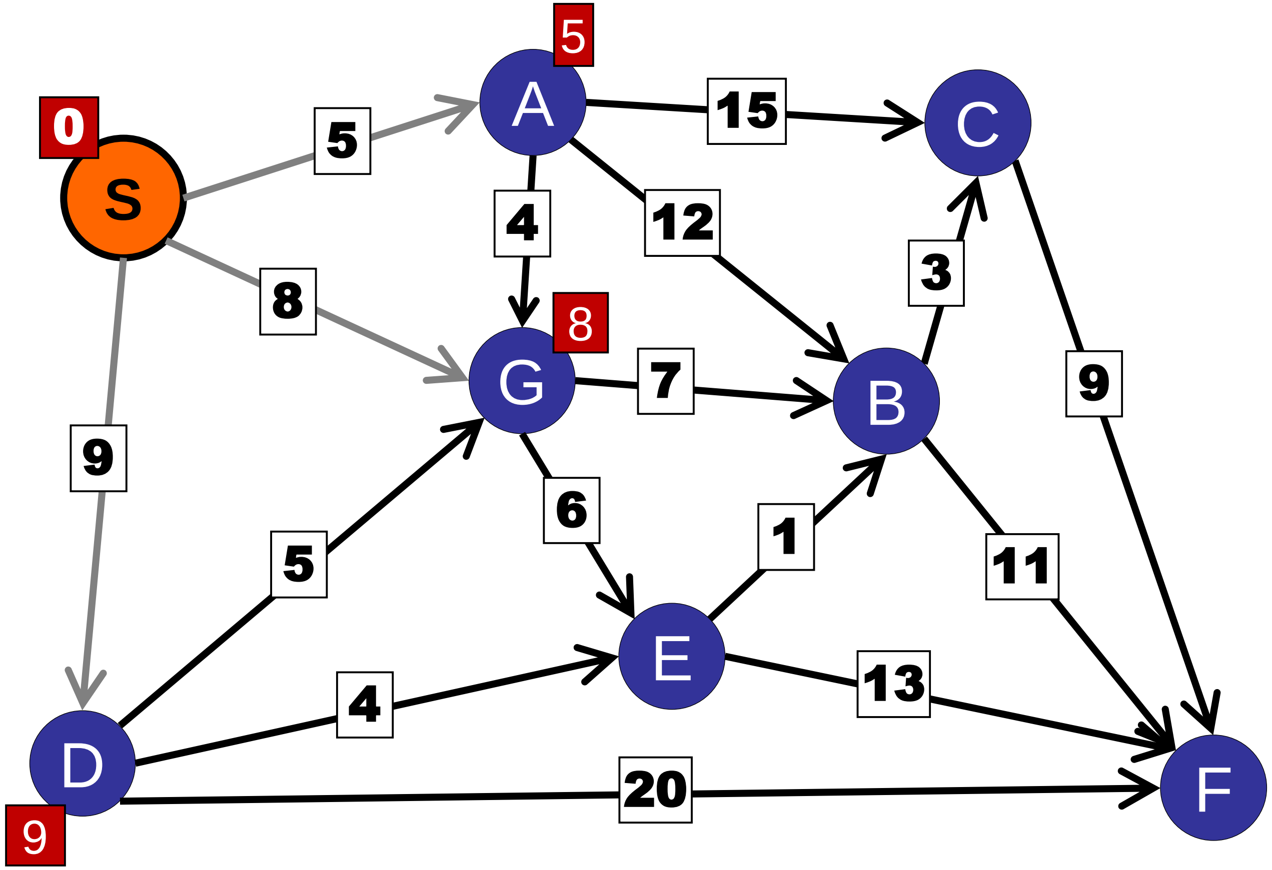
Step 1: Add source



Vertex	Dist.
S	0

Dijkstra's Algorithm

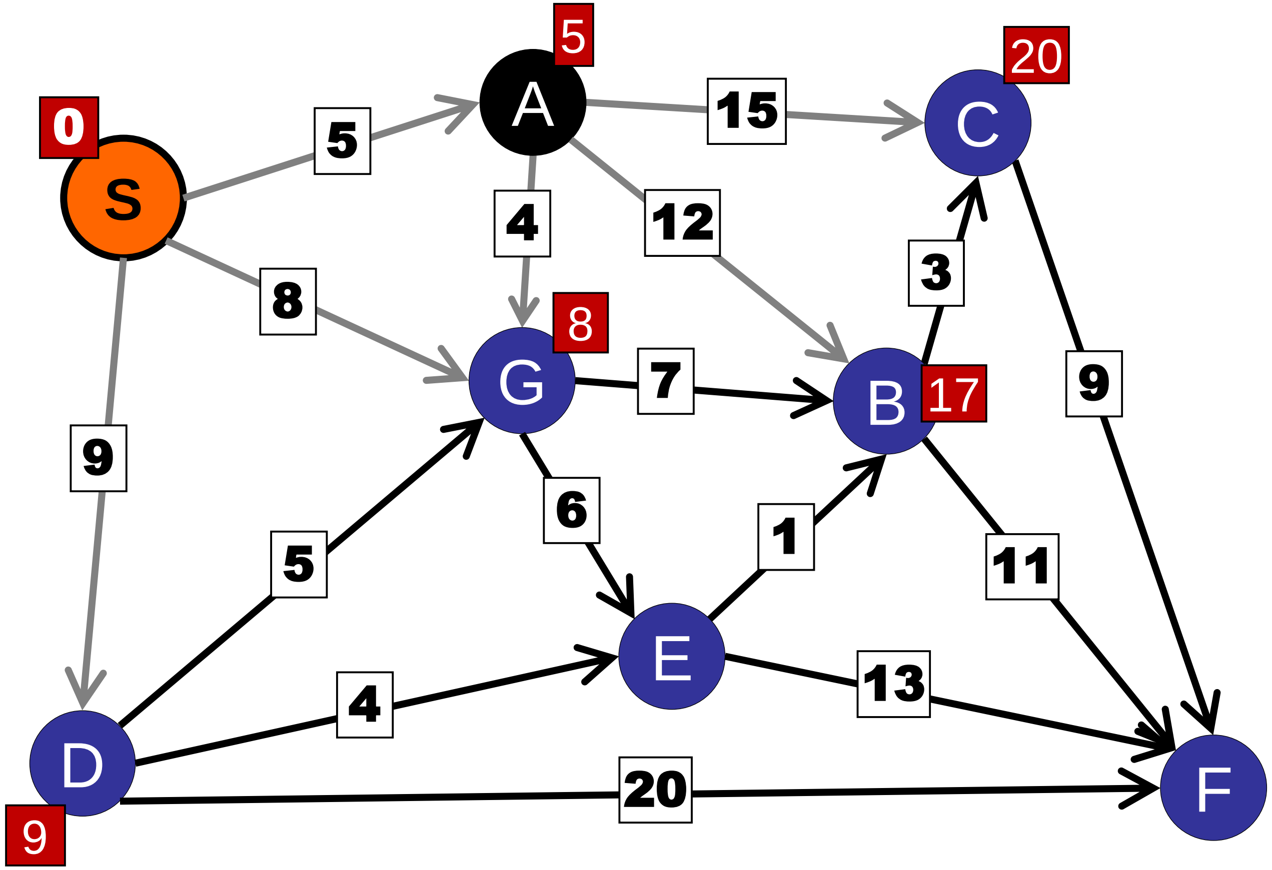
Step 2: Remove S and relax.



Vertex	Dist.
A	5
G	8
D	9

Dijkstra's Algorithm

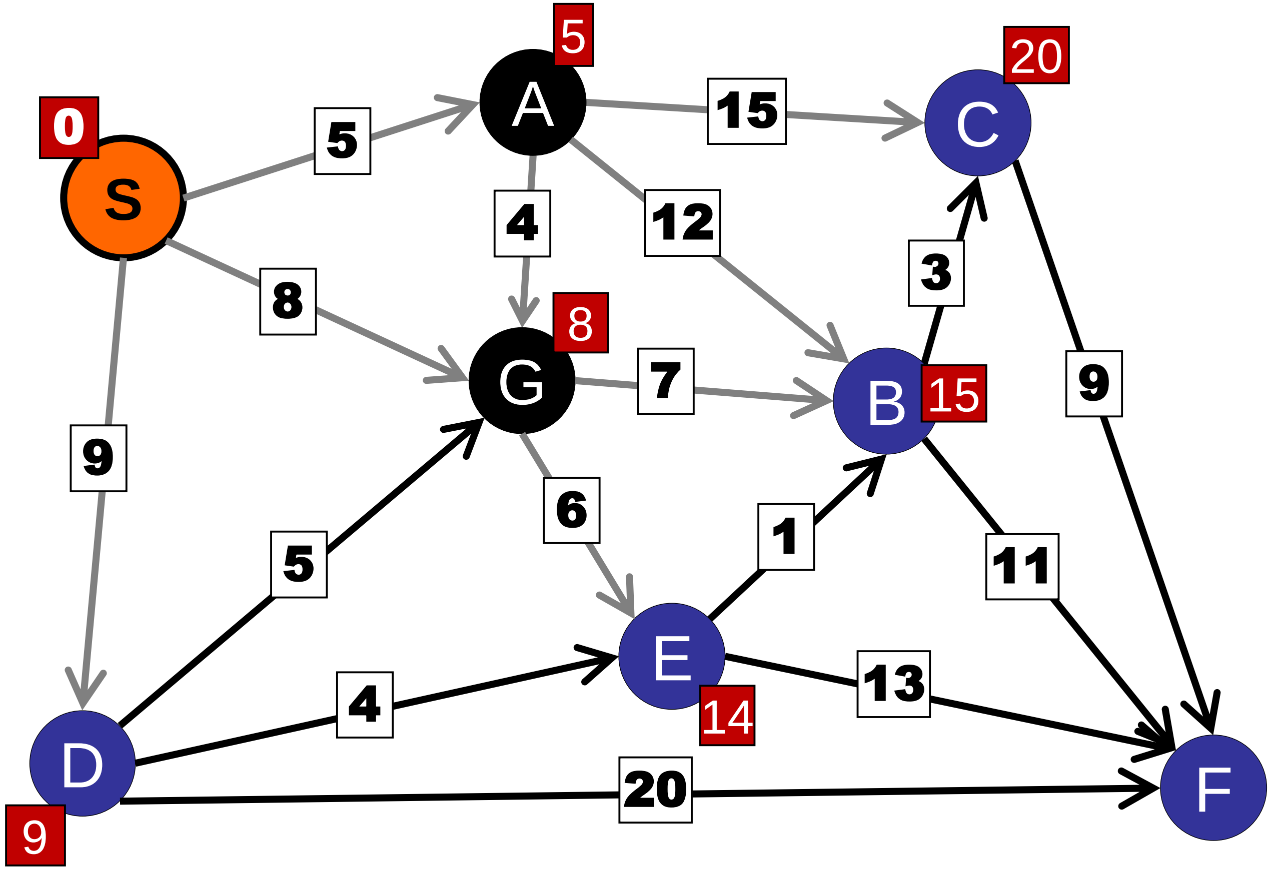
Step 3: Remove A and relax.



