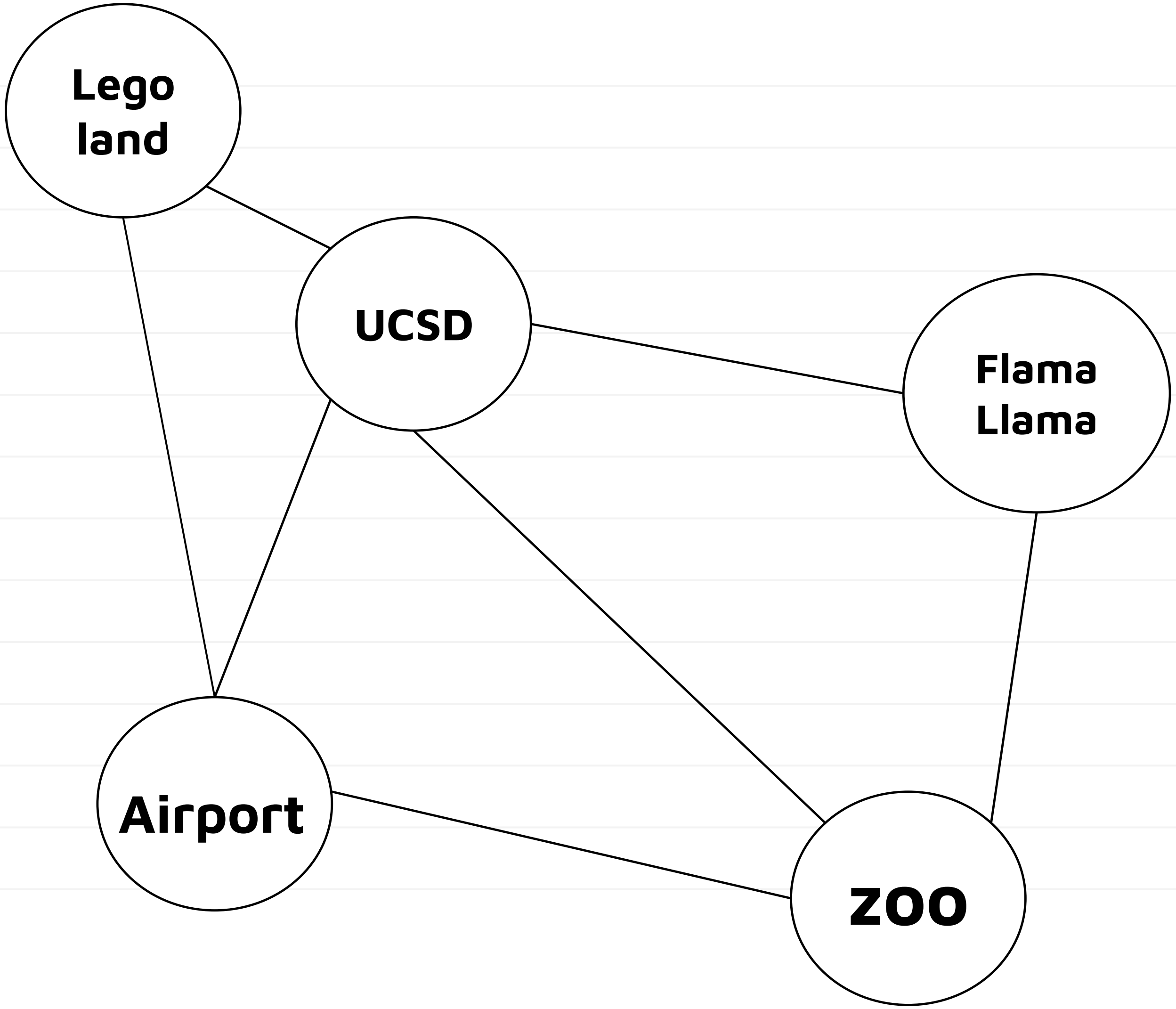
DSC 40B

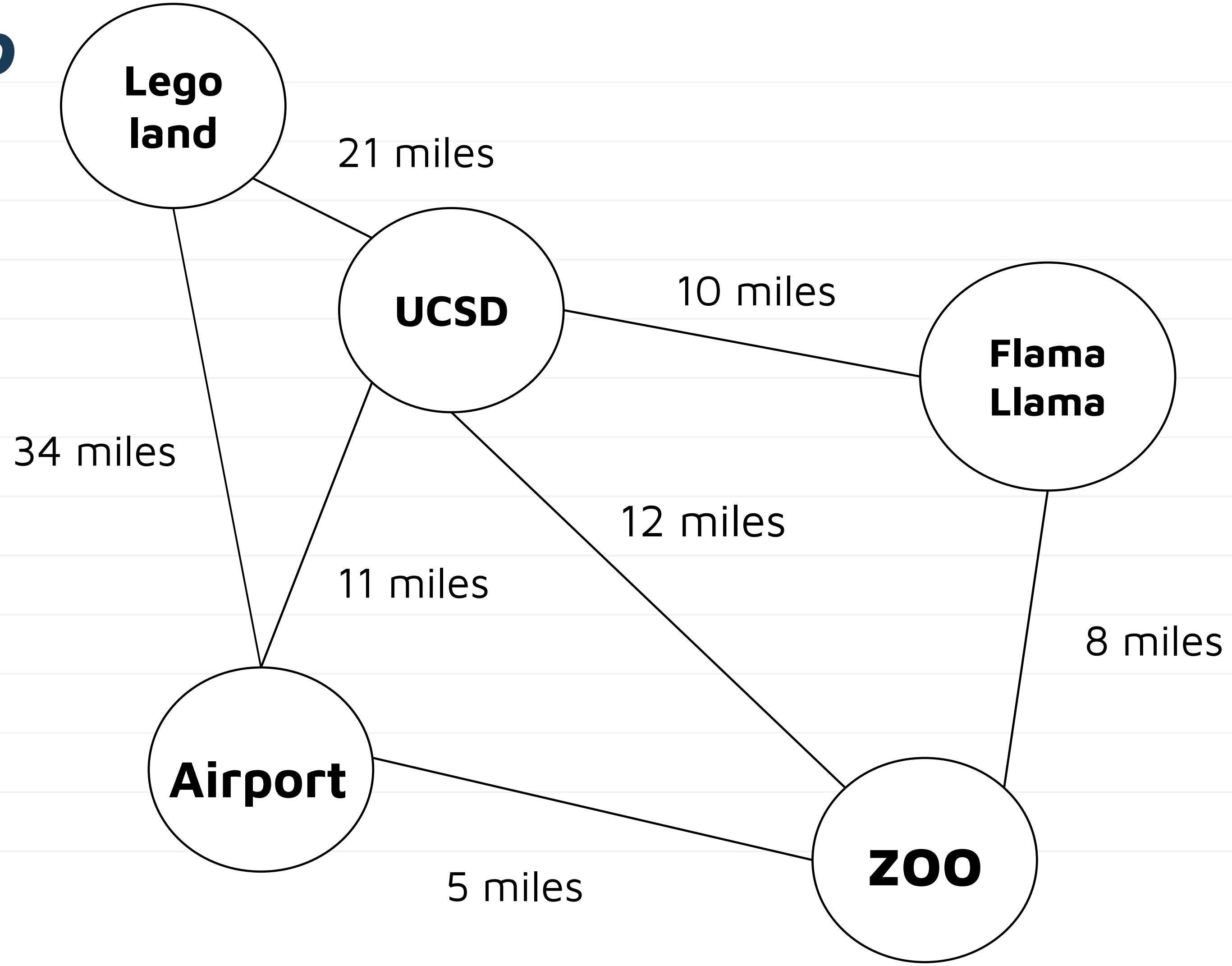
Lecture 23. : Shortest Paths in Weighted Graphs. Dijkstra

Shortest Paths in Weighted Graphs

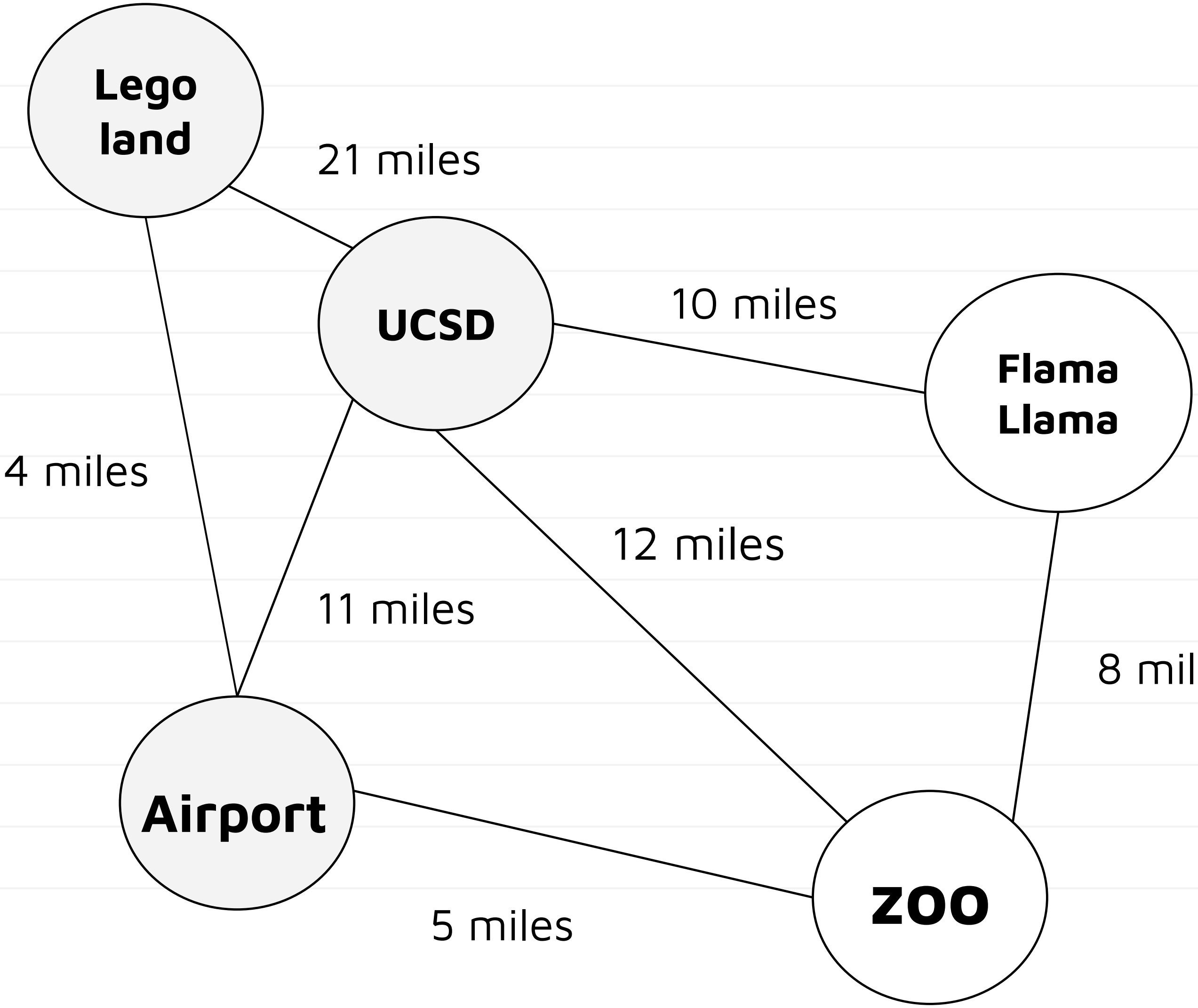
Map



Map


Lego land
21 miles
UCSD
10 miles
Flama Llama
34 miles
12 miles
11 miles
8 miles
Airport
5 miles
ZOO

Map


Lego land
21 miles
UCSD
10 miles
Flama Llama
34 miles
12 miles
11 miles
8 miles
Airport
ZOO
5 miles

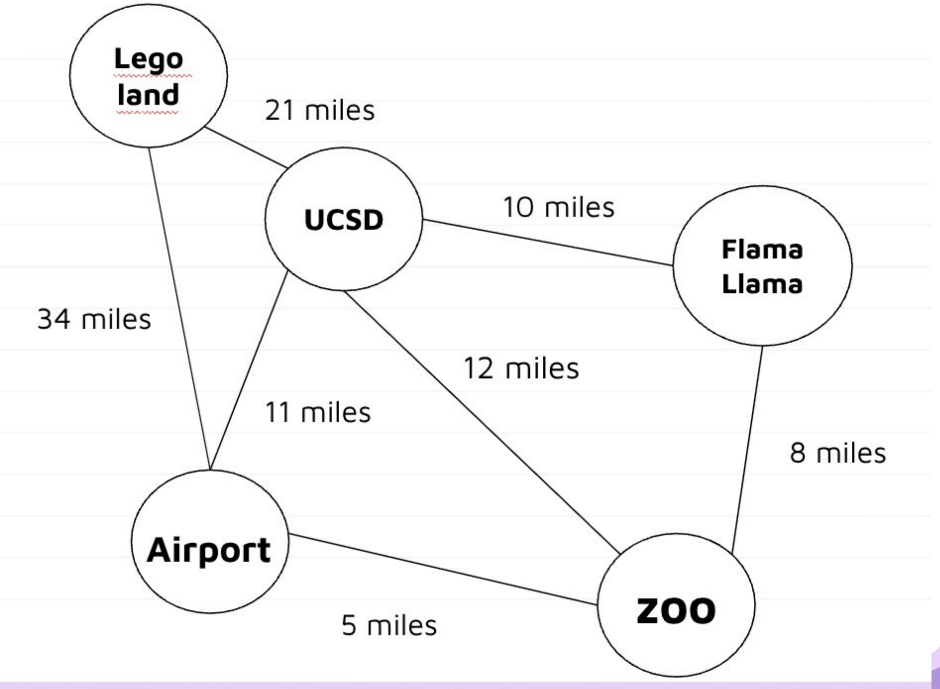
Weighted Graphs

- An **edge weighted** graph $G = (V, E, \omega)$ is a triple where (V, E) is a graph and $\omega : E \rightarrow \mathbb{R}$ maps each edge to a **weight**.
- Can be directed or undirected.
- In general, weights can be positive, negative, zero.
- Many uses, such as representing **metric spaces**.