Vertex	Dist.
G	8
D	9
B	17
C	20

Dijkstra's Algorithm

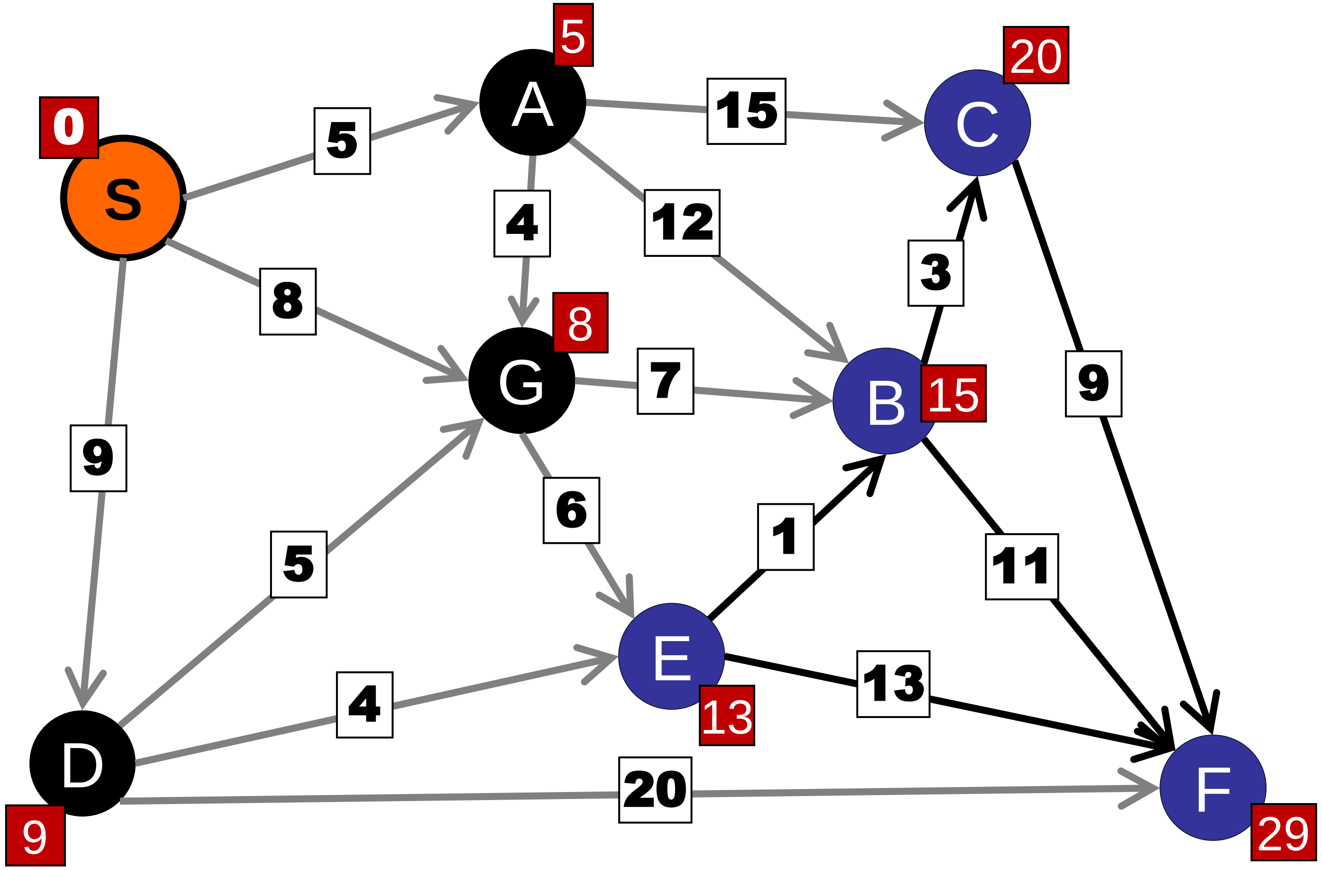
Step 4: Remove G and relax.



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

Dijkstra's Algorithm

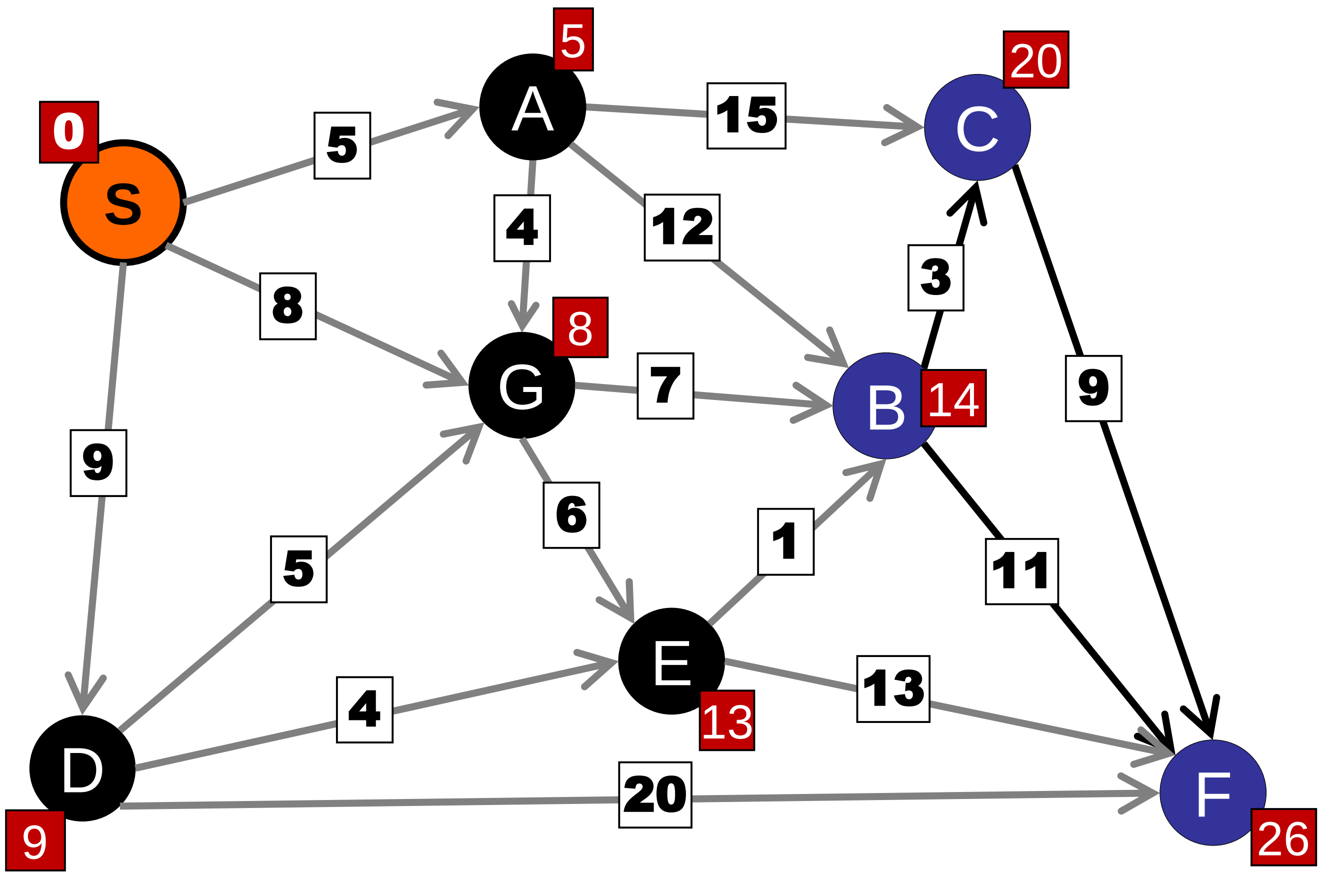
Step 5: Remove D and relax.



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

Dijkstra's Algorithm

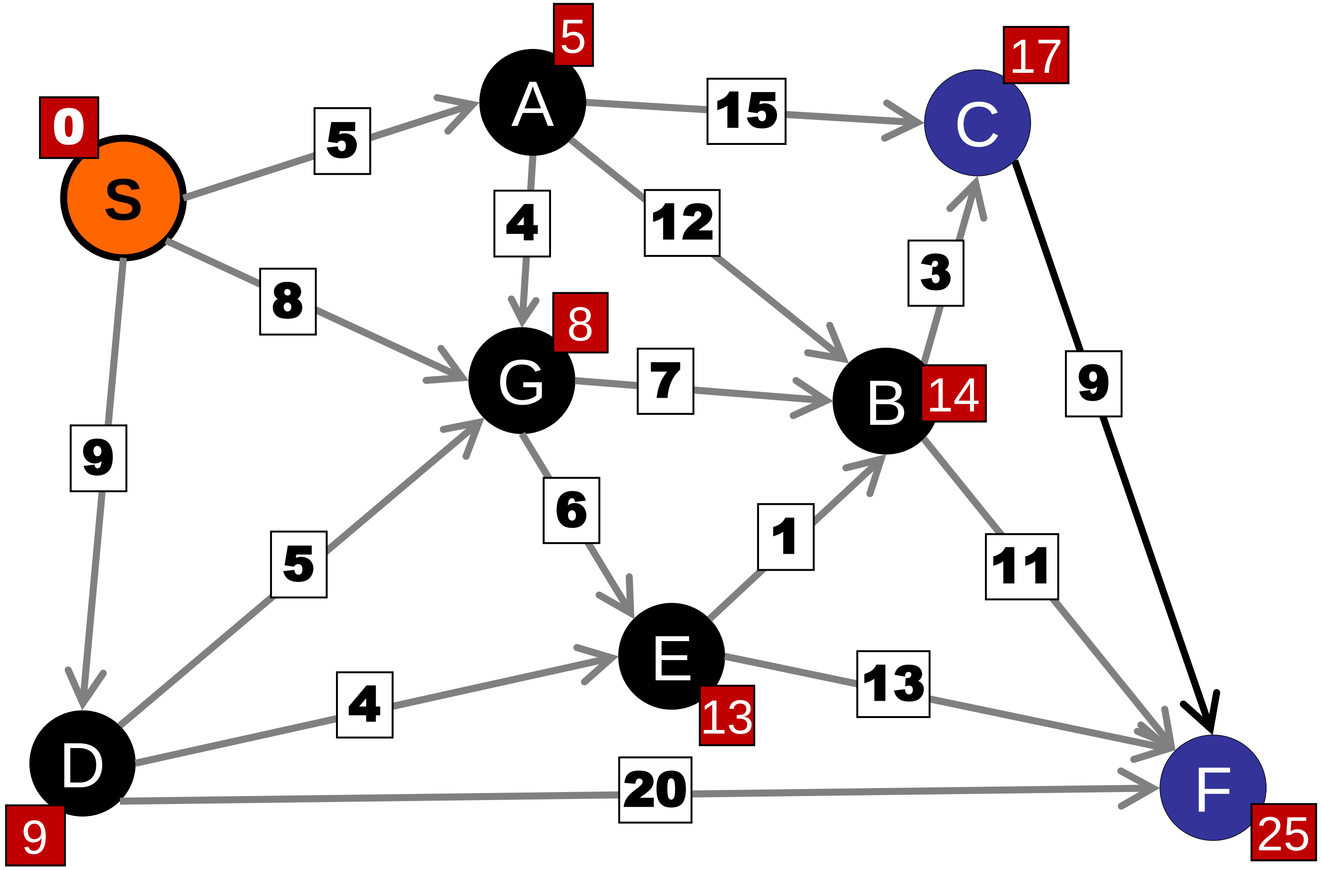
Step 5: Remove E and relax.



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

Dijkstra’s Algorithm

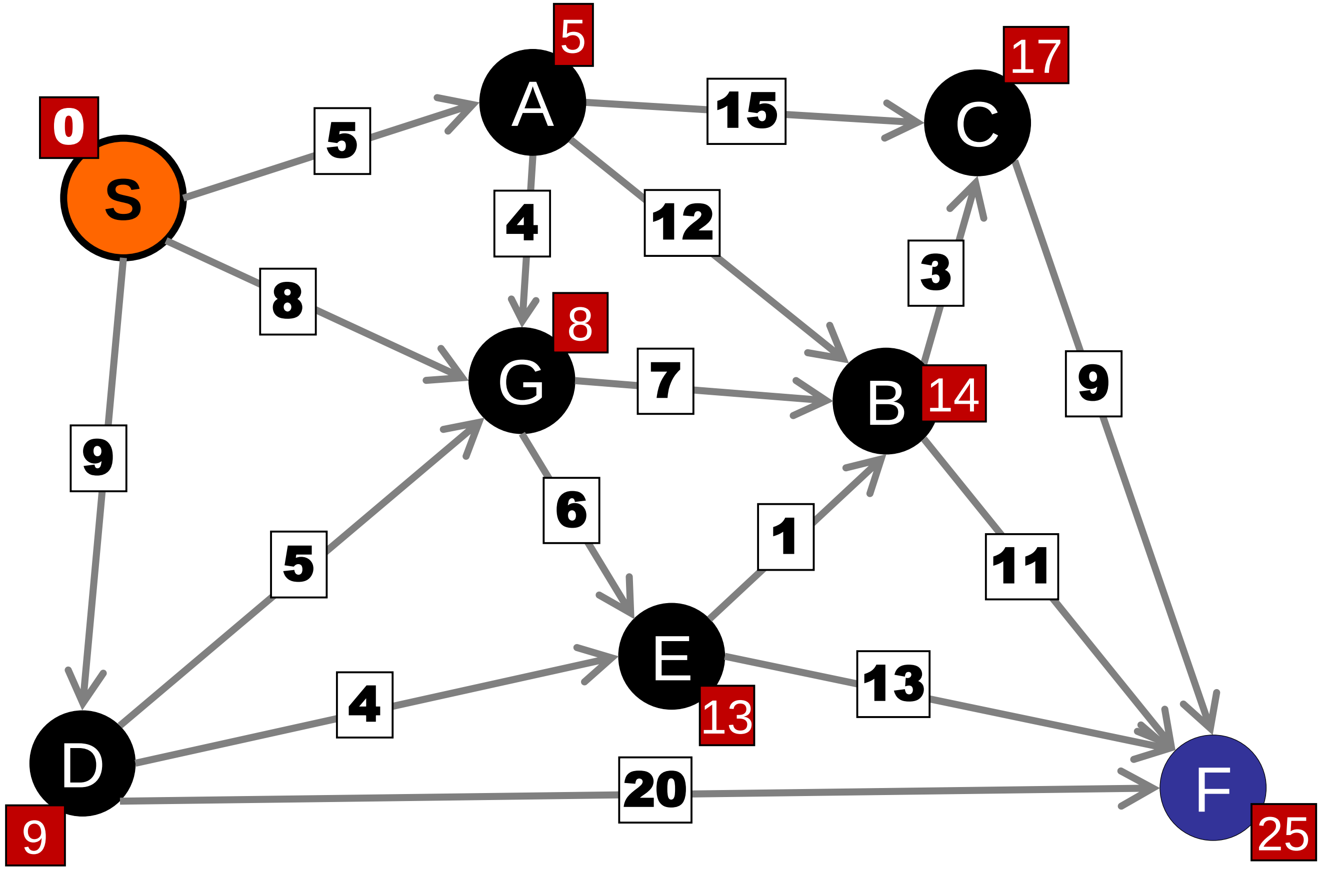
Step 5: Remove B and relax.



Vertex	Dist.
C	20
F	25

Dijkstra's Algorithm

Step 5: Remove C and relax.

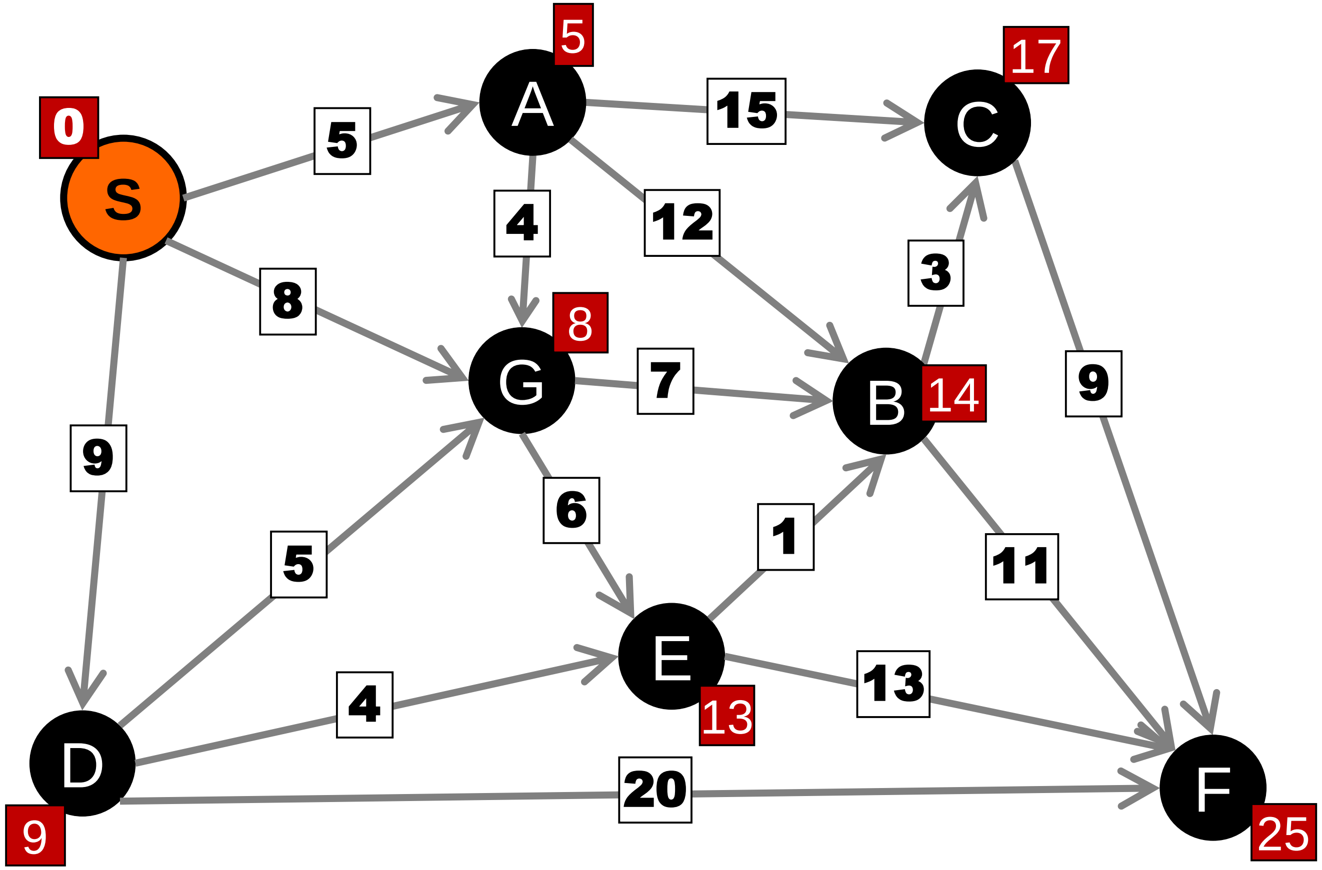


Vertex	Dist.
F	25

Dijkstra’s Algorithm

Step 5: Remove F and relax.

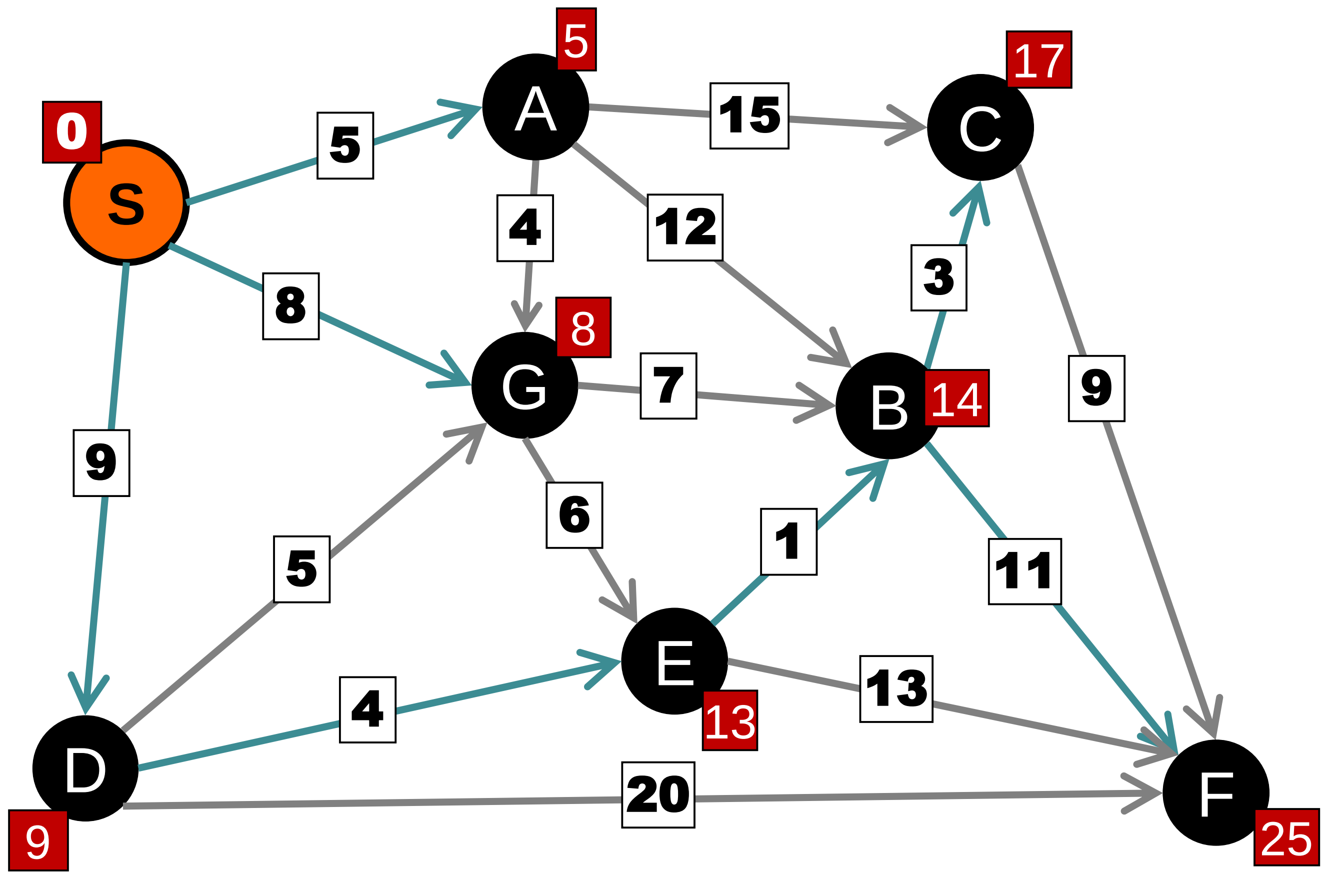
Vertex	Dist.



Dijkstra's Algorithm

Done

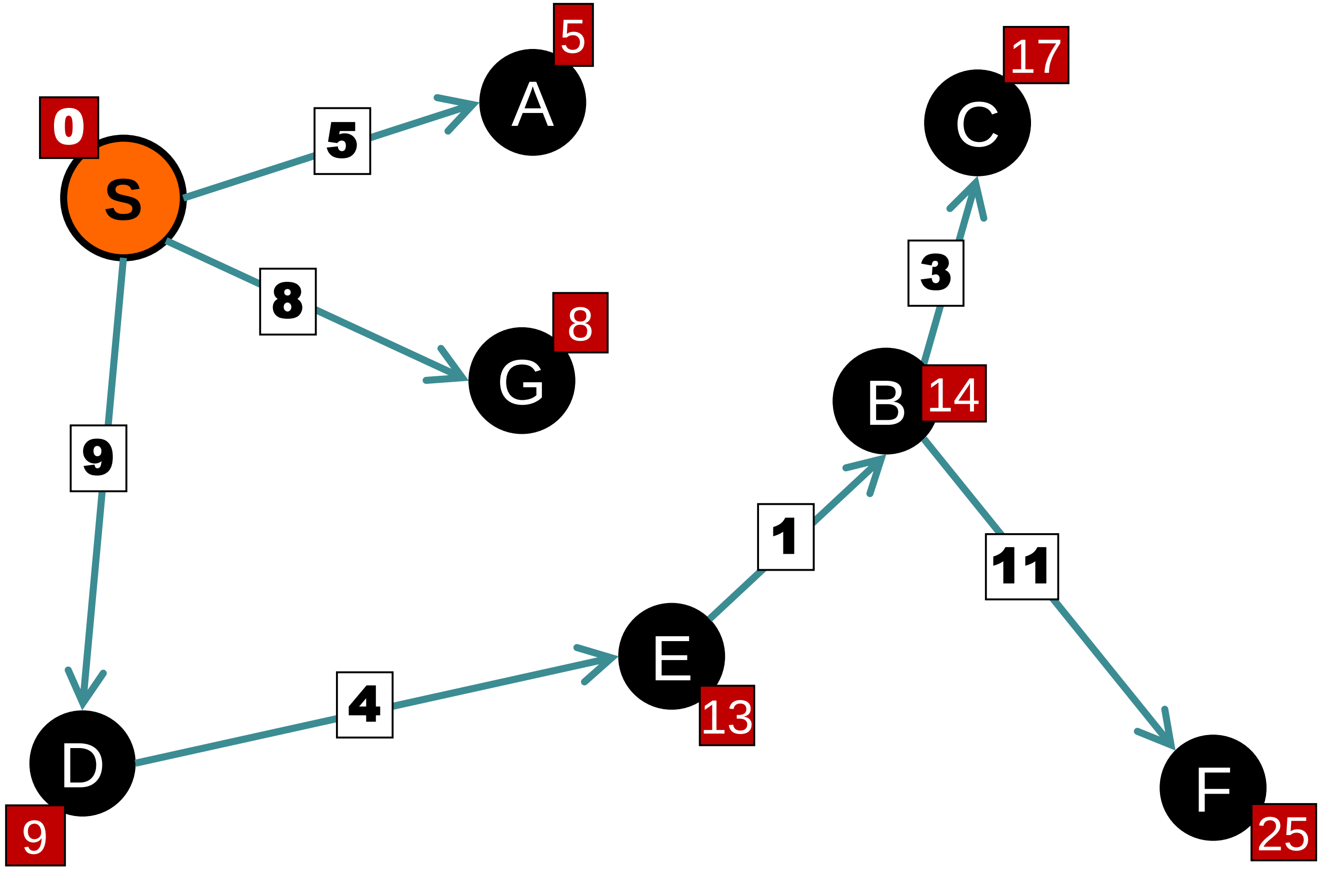
Vertex	Dist.