Path Lengths

- The **length** of a path in a **weighted** graph (usually) refers to the **total weight of all edges** in the path.

Example:
(LegoLand, Airport, Zoo)
34 + 5 = 39

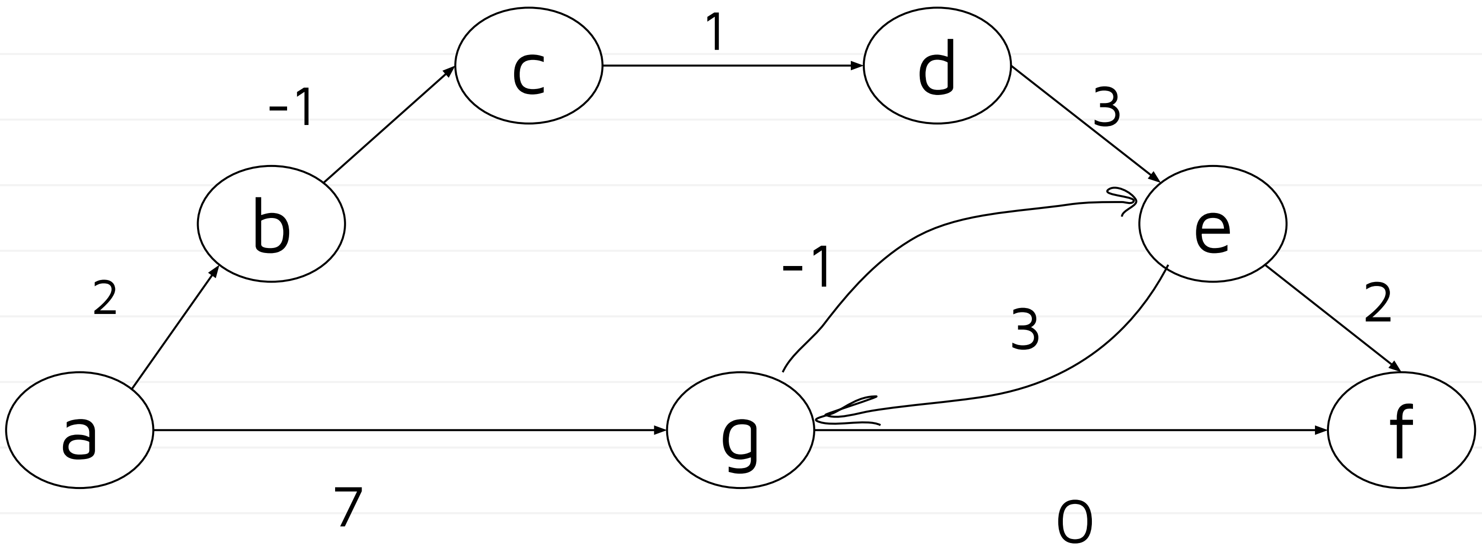


Shortest Paths

- A **shortest path** between u and v is a path between u and v with **minimum** length.
- In other words, **minimum total weight**.

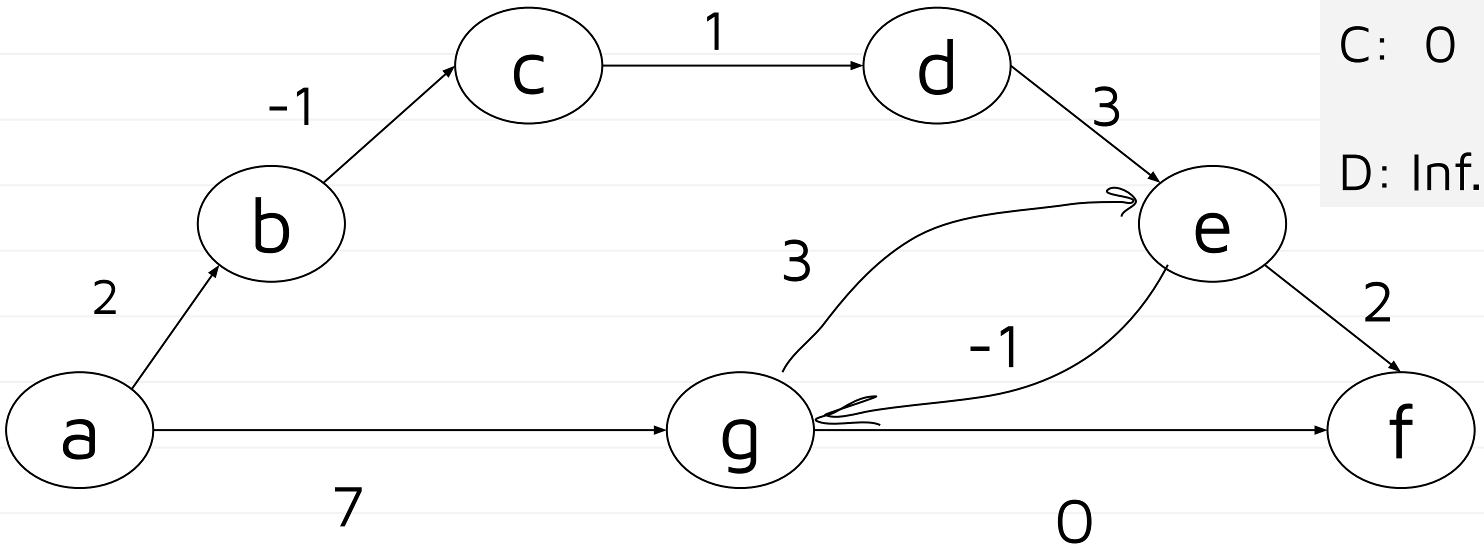
Example

What is the shortest path from **a** to **f**?



Example

What is the shortest path from **a** to **f**?



A: Some positive number

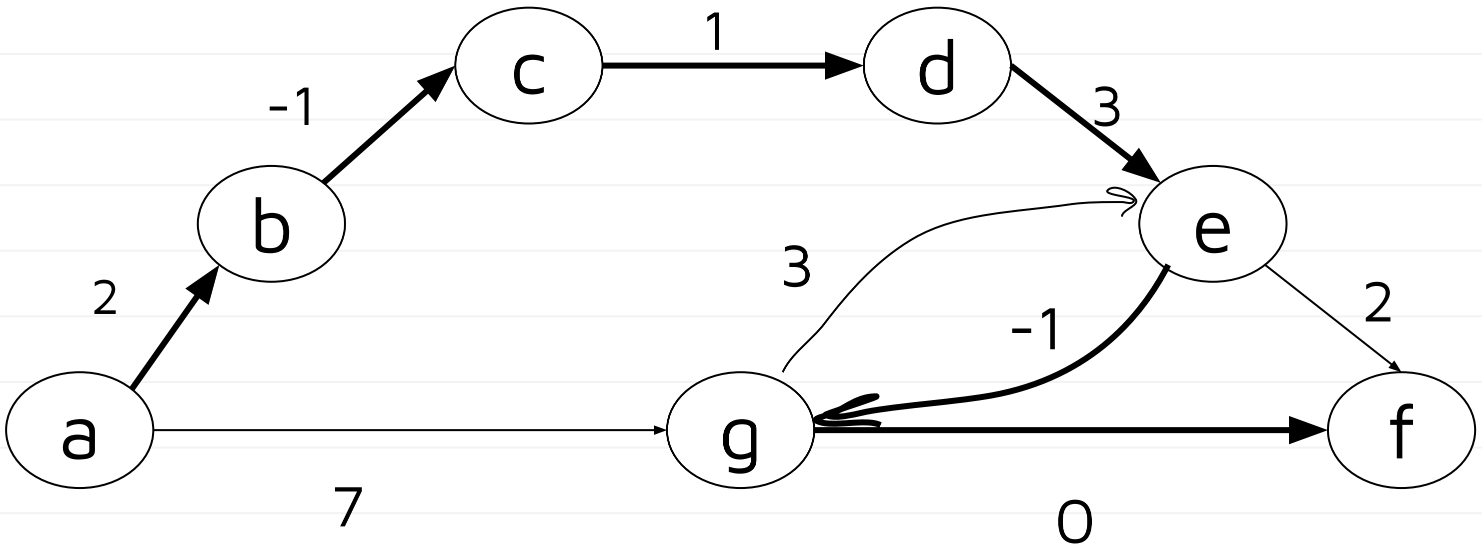
B: Some negative number

C: 0

D: Inf. loop

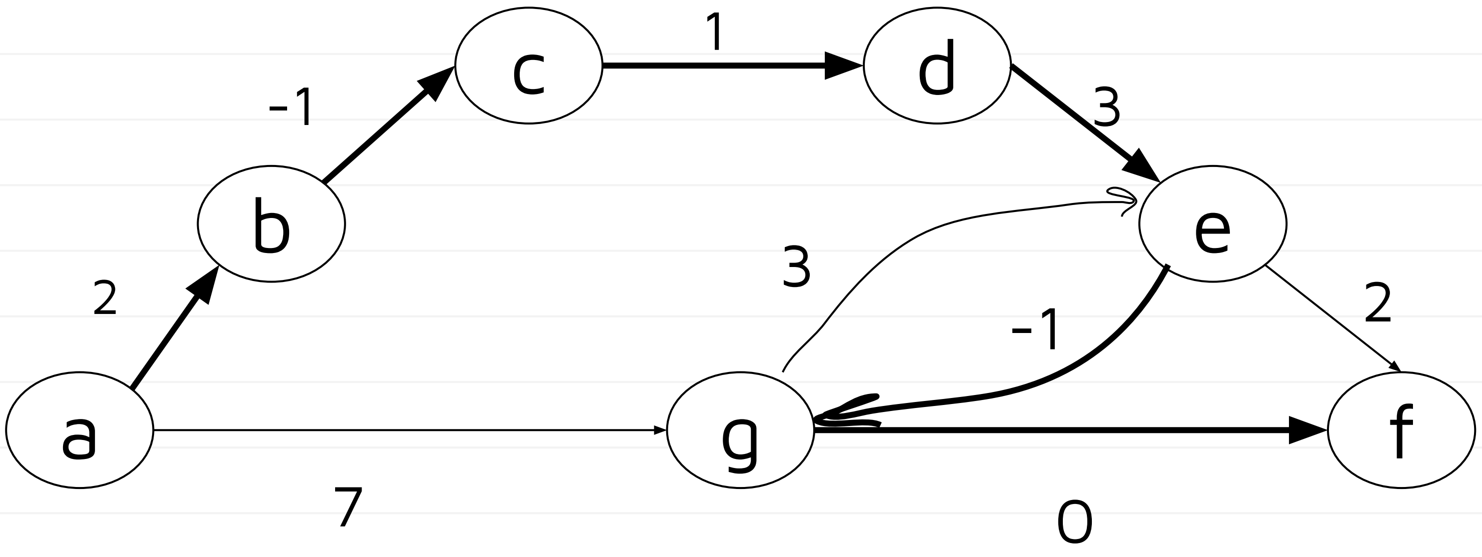
Example

What is the shortest path from **a** to **f**?