Dijkstra’s Algorithm

Shortest Path Tree

Vertex	Dist.



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?*

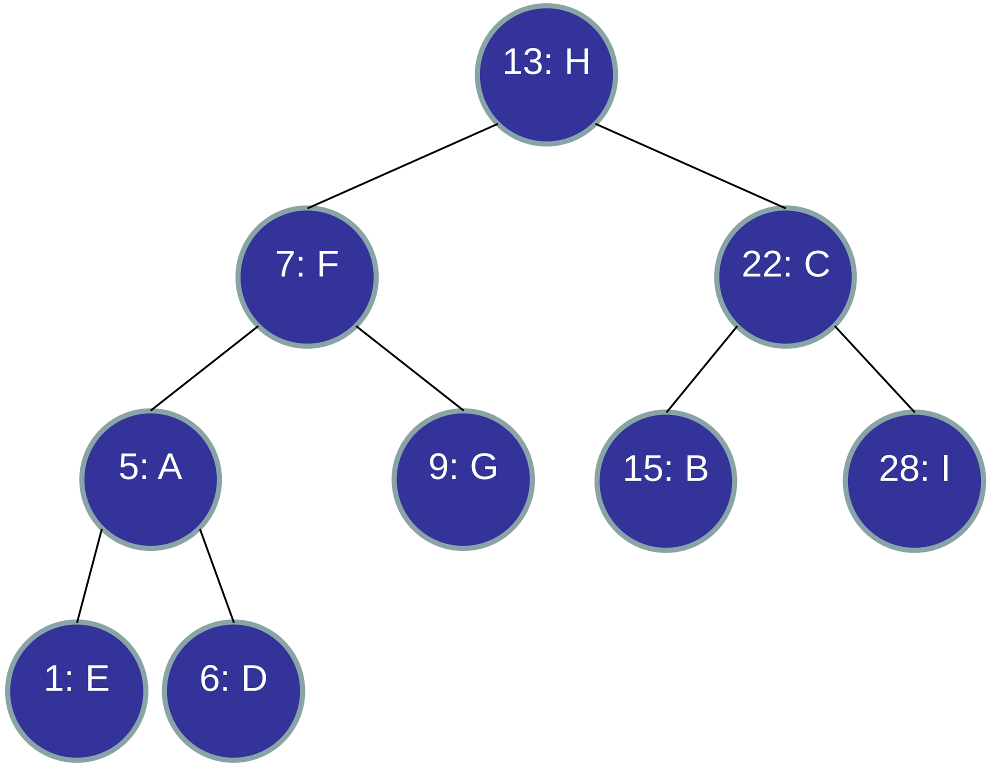
<u>Notes:</u>

Assume data items are unique.

Priority Queue

AVL Tree

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

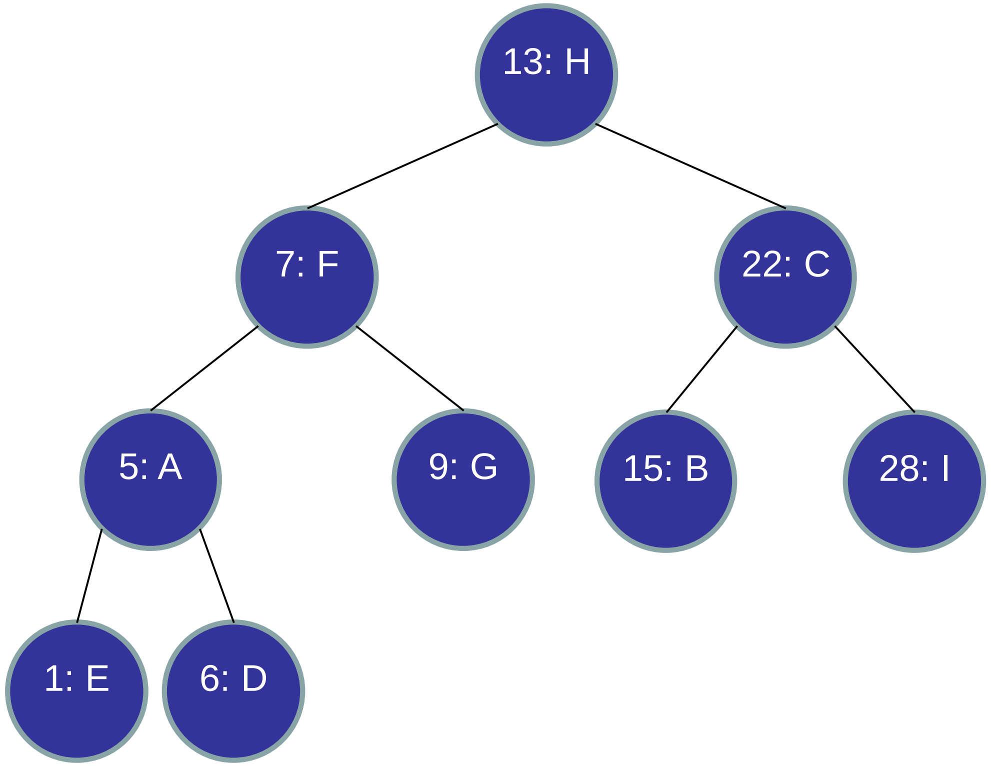


Priority Queue

AVL Tree

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

- How to find a vertex?

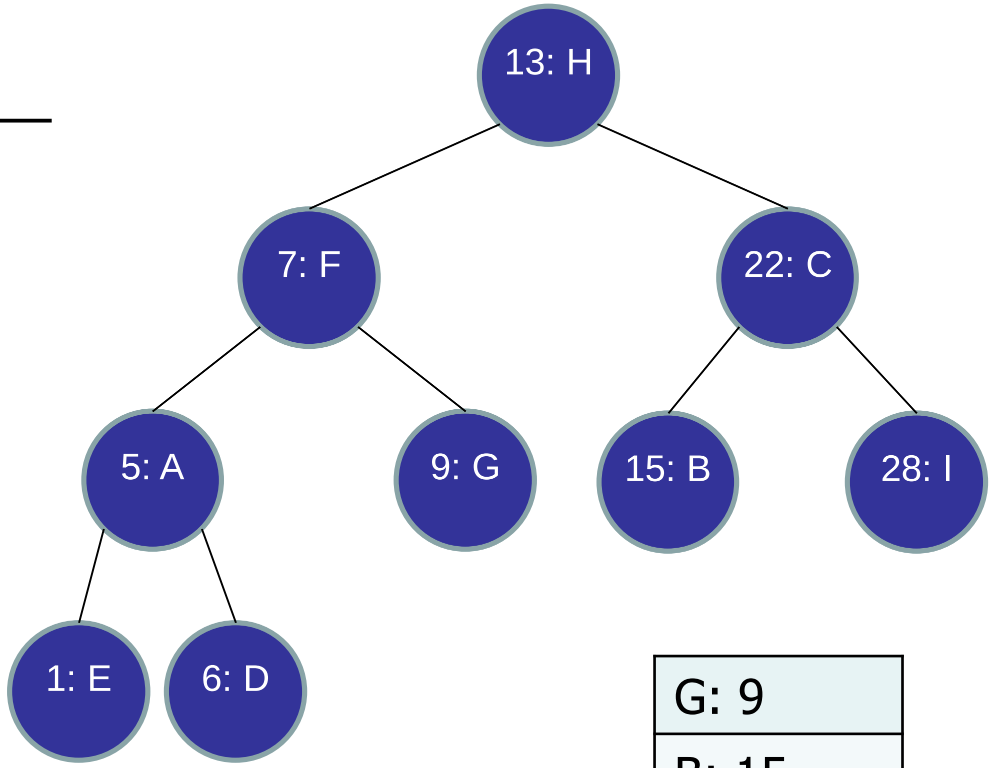


Priority Queue

AVL Tree

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

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



G: 9
B: 15
H: 13
C: 22
I: 28
A: 5
D: 6
E: 1
F: 7

Dijkstra’s Algorithm

Priority Queue by AVL tree:

- insert(`key, priority`): O(log n)
  - Insert (`priority, key`) in AVL tree indexed by `priority`
  - Insert (`key, priority`) in hash table
- deleteMin(): O(log n)
  - Find node with the minimum priority and delete it from AVL tree
- decreaseKey(`key, priority`): O(log n)
  - Find current priority (`curPri`) of key in hash table, remove (`curPri,key`) from AVL tree, insert (`priority, key`) into AVL tree, update hash table record for `key`
- contains(key): O(1)
  - Search in the hash table for `key`

Dijkstra's Algorithm

Priority Queue by AVL tree:

- insert(`key, priority`): O(log n)
  - Insert (`priority, key`) in AVL tree indexed b
  - Insert (`key, priority`) in hash table
- deleteMin(): O(log n)
  - Find node with the minimum priority and delete
- decreaseKey(`key, priority`): O(log n)
  - Find current priority (`curPri`) of key in hash table, remove (`curPri,key`) from AVL tree, insert (`priority, key`) into AVL tree, update hash table record for `key`
- contains(key): O(1)
  - Search in the hash table for `key`

What if there are multiple keys with same priority? Possible approaches:

- Put all keys with same priority in the same node in AVL tree
- Have distinct nodes in AVL tree for different keys, but in hash table, include pointer to the particular node for each key.