Example

What is the shortest path from **a** to **f**?



Path: a, b, c, d, e, g, f

Length: 2 - 1 + 1 + 3 - 1 + 0 = 4

Today (and next time)

- How do we find shortest paths in weighted graphs?

Idea #0

- Does BFS work?
  - **No**, not really. *Only if all weights are the same*.
- Can we "***convert***" a weighted graph to an unweighted one?

Idea #0. How can we convert?

Idea #0. When will it fail?

Idea #0

- **Step 1**: “Convert” weighted graph to unweighted one with dummy nodes.
- **Step 2**: Call BFS on this new graph.

Idea #0

- Very **inefficient** for large weights.
- What if edge weights are floats, or negative?

Ideas #1 and #2

- We'll look at two algorithms: Bellman-Ford and Dijkstra's.
  - **Input**: weighted graph, source vertex s.
  - **Output**: shortest paths from s to *every other* node.
- Both work by:
  - keeping track of shortest known path (**estimates**).
  - iteratively **updating** these until they're correct.

Shortest Path Estimates

mic

- Bellman-Ford and Dijkstra's keep track of the shortest paths found so far; we call these the **estimated shortest paths**.
- For each node u, remember u's:
  - predecessor in estimated shortest path;
  - distance from source s in estimated shortest path.
- **Key**: estimated distance will always be ≥ actual distance.

Updates: operations on an edge

- Both algorithms work by iteratively **updating** their estimates.
- On each iteration, consider a **new** edge (u, v).

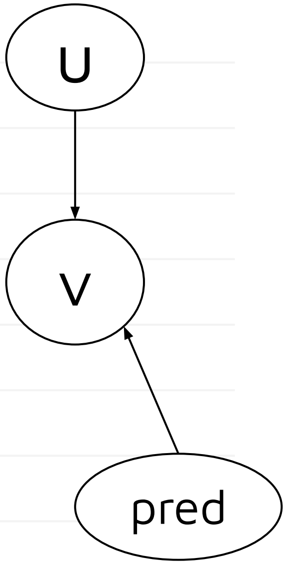
Ask: is the ***best*** known shortest path from

source → ⋯ → $\boldsymbol{u}$ → $\boldsymbol{v}$

shorter than the best known shortest path from

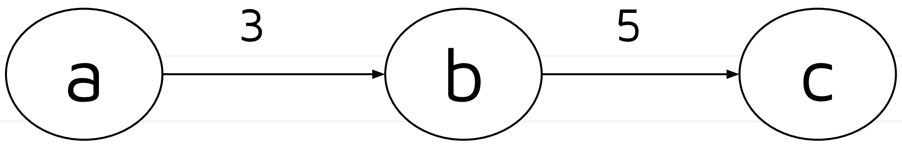
source → ⋯ → predecessor[v] → v?

- If it is, we have discovered a shorter path to v.



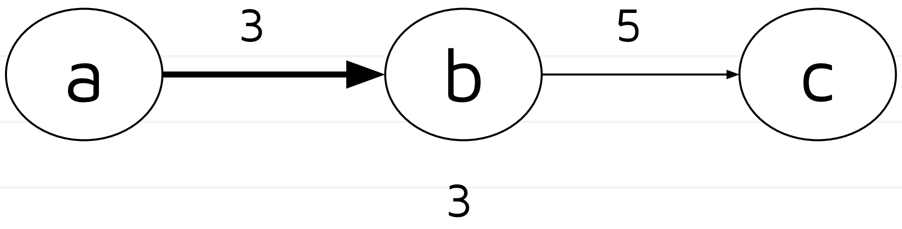
Dijkstra's Algorithm

- Intuition



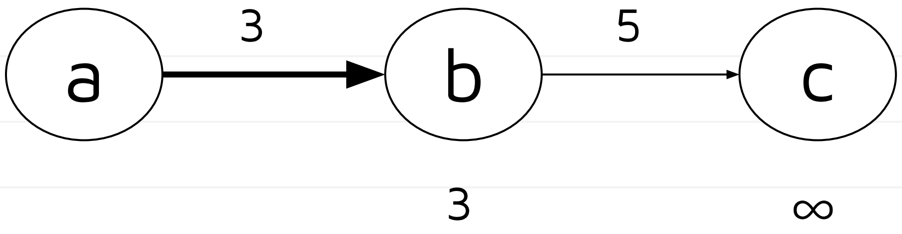
Dijkstra’s Algorithm

- Intuition



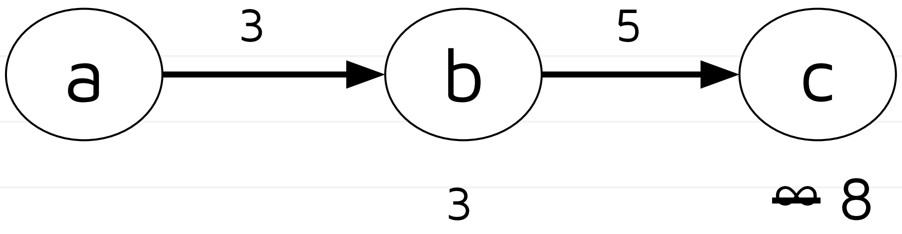
Dijkstra's Algorithm

- Intuition



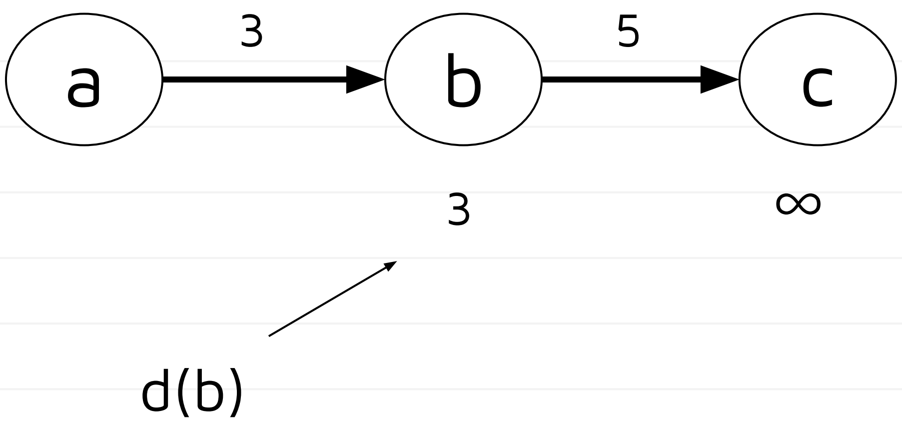
Dijkstra's Algorithm

- Intuition



Dijkstra's Algorithm

- Intuition

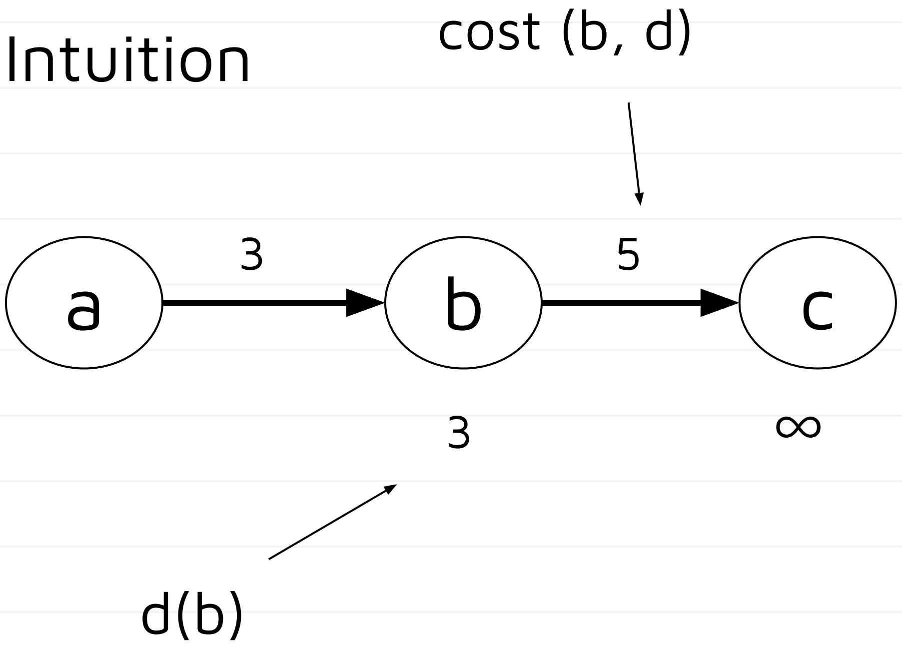


Update Rule:

if d(b)

Dijkstra's Algorithm

- Intuition

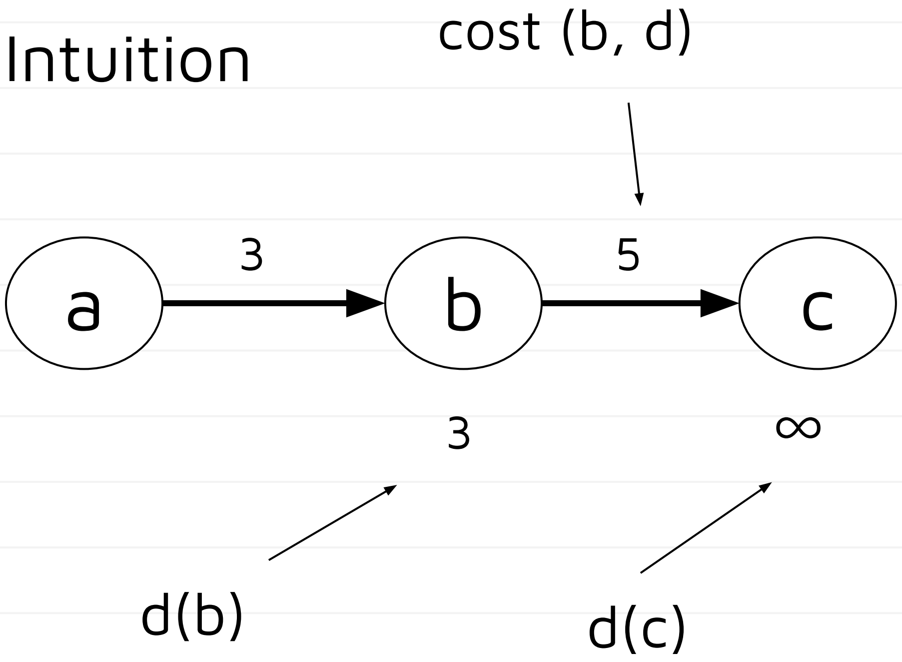


Update Rule:

if d(b) + cost(b, c)

Dijkstra's Algorithm

- Intuition

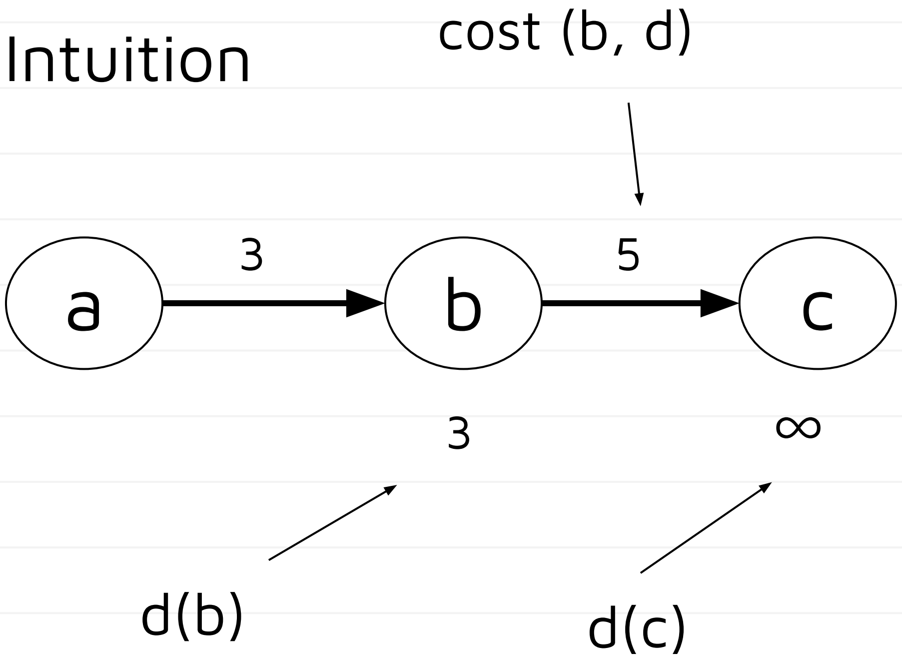


Update Rule:

if d(b) + cost(b, c)< d(c):