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

ARCHIPELAGO

is open

```
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:

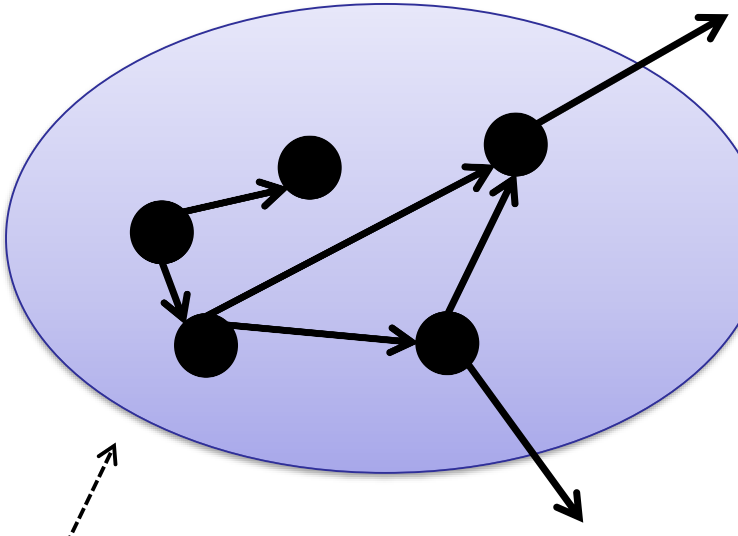
- insert / deleteMin: |V| times each
  - Each node is added to the priority queue **once**.
- relax / decreaseKey: |E| times
  - Each edge is relaxed once.
- Priority queue operations: O(log V)
- Total: O((V+E)log V) = O(E log V)

Dijkstra’s Algorithm

Why does it work?

Dijkstra’s Algorithm

Every edge crossing the boundary has been relaxed.

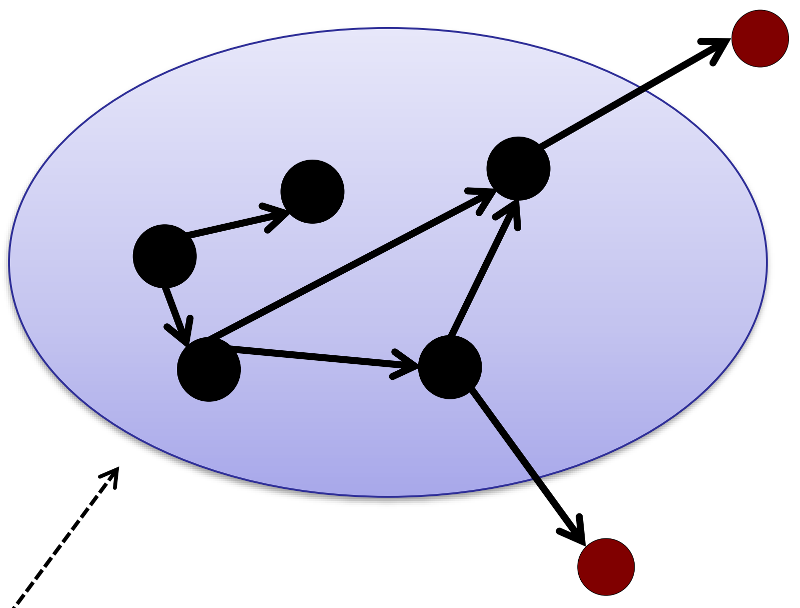


finished vertices:
distance is accurate.
Initially: just the source.

Dijkstra’s Algorithm

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

Every edge crossing the boundary has been relaxed.



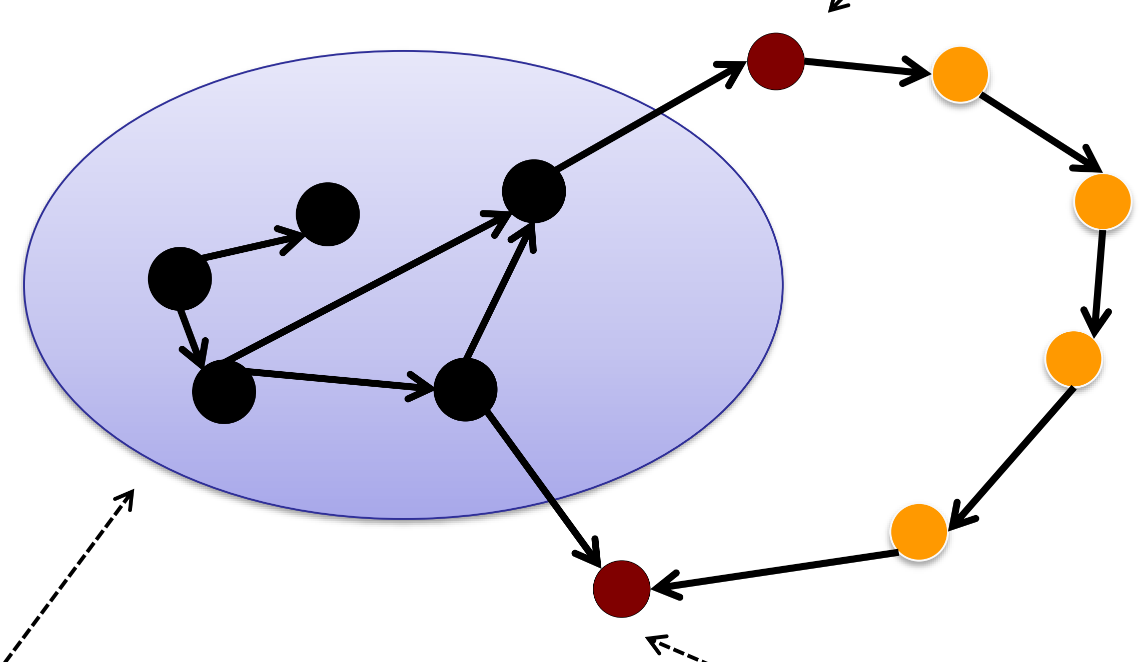
**finished vertices:
distance is accurate.**

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

Dijkstra's Algorithm

fringe vertices:
neighbor of a
finished vertex.

Every edge crossing the boundary has been relaxed.



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

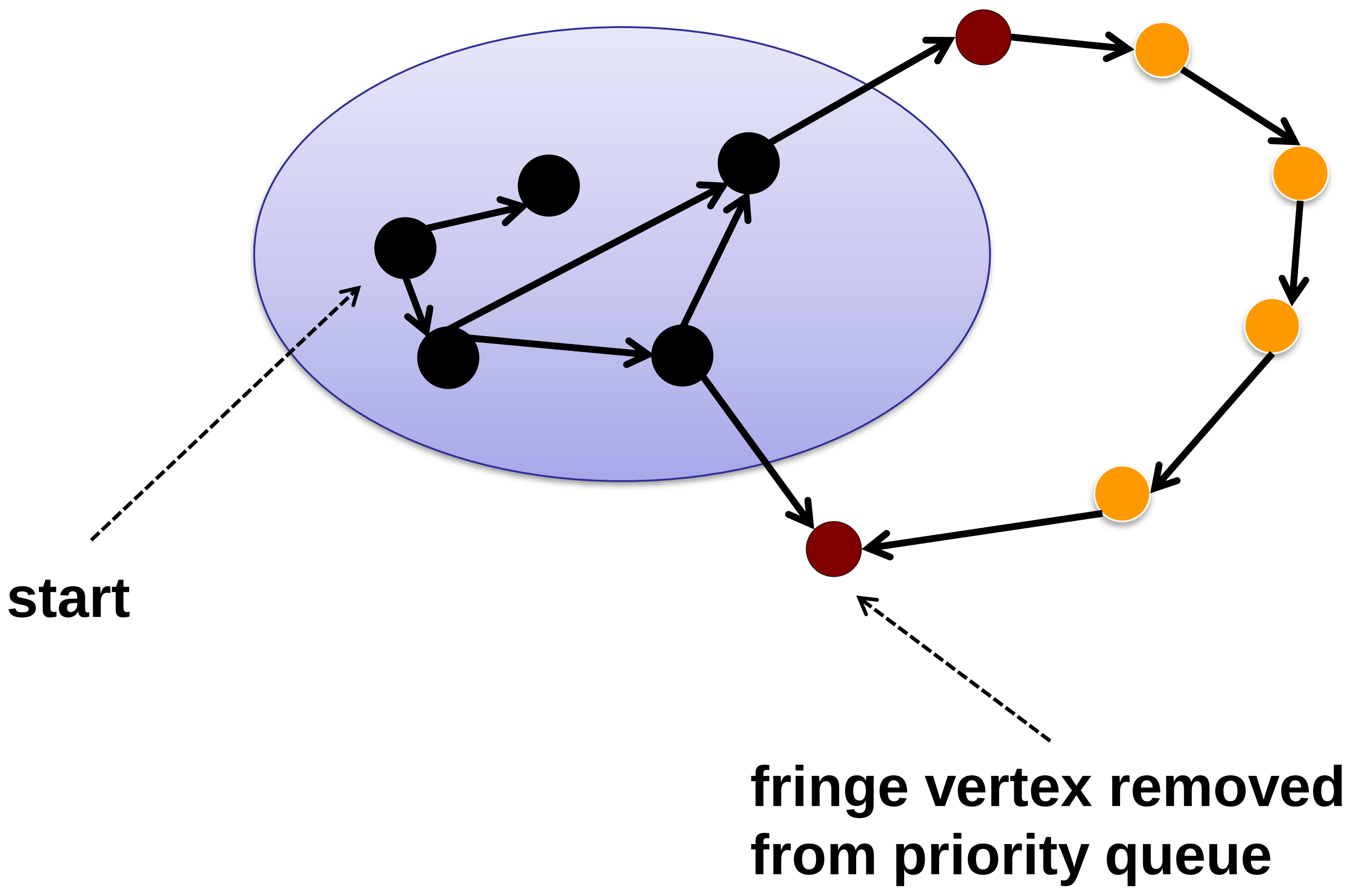
Proof by induction:

- 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.**

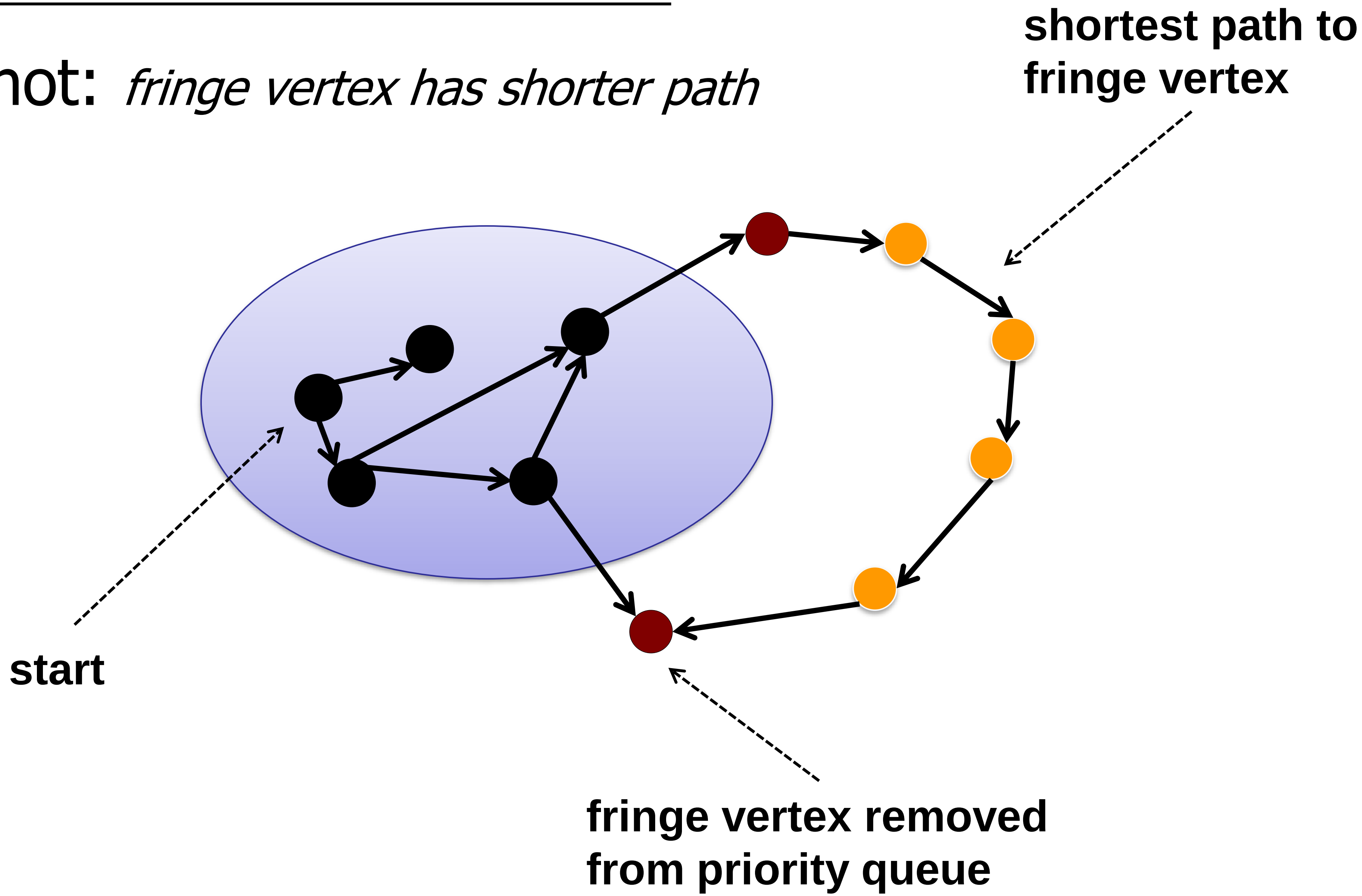
Dijkstra's Algorithm

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



Dijkstra's Algorithm

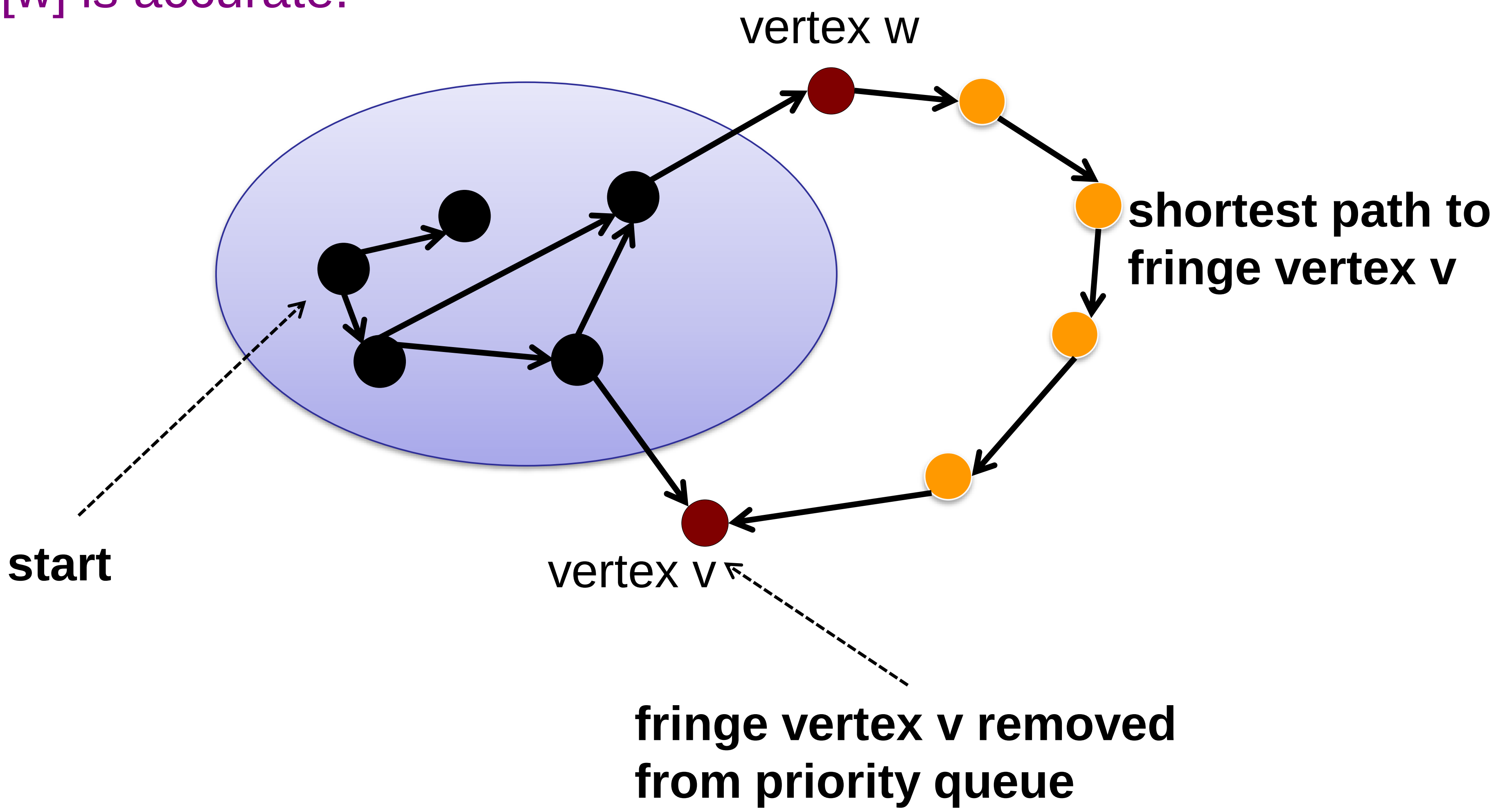
Assume not: *fringe vertex has shorter path*



Dijkstra's Algorithm

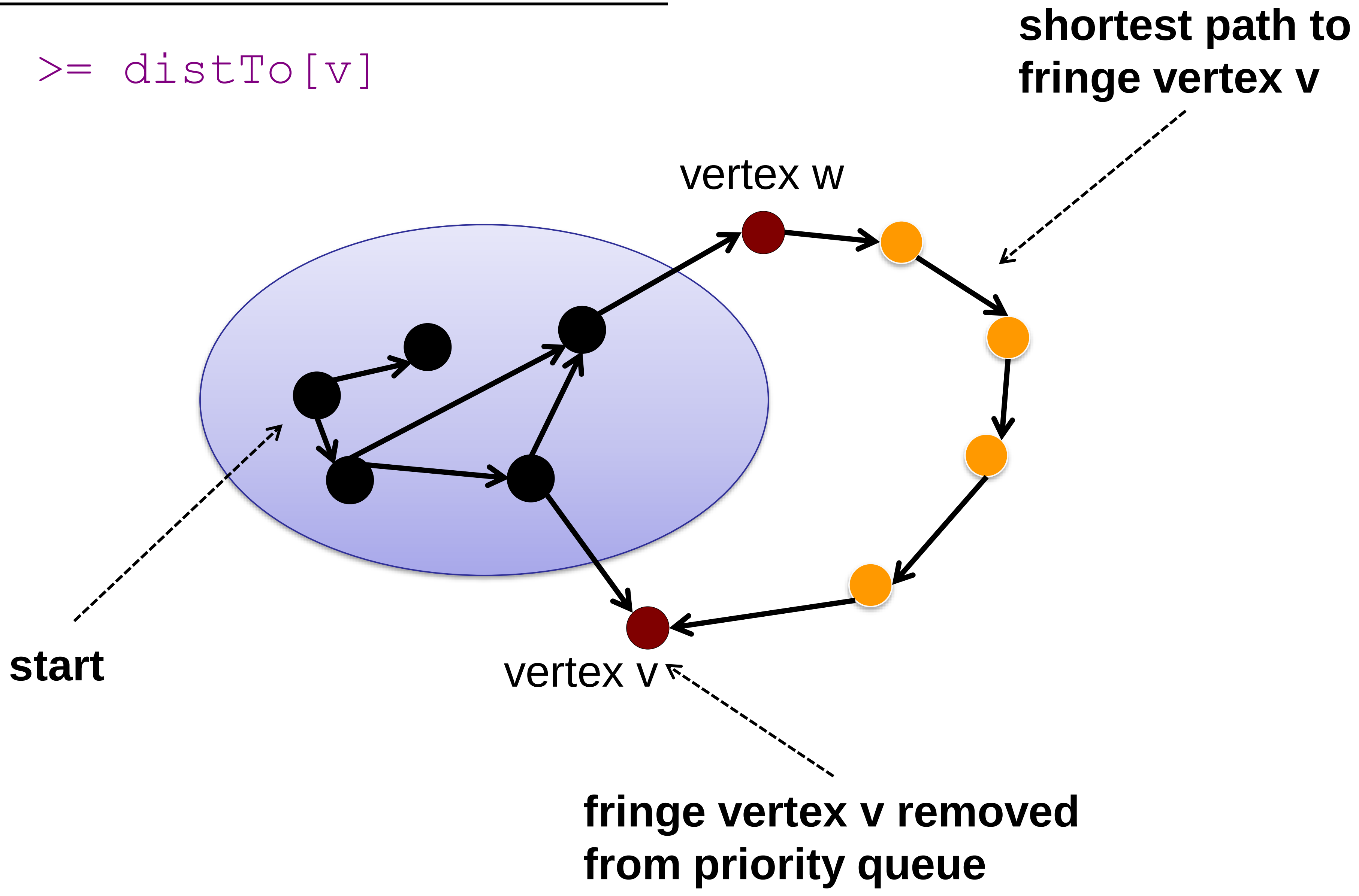
If P is shortest path to v, then prefix of P is shortest path to w.

Then distTo[w] is accurate.



Dijkstra's Algorithm

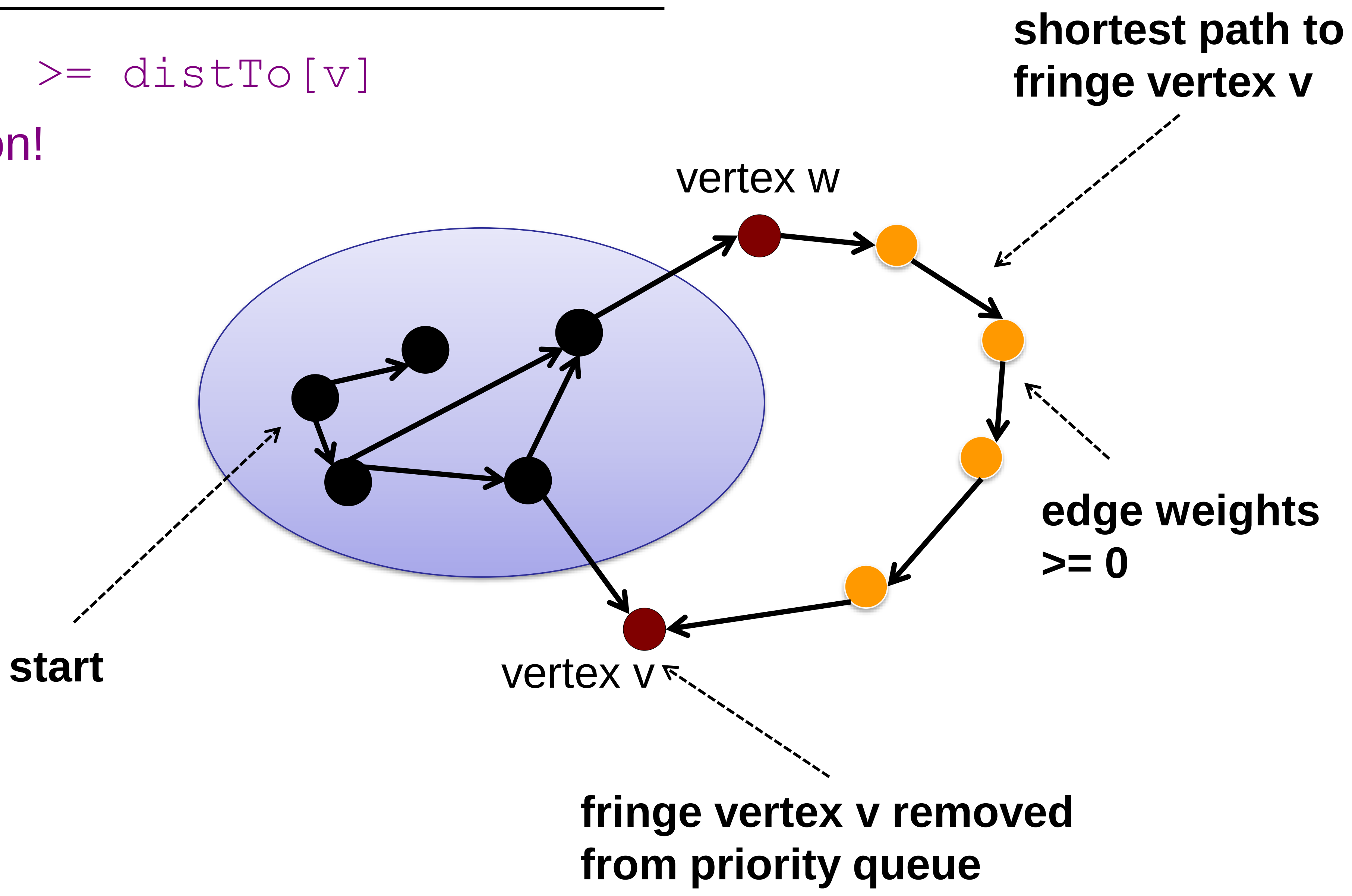
`distTo[w] >= distTo[v]`



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
  - the estimate is *never* changed again
  - 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

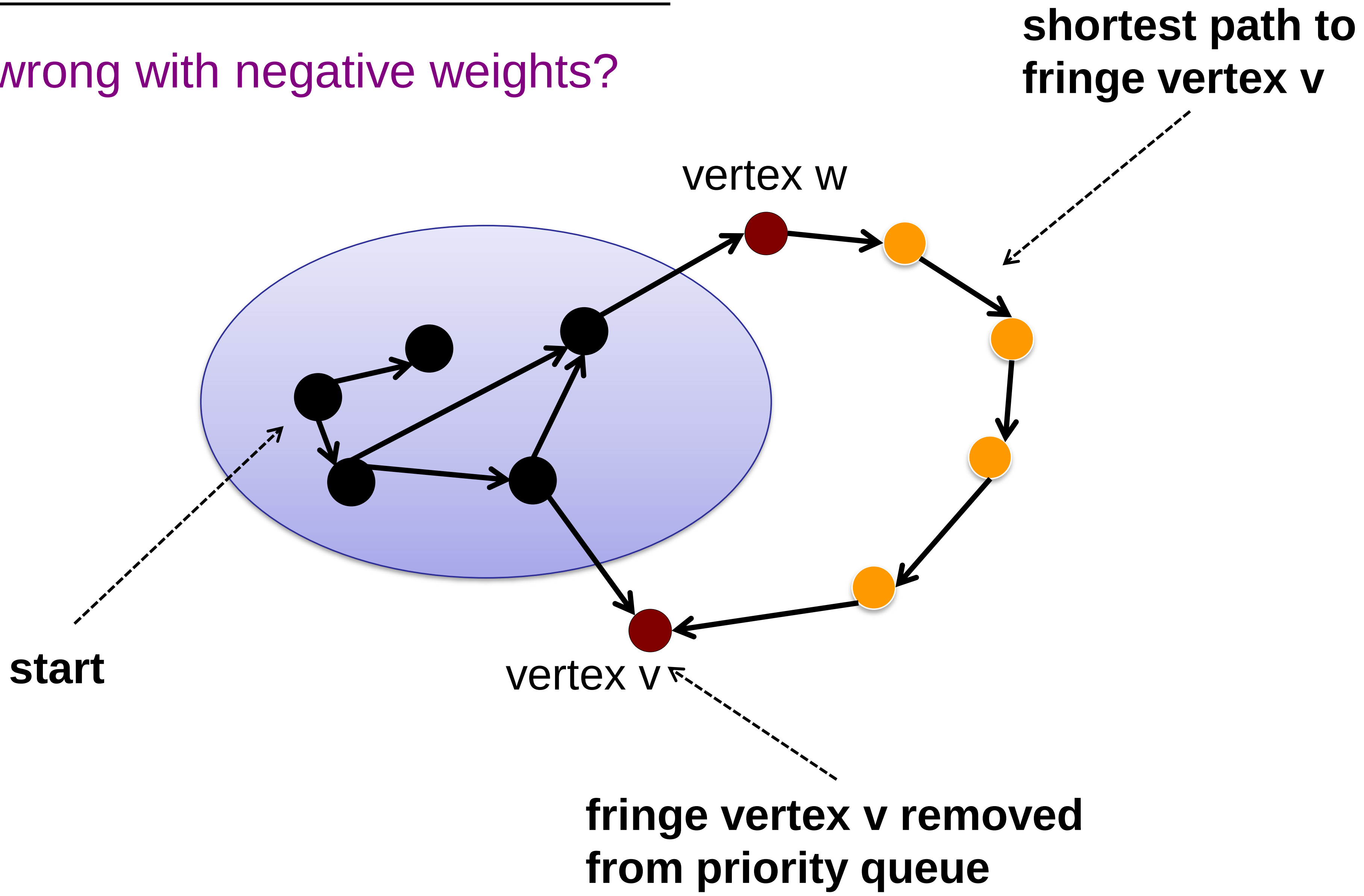
PQ Implementation	insert	deleteMin	decreaseKey	Total
Array	1	V	1	**$O(V^2)$**
AVL Tree	log V	log V	log V	**$O(E \log V)$**
d-way Heap	$d\log_d V$	$d\log_d V$	$\log_d V$	**$O(E\log_{E/V} V)$**
Fibonacci Heap	1	log V	1	**$O(E + V \log V)$**

Dijkstra Summary