Dijkstra's Algorithm

- Intuition



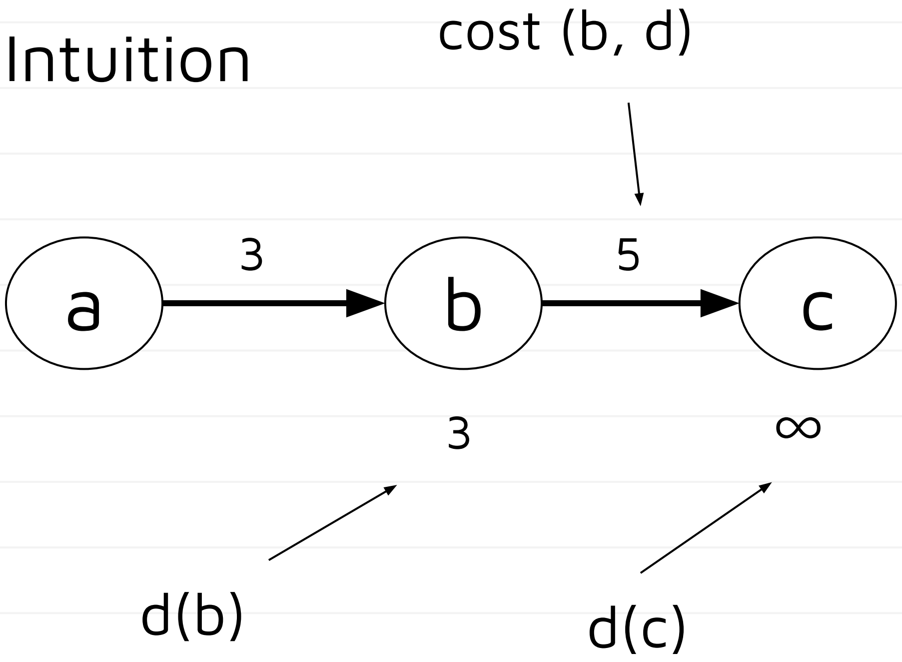
Update Rule:

if d(b) + cost(b, c)< d(c):

d(c)=d(b) + cost(b, d)

Dijkstra's Algorithm

- Intuition



Update Rule:

if d(b) + cost(b, c)< d(c):

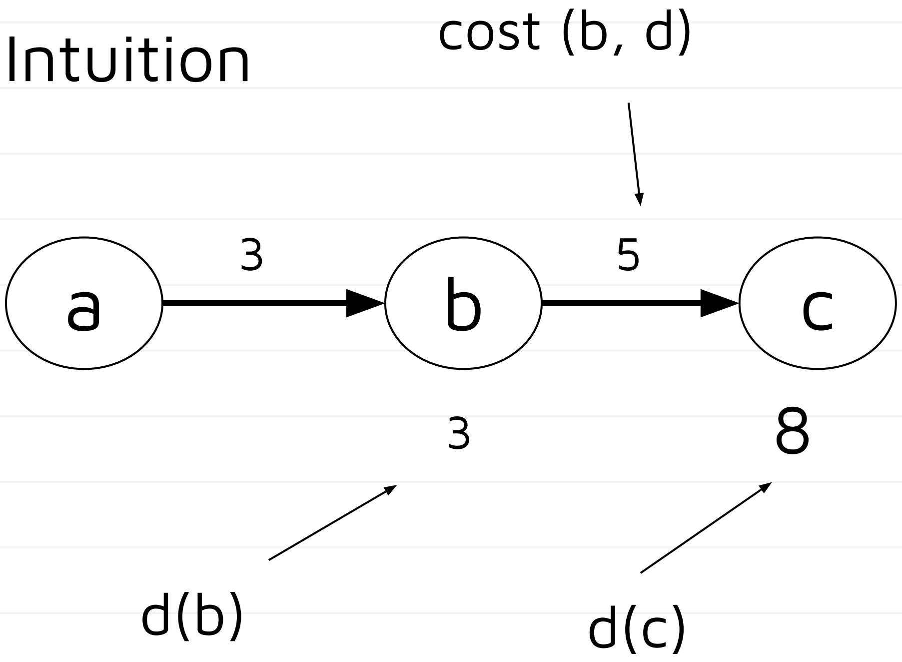
d(c)=d(b) + cost(b, d)

If 3 + 5 < ∞:

d(c) = 8

Dijkstra's Algorithm

- Intuition



Update Rule:

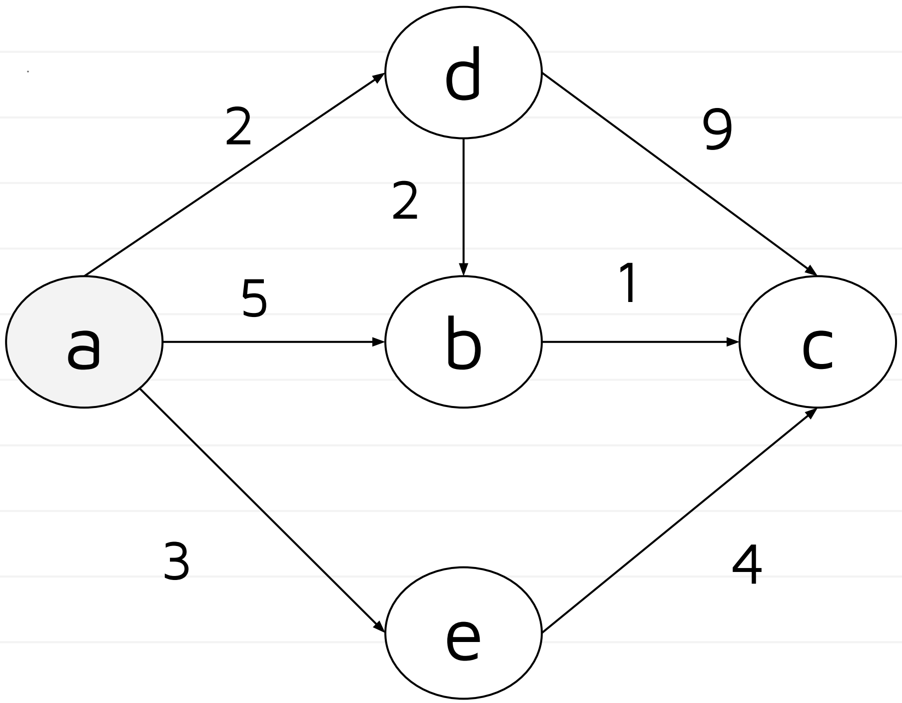
if d(b) + cost(b, c)< d(c):

    d(c)=d(b) + cost(b, d)

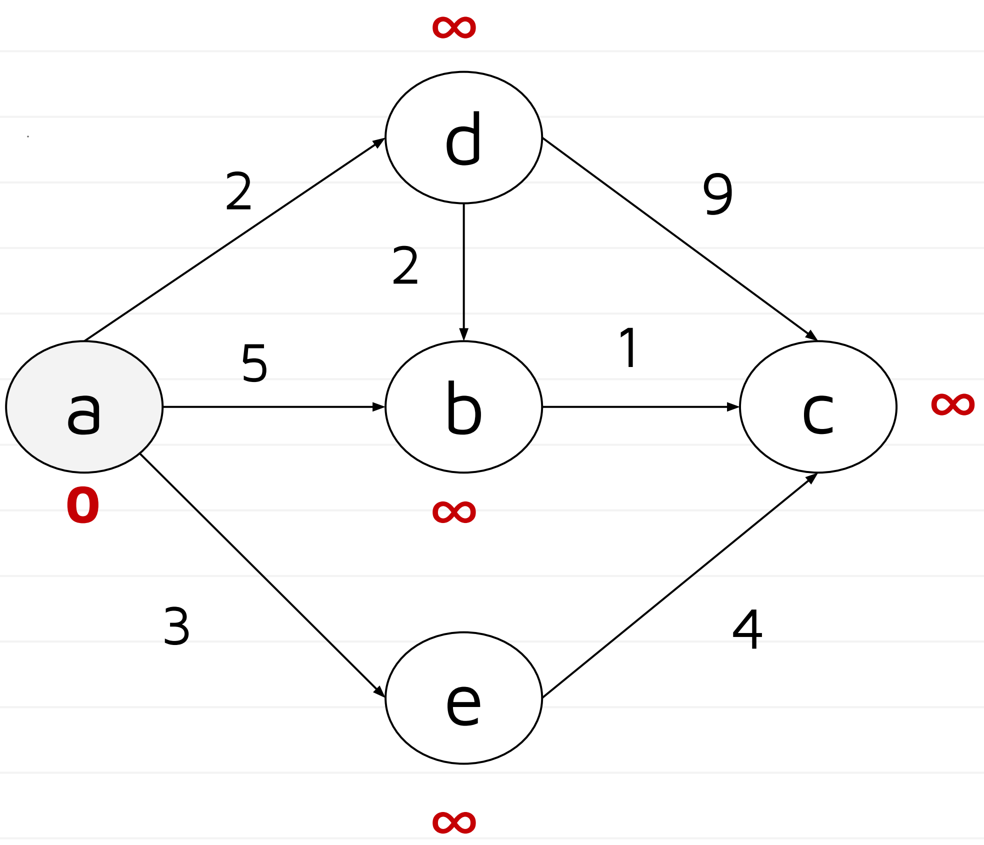
If 3 + 5 < ∞:

    d(c) = 8

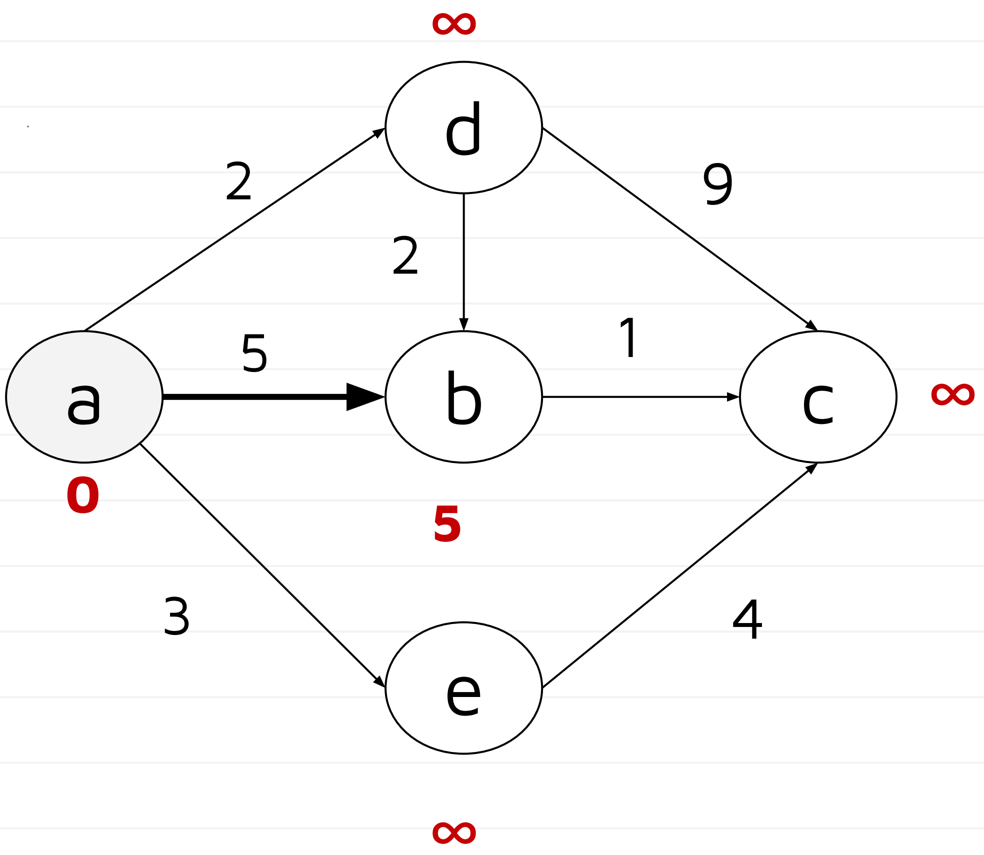
Dijkstra's Algorithm



Dijkstra’s Algorithm



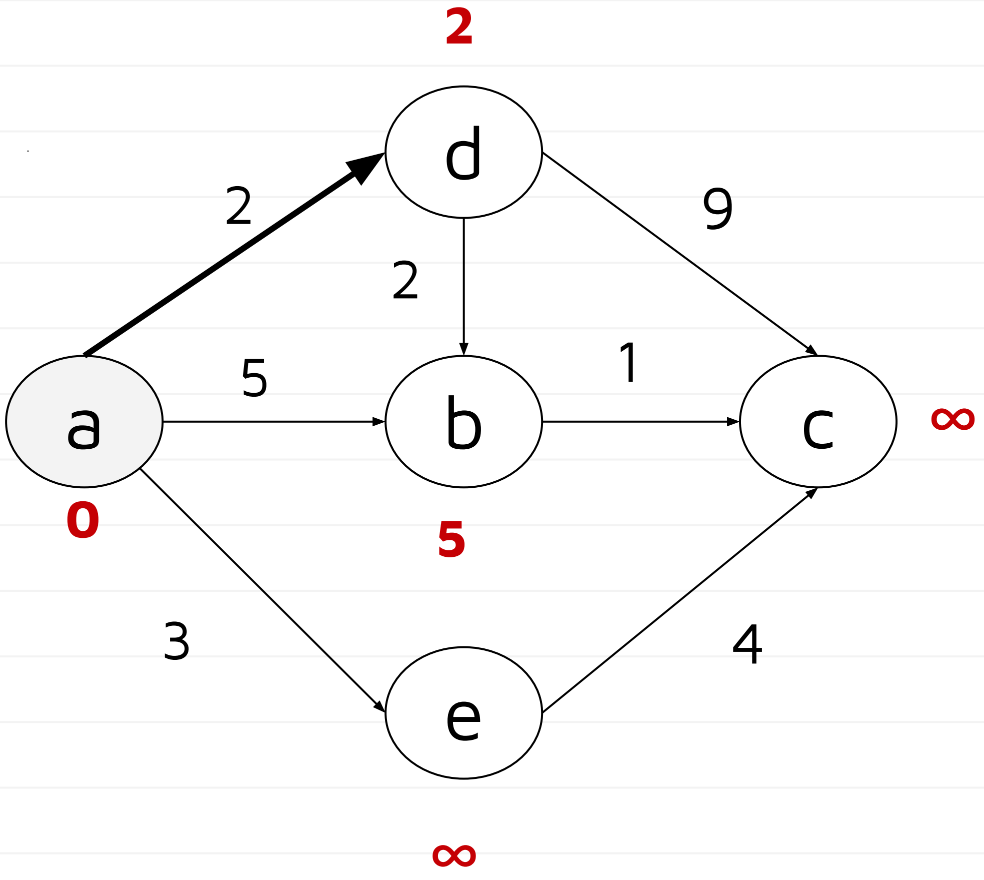
Dijkstra's Algorithm



Update rule:

If 0 + 5 < ∞ , update

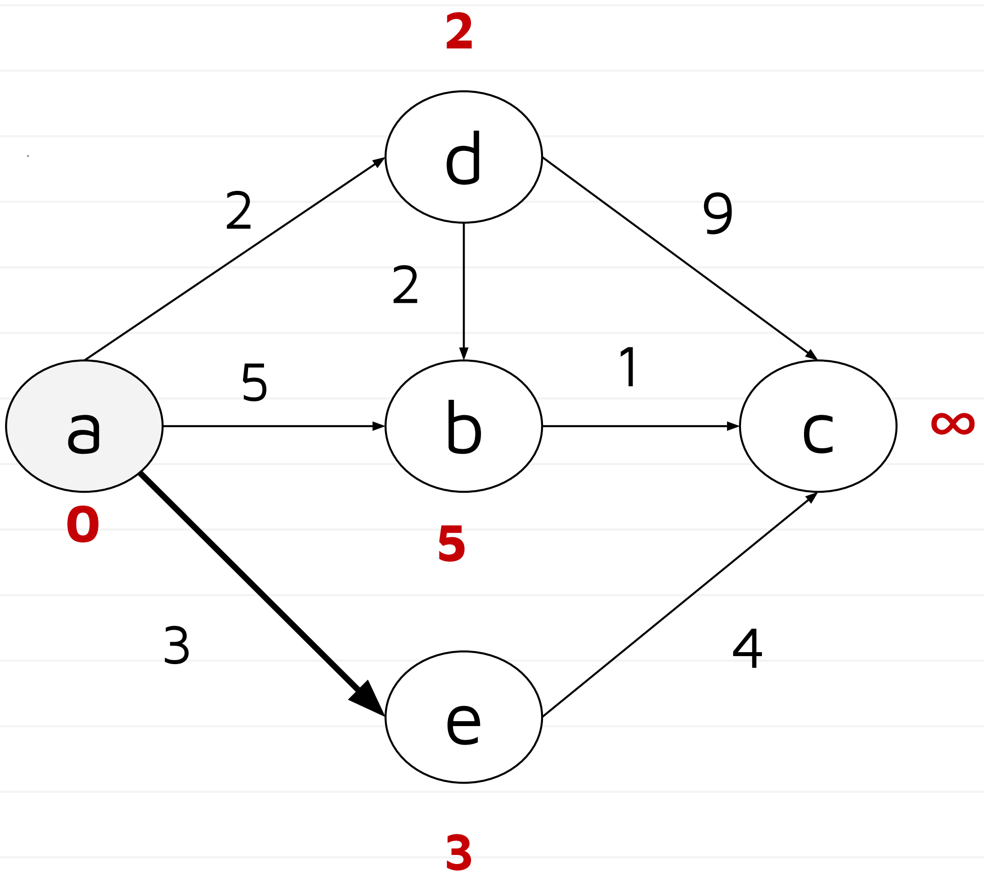
Dijkstra's Algorithm



Update rule:

If 0 + 2 < ∞ , update

Dijkstra's Algorithm

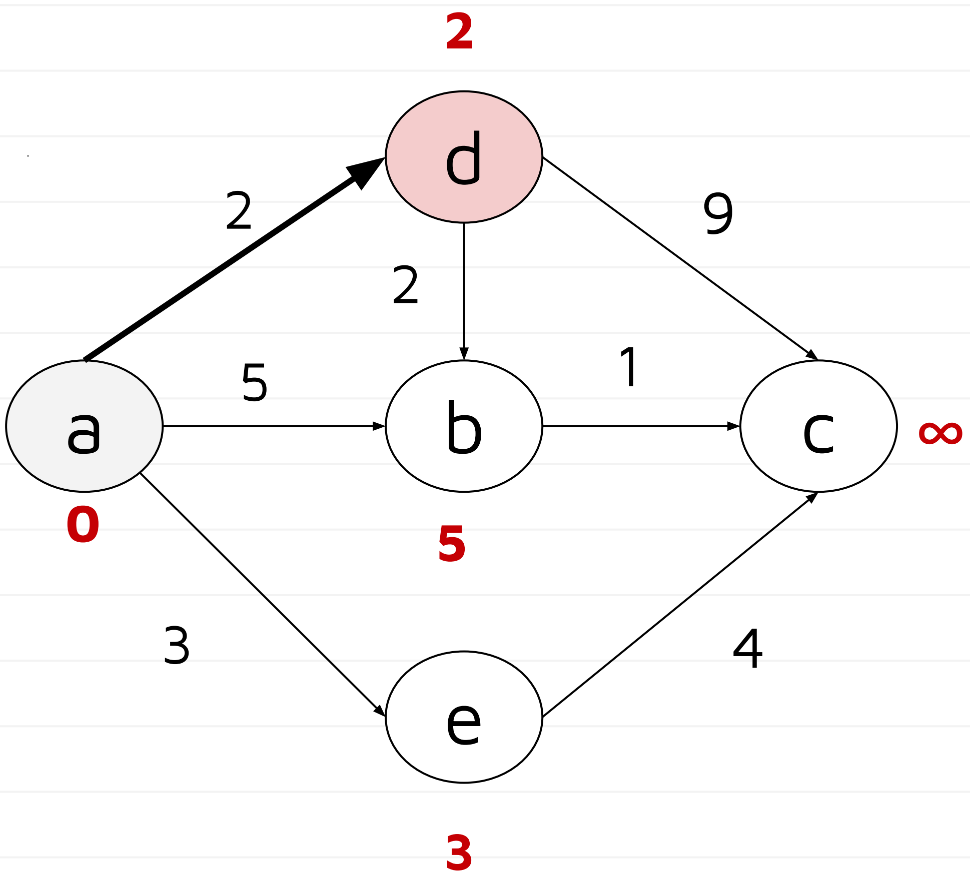


Update rule:

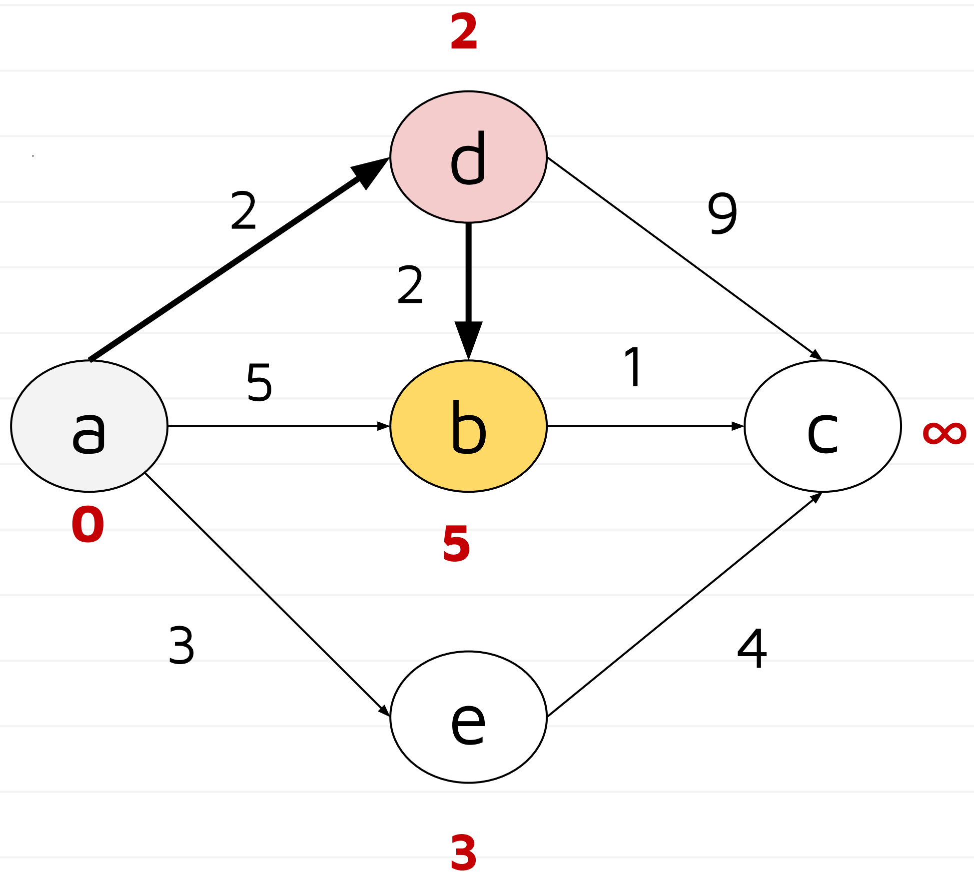
If 0 + 3 < ∞ , update

Dijkstra's Algorithm

Select the **shortest** path: 2



Dijkstra's Algorithm



Select the **shortest** path: 2

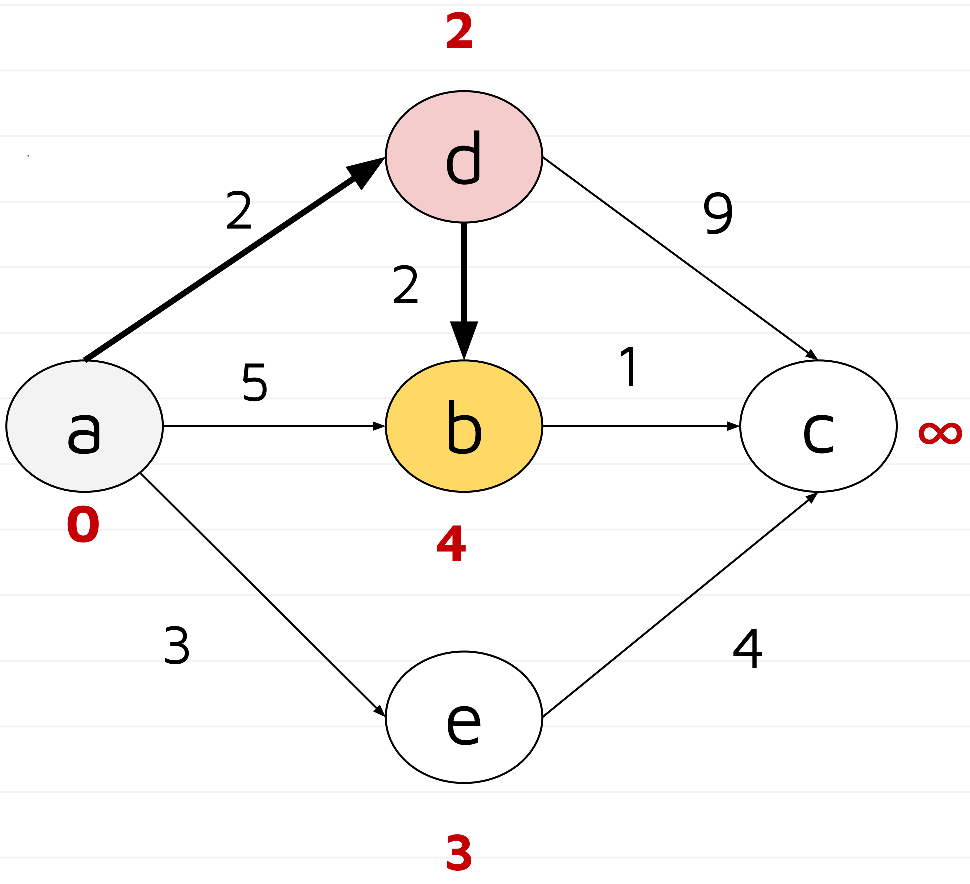
Update rule:

If 2 + 2 < 5, update

if d(u) + cost(u, v)< d(v):

d(v)=d(u) + cost(u, v)

Dijkstra's Algorithm



Select the **shortest** path: 2

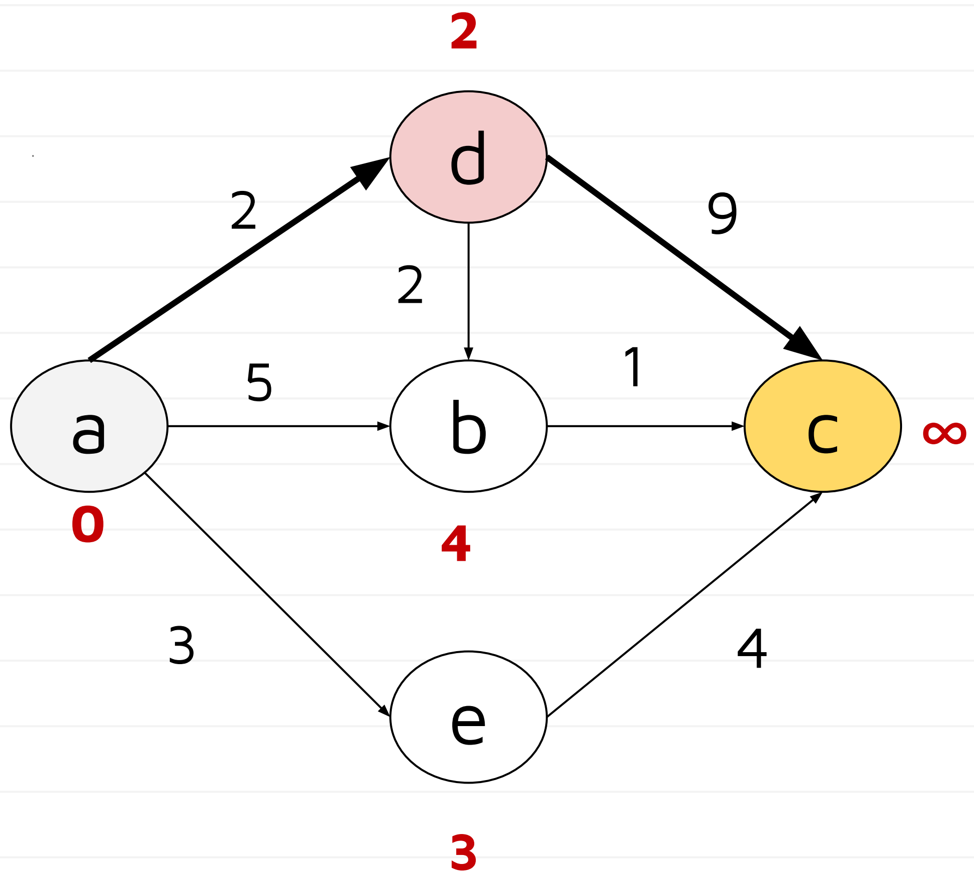
Update rule:

If 2 + 2 < 5, update

if d(u) + cost(u, v)< d(v):

d(v)=d(u) + cost(u, v)

Dijkstra's Algorithm



Select the **shortest** path: 2

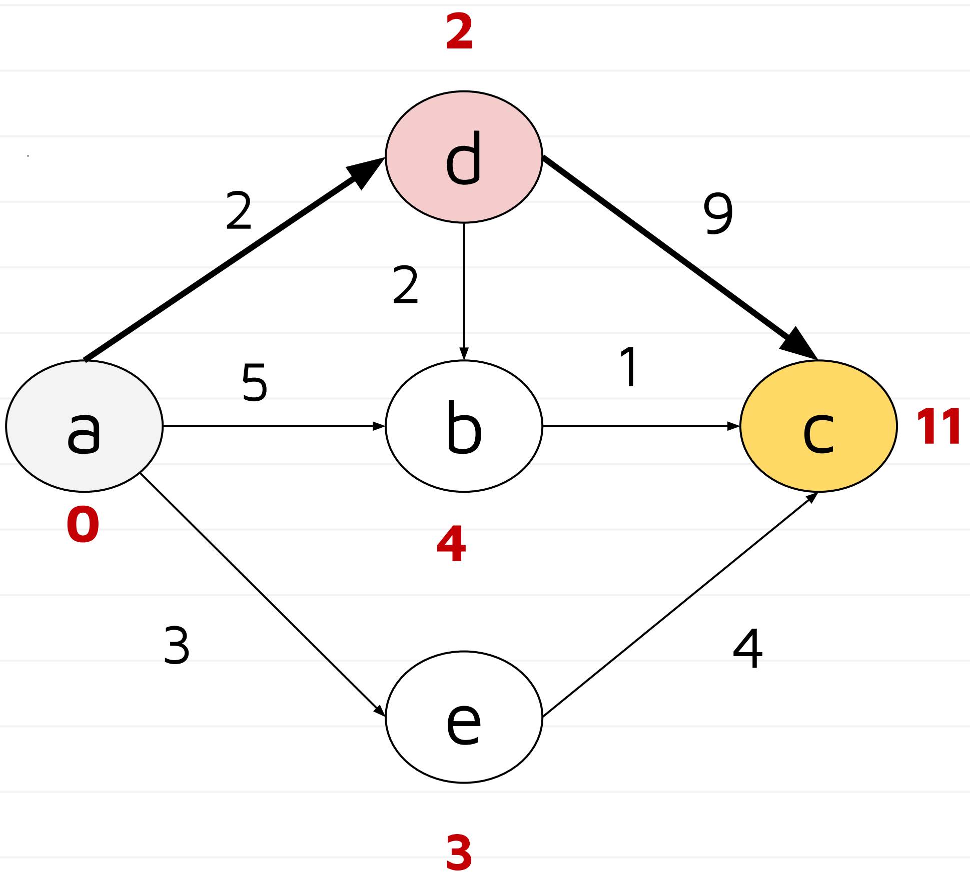
Update rule:

If 2 + 9 < inf, update

if d(u) + cost(u, v)< d(v):

d(v)=d(u) + cost(u, v)

Dijkstra's Algorithm



Select the **shortest** path: 2

Update rule:

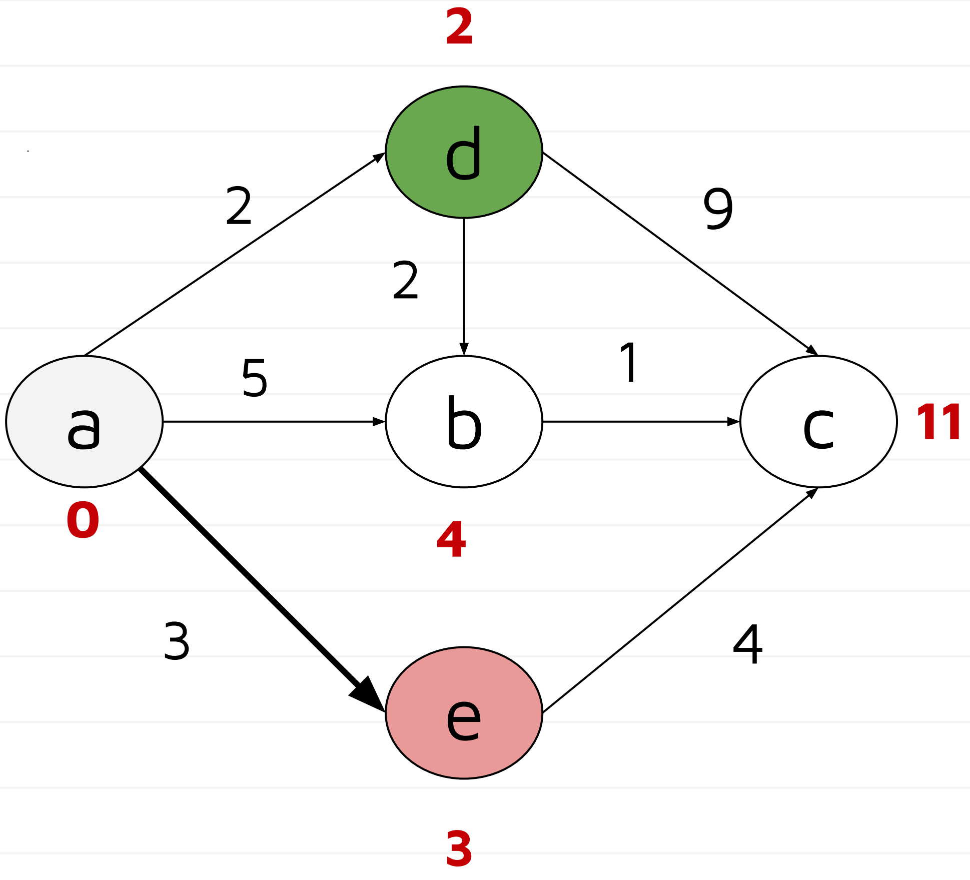
If 2 + 9 < inf, update

if d(u) + cost(u, v)< d(v):

d(v)=d(u) + cost(u, v)

Dijkstra's Algorithm

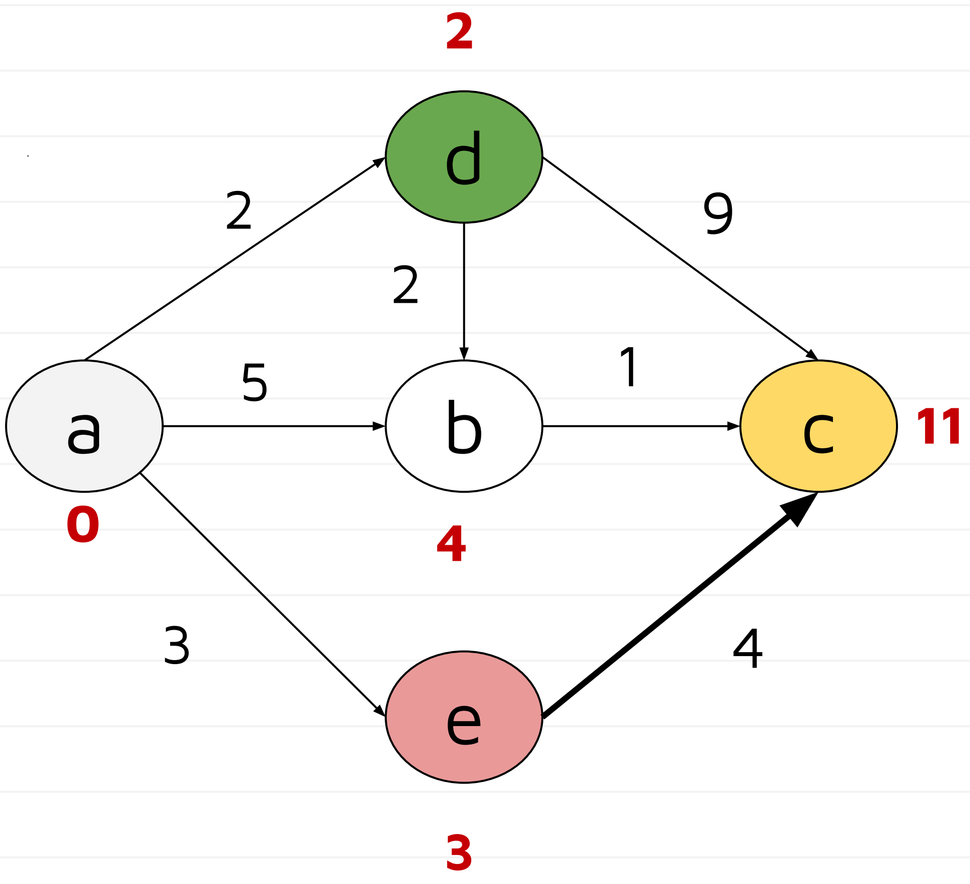
Select the **shortest** path: 3



if d(u) + cost(u, v)< d(v):

d(v)=d(u) + cost(u, v)