Edges with negative weights?

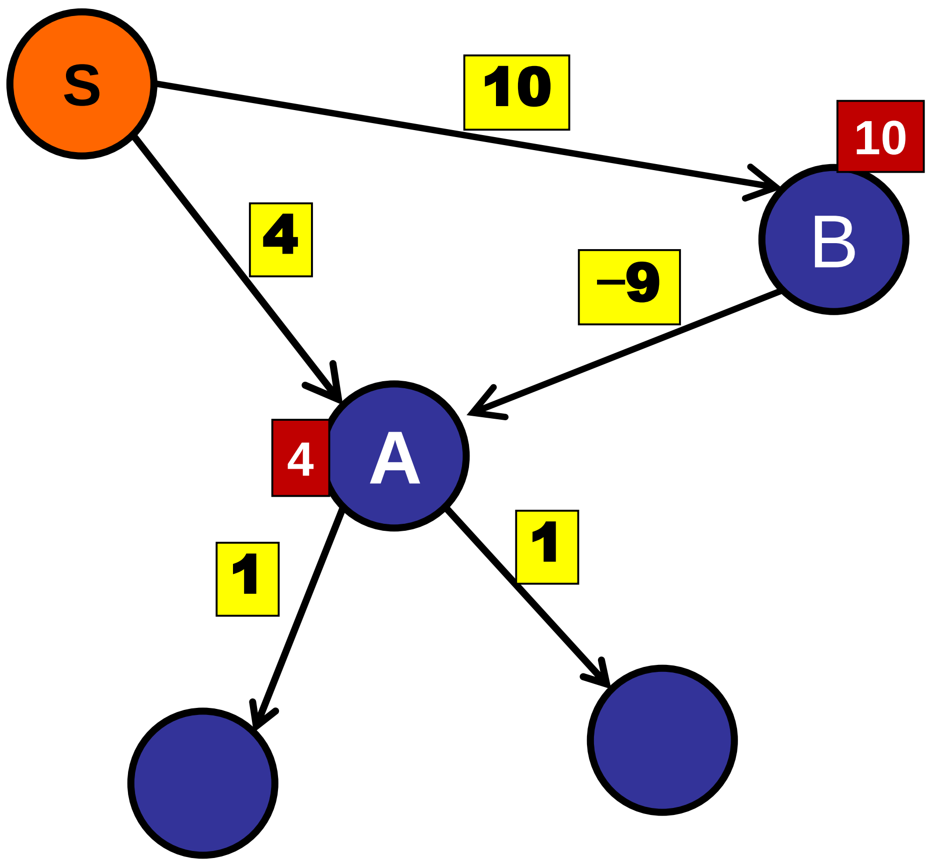
Dijkstra’s Algorithm

What goes wrong with negative weights?



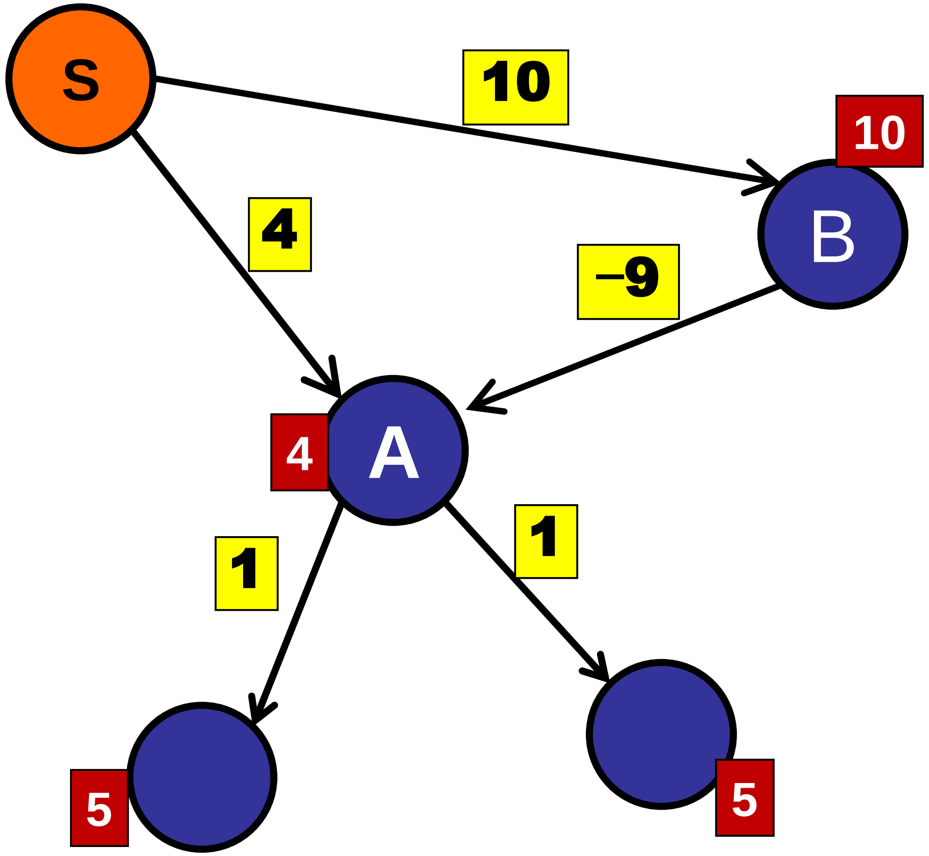
Dijkstra’s Algorithm

Edges with negative weights?



Dijkstra's Algorithm

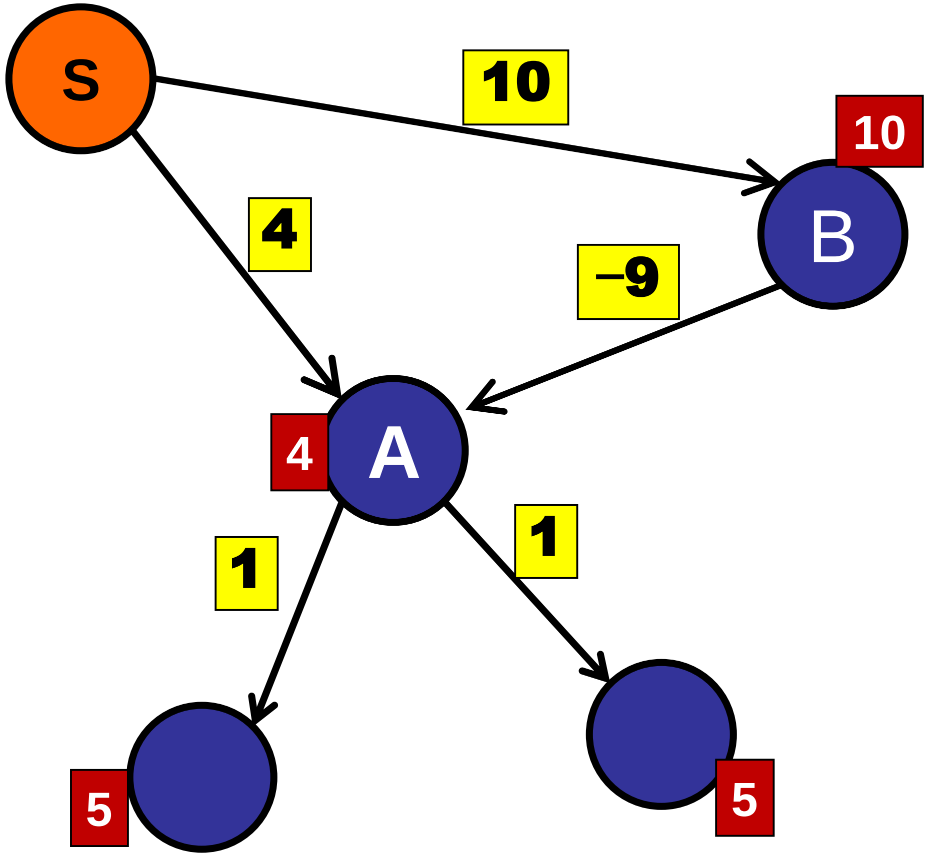
Edges with negative weights?



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.

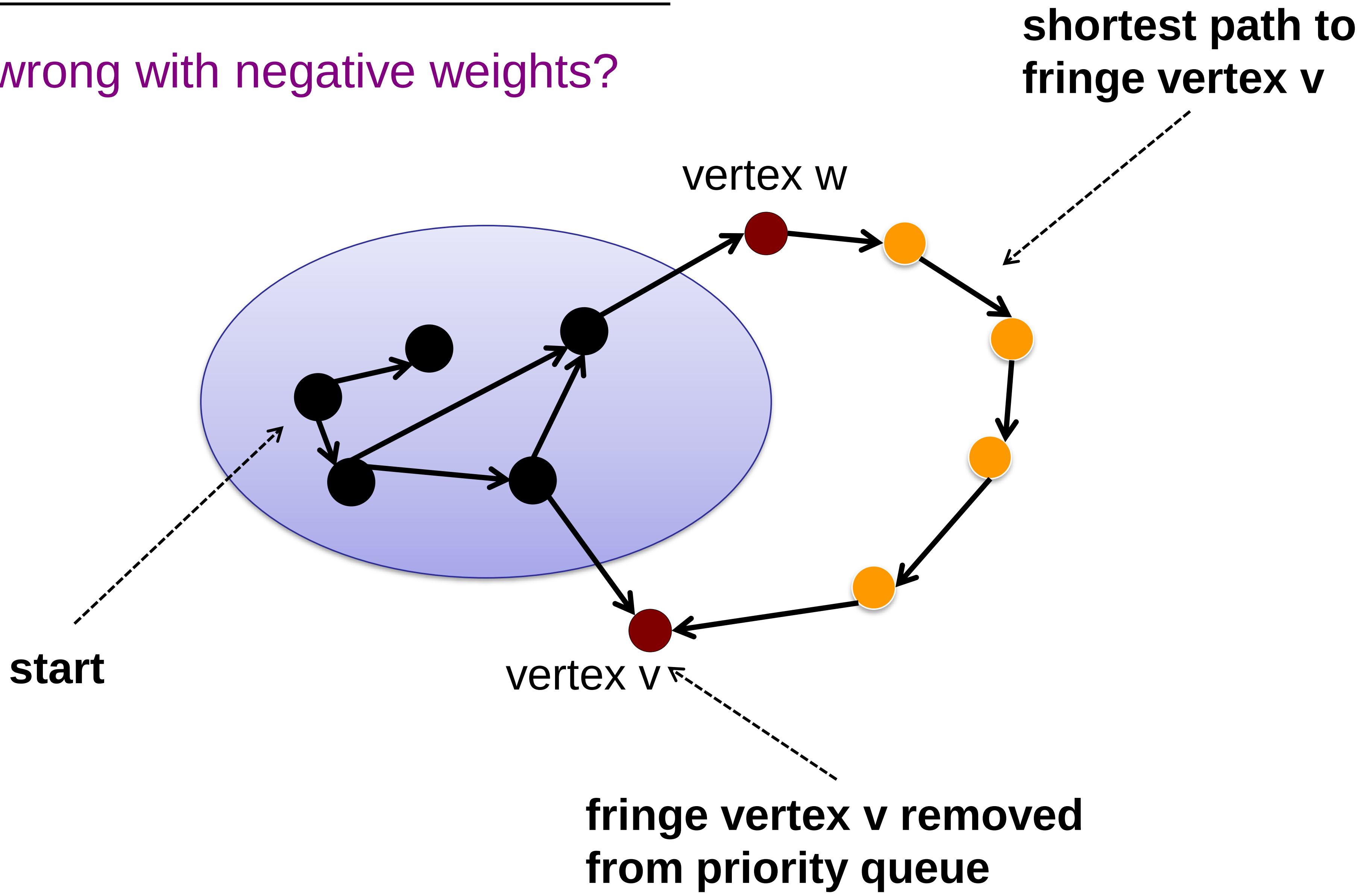
…

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

Oops: We need to update A.

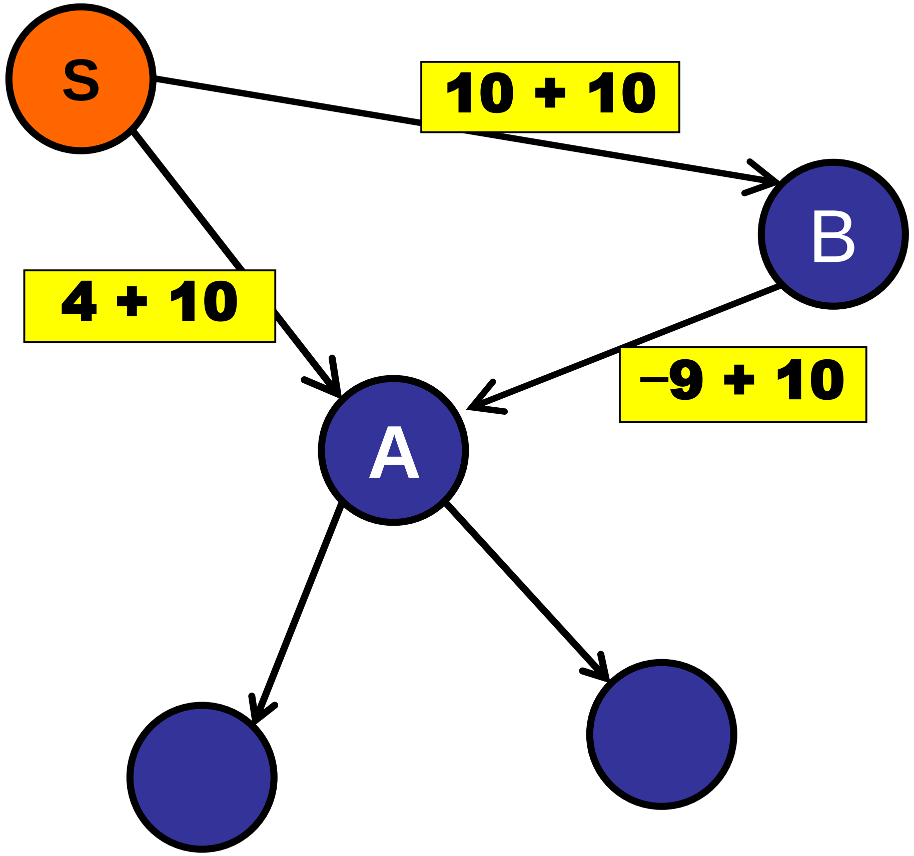
Dijkstra's Algorithm

What goes wrong with negative weights?



Dijkstra's Algorithm

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

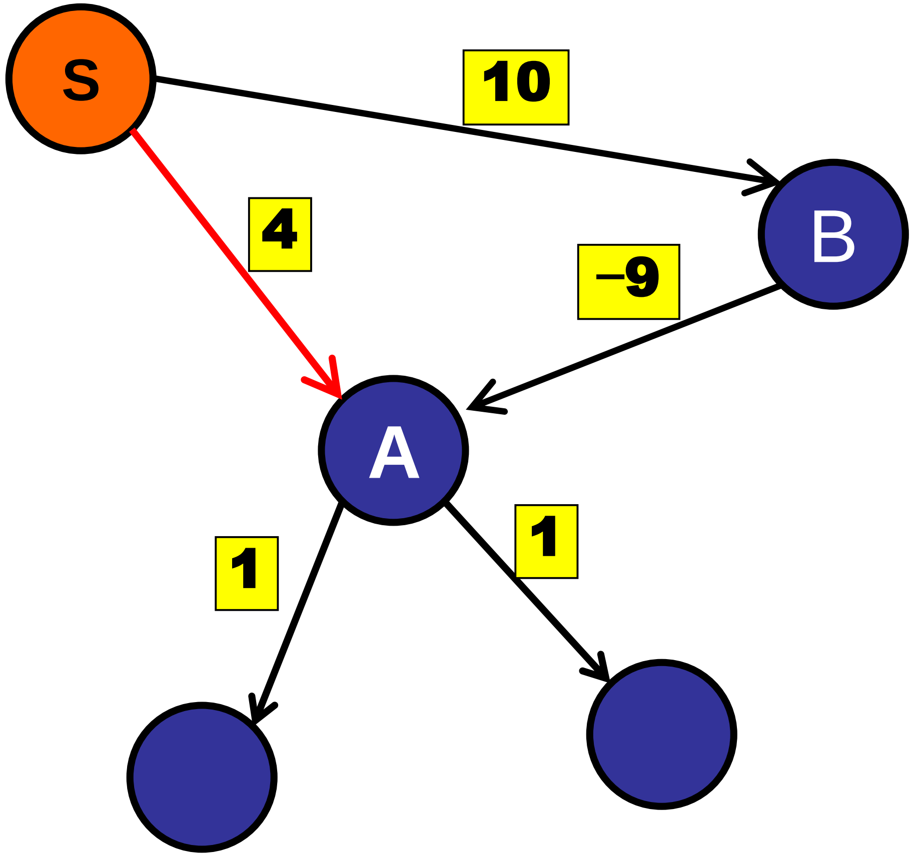


Can we reweight the graph?

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

Dijkstra’s Algorithm

Can we reweight?

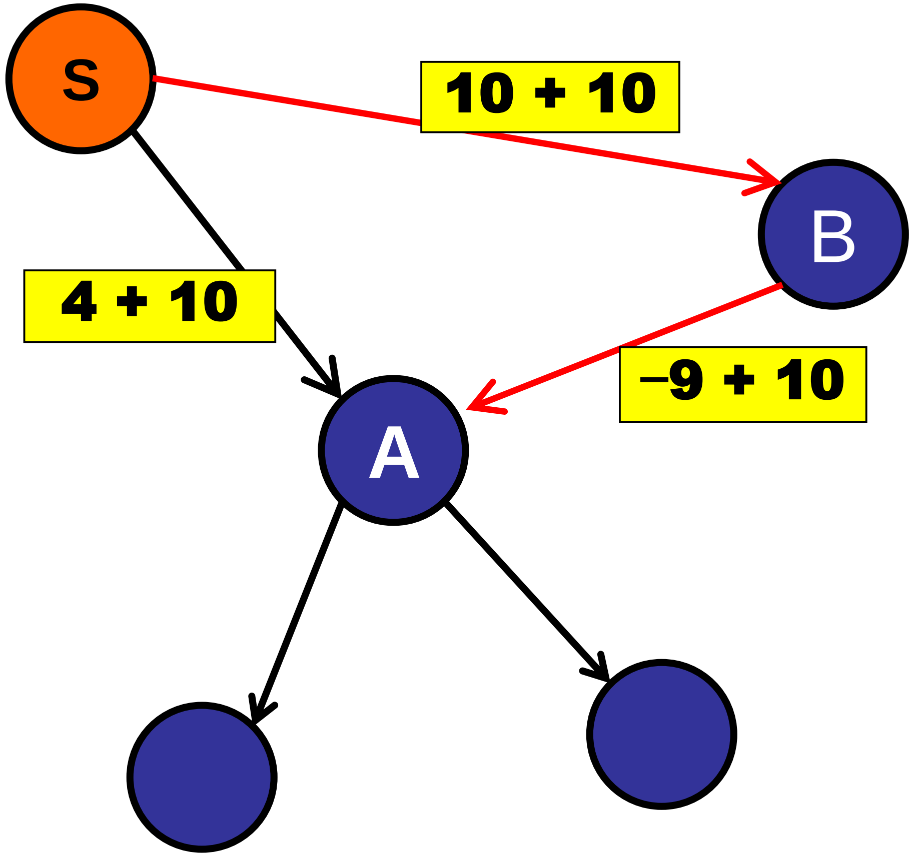


Path S-B-A: 1

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.

What about for Negative Weights?

- Only in October ‘22, an algorithm solving SSSP with negative weight edges with $\tilde{O}(E)$ running time proposed!

Negative-Weight Single-Source Shortest Paths in Near-linear Time

Aaron Bernstein* Danupon Nanongkai† Christian Wulff-Nilsen‡

Abstract

We present a randomized algorithm that computes single-source shortest paths (SSSP) in $O(m \log^8(n) \log W)$ time when edge weights are integral and can be negative.[1] This essentially resolves the classic negative-weight SSSP problem. The previous bounds are $\tilde{O}((m+n^{1.5}) \log W)$ [BLNPSSSW FOCS'20] and $m^{4/3+o(1)} \log W$ [AMV FOCS'20]. Near-linear time algorithms were known previously only for the special case of planar directed graphs [Fakcharoenphol and Rao FOCS'01].

In contrast to all recent developments that rely on sophisticated continuous optimization methods and dynamic algorithms, our algorithm is simple: it requires only a simple graph decomposition and elementary combinatorial tools. In fact, ours is the first combinatorial algorithm for negative-weight SSSP to break through the classic $\tilde{O}(m\sqrt{n} \log W)$ bound from over three decades ago [Gabow and Tarjan SICOMP'89].