Dijkstra's Algorithm



Select the **shortest** path: 3

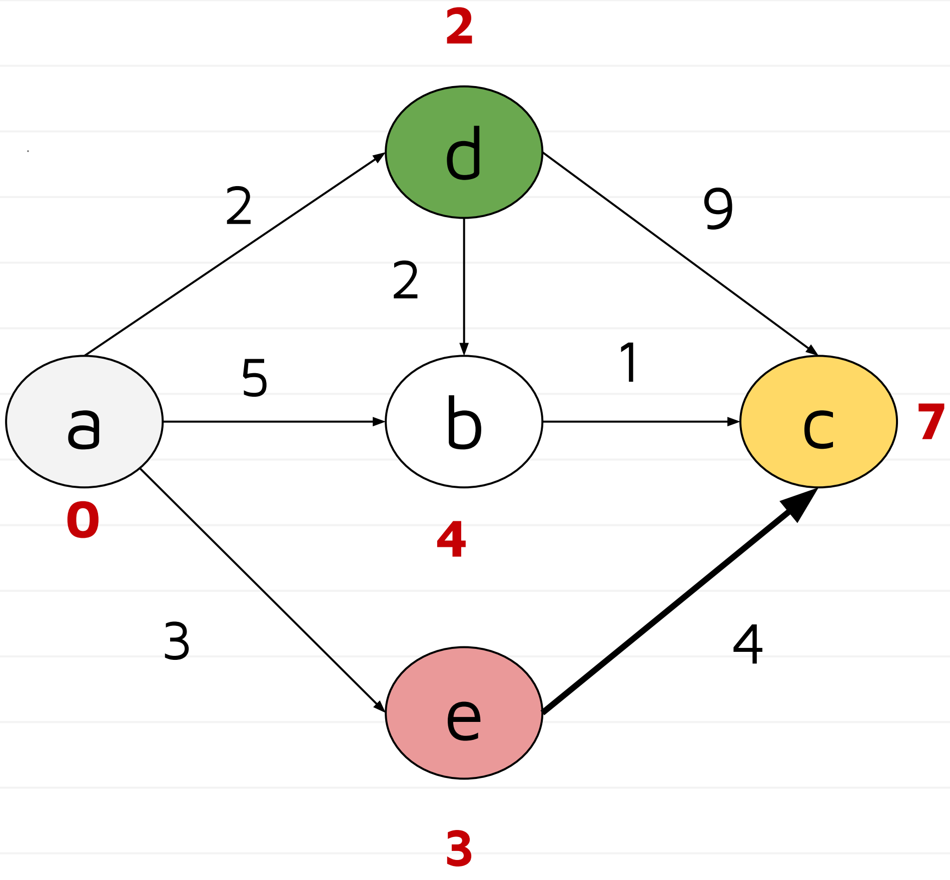
Update rule:

If 3 + 4 < 11, update

if d(u) + cost(u, v)< d(v):

d(v)=d(u) + cost(u, v)

Dijkstra's Algorithm



Select the **shortest** path: 3

Update rule:

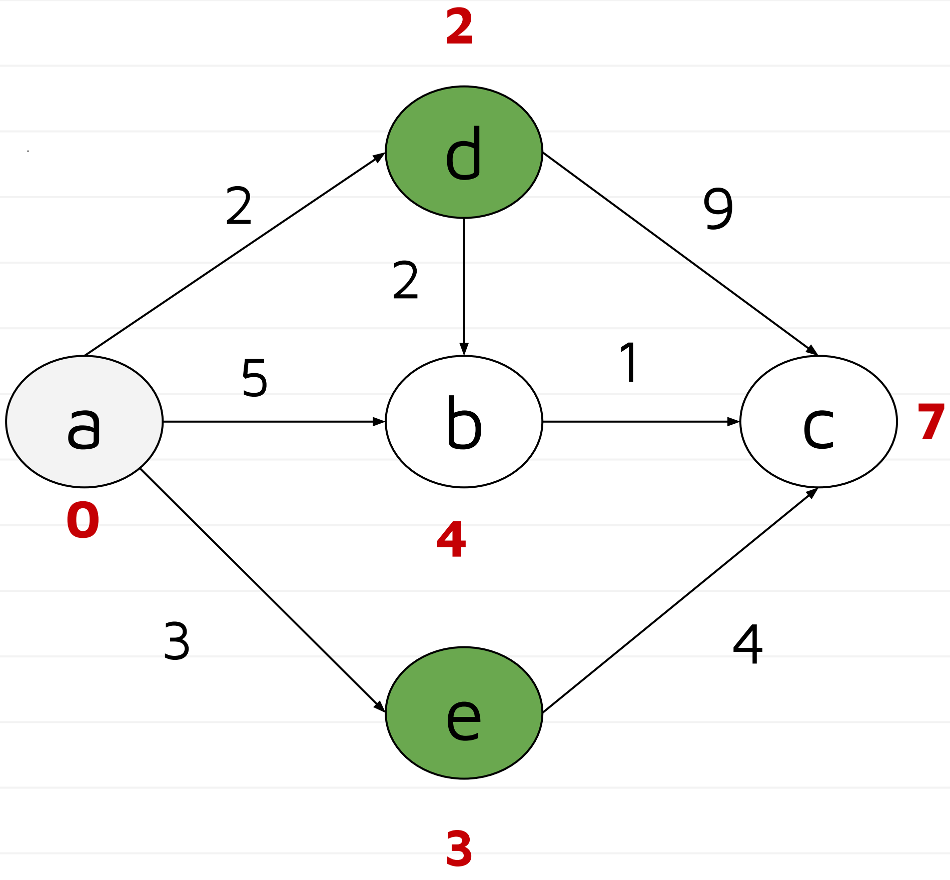
If 3 + 4 < 11, update

if d(u) + cost(u, v)< d(v):

d(v)=d(u) + cost(u, v)

Dijkstra's Algorithm

Select the **shortest** path: 4

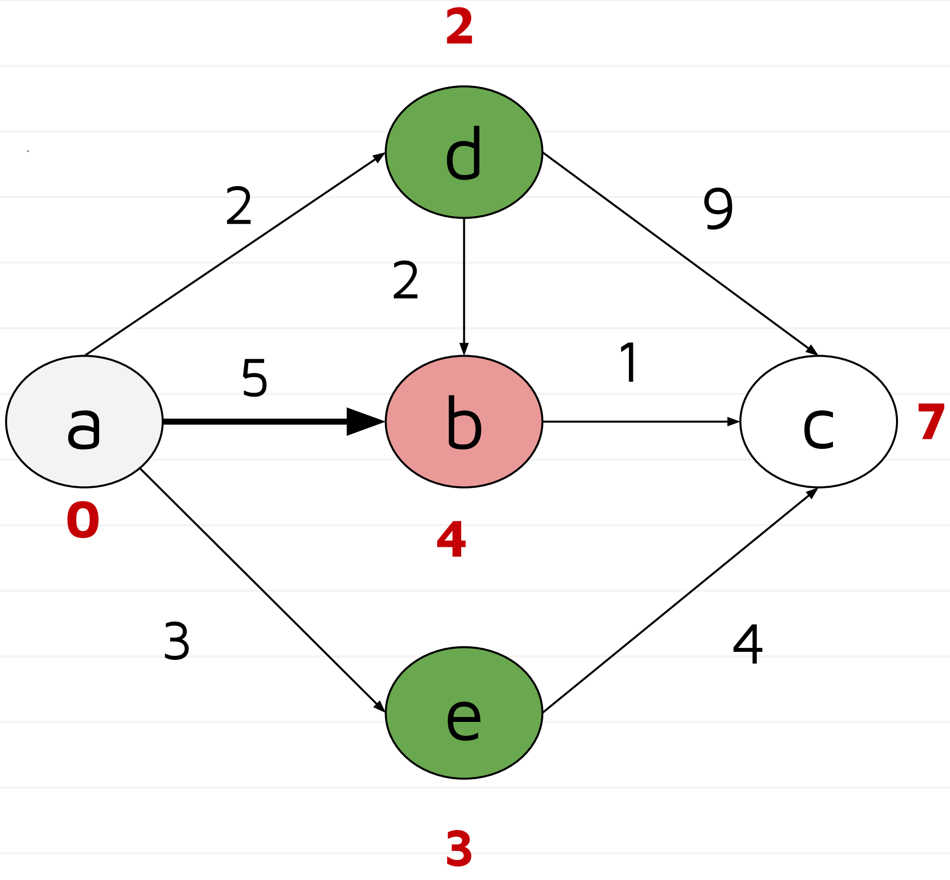


if d(u) + cost(u, v)< d(v):

d(v)=d(u) + cost(u, v)

Dijkstra's Algorithm

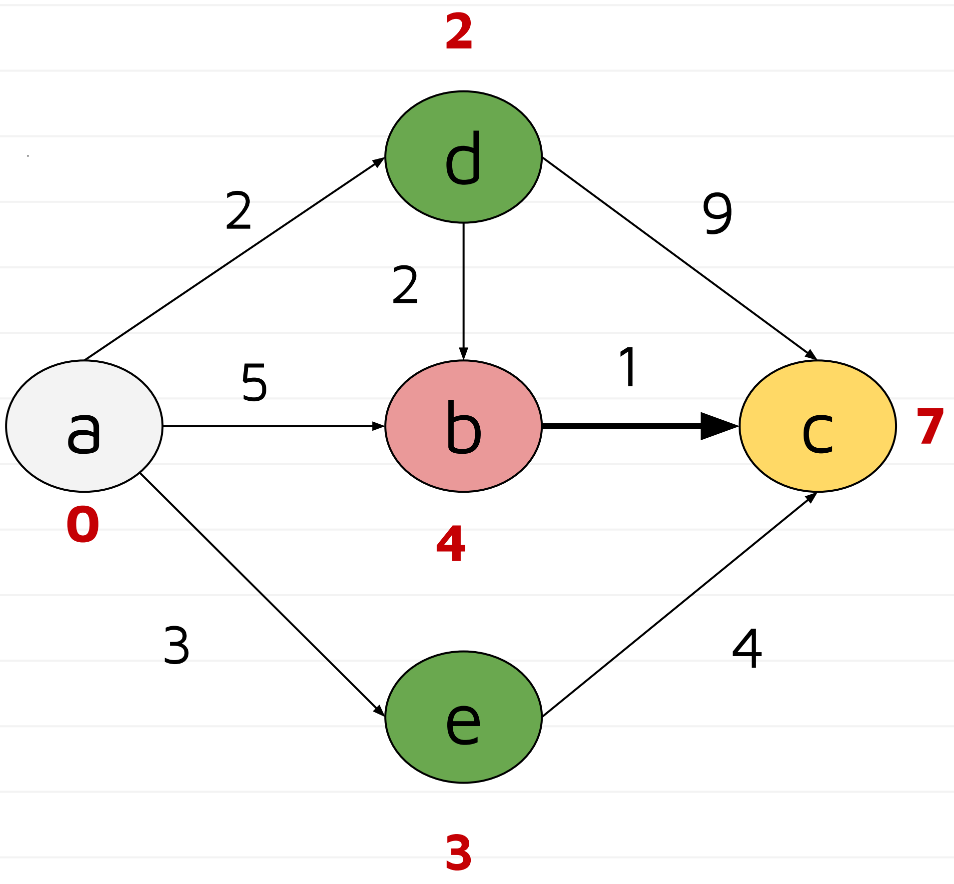
Select the **shortest** path: 4



if d(u) + cost(u, v)< d(v):

d(v)=d(u) + cost(u, v)

Dijkstra's Algorithm



Select the **shortest** path: 4

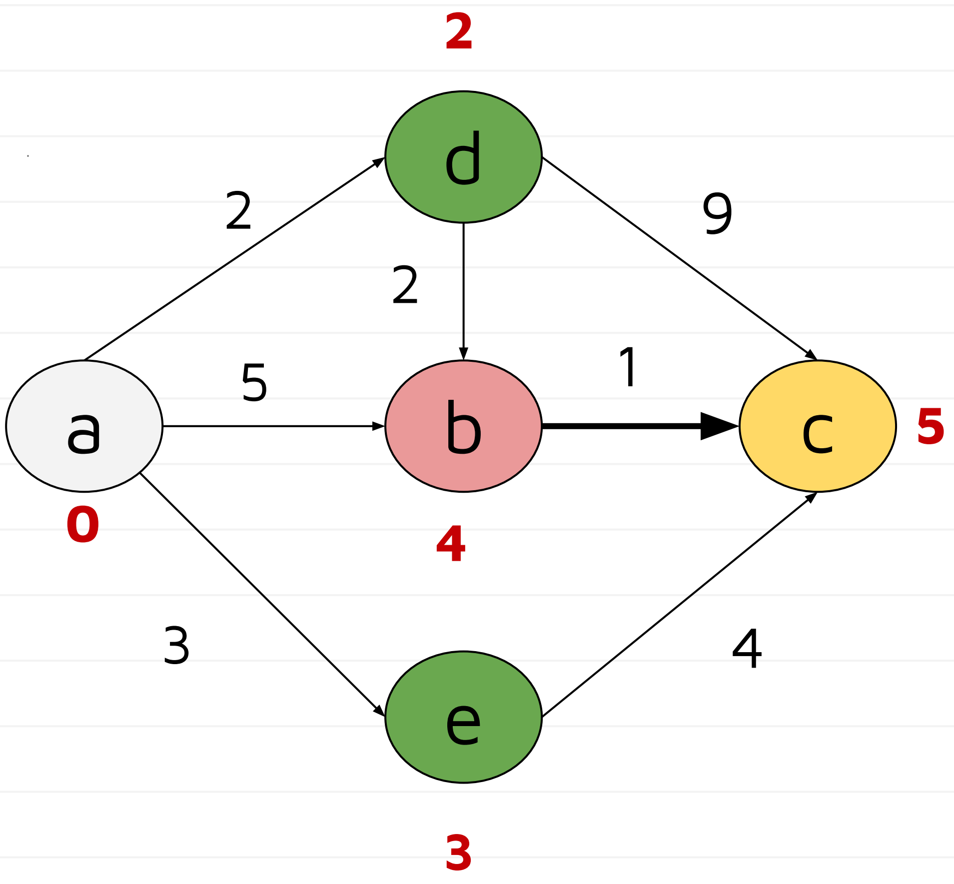
Update rule:

If 4 + 1 < 7, update

if d(u) + cost(u, v)< d(v):

d(v)=d(u) + cost(u, v)

Dijkstra's Algorithm



Select the **shortest** path: 4

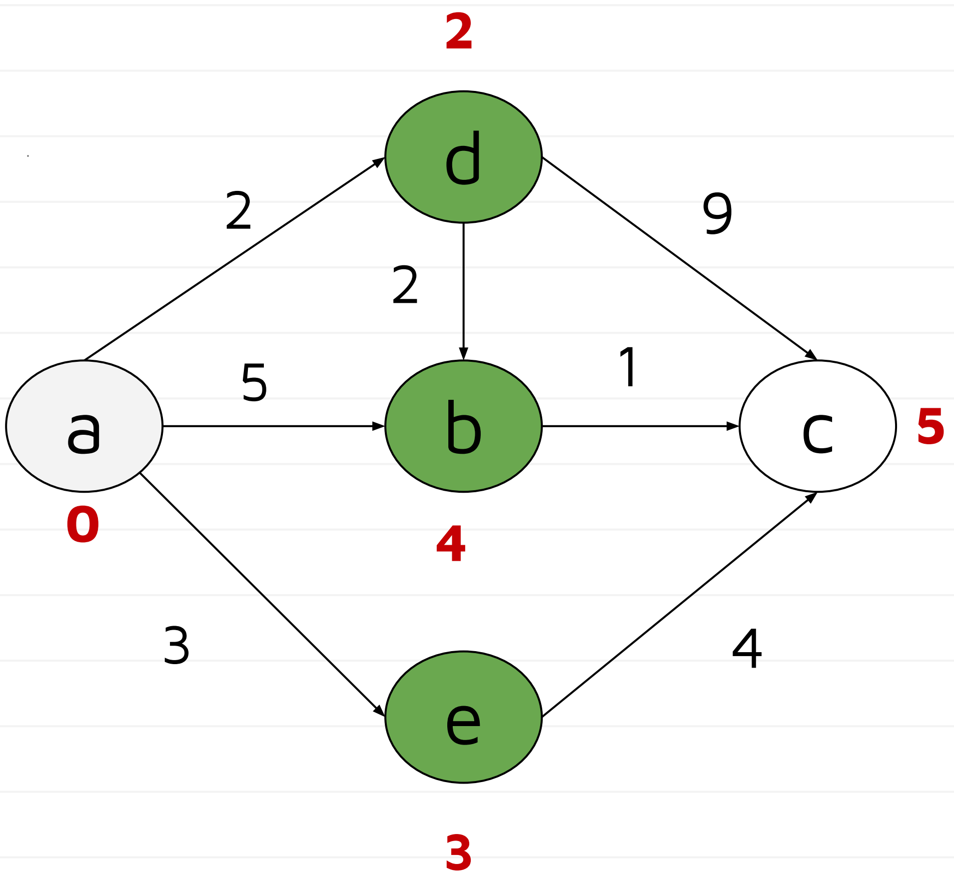
Update rule:

If 4 + 1 < 7, update

if d(u) + cost(u, v)< d(v):

d(v)=d(u) + cost(u, v)

Dijkstra's Algorithm



Select the **shortest** path: 4

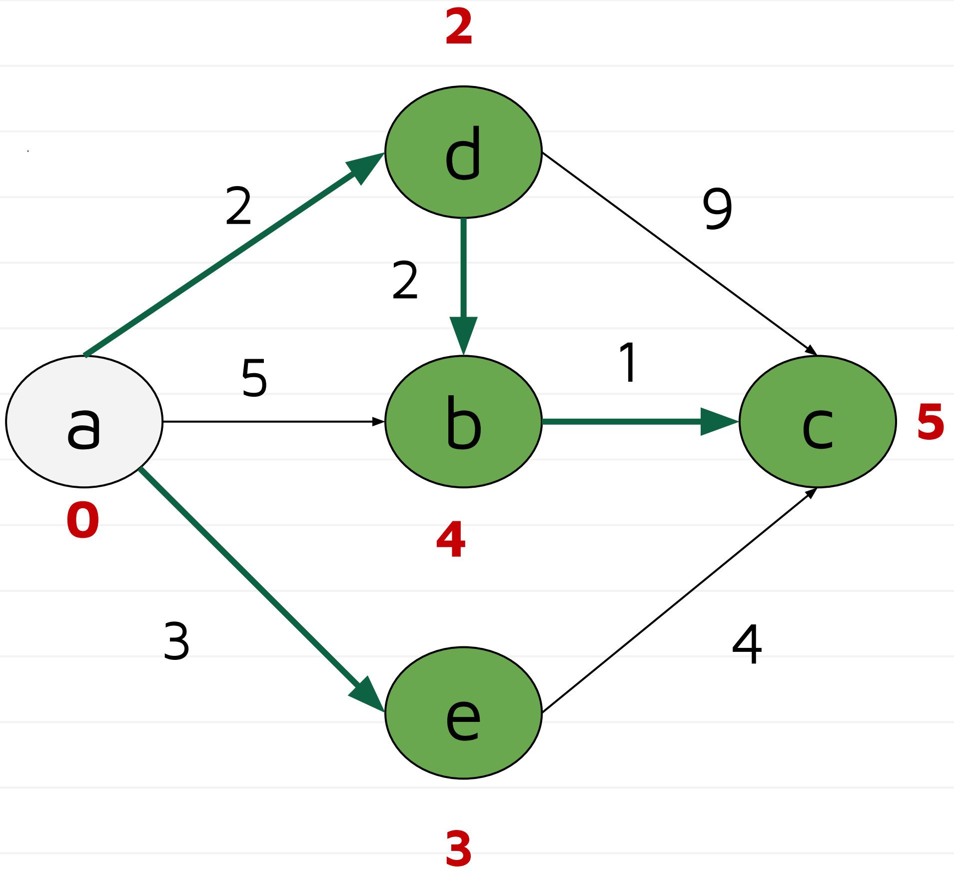
Update rule:

If 4 + 1 < 7, update

if d(u) + cost(u, v)< d(v):

d(v)=d(u) + cost(u, v)

Dijkstra's Algorithm



	a
b	4
c	5
d	2
e	3

if d(u) + cost(u, v)< d(v):

d(v)=d(u) + cost(u, v)

Dijkstra's Idea

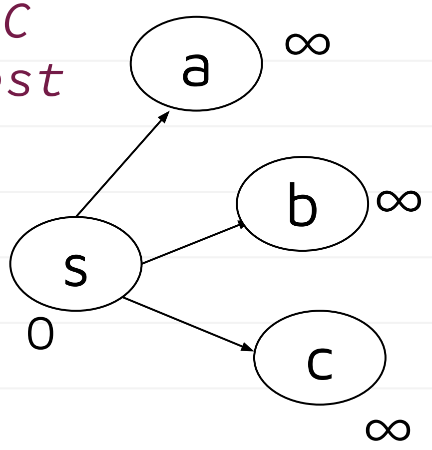
- Keep track of set C of "correct" nodes.
  - Nodes whose distance ***estimate*** is correct.
- At every step, add node *outside* of C with **smallest** estimated distance; update only its **neighbors**.
- A "**greedy**" algorithm.

Outline of Dijkstra's Algorithm

```python
def dijkstra(graph, weights, source):
    est = {node: float('inf') for node in graph.nodes}
    est[source] = 0
    pred = {node: None for node in graph.nodes}

    # empty set
    C = set()

    # while there are nodes still outside of C
        # find node u outside of C with smallest
        # estimated distance
    C.add(u)

    for v in graph.neighbors(u):
        update(u, v, weights, est, pred)

    return est, pred
```

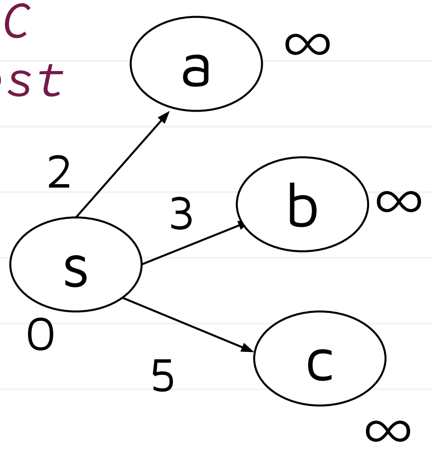
Outline of Dijkstra's Algorithm

```
def dijkstra(graph, weights, source):
    est = {node: float('inf') for node in graph.nodes}
    est[source] = 0
    pred = {node: None for node in graph.nodes}

    # empty set
    C = set()

    # while there are nodes still outside of C
        # find node u outside of C with smallest
        # estimated distance
    C.add(u)

    for v in graph.neighbors(u):
        update(u, v, weights, est, pred)

    return est, pred
```



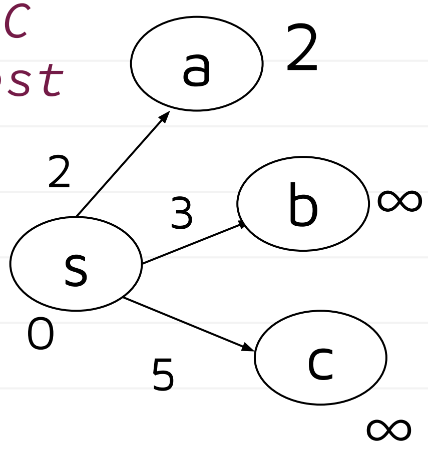
Outline of Dijkstra's Algorithm

```
def dijkstra(graph, weights, source):
    est = {node: float('inf') for node in graph.nodes}
    est[source] = 0
    pred = {node: None for node in graph.nodes}

    # empty set
    C = set()

    # while there are nodes still outside of C
        # find node u outside of C with smallest
        # estimated distance
    C.add(u)

    for v in graph.neighbors(u):
        update(u, v, weights, est, pred)

    return est, pred
```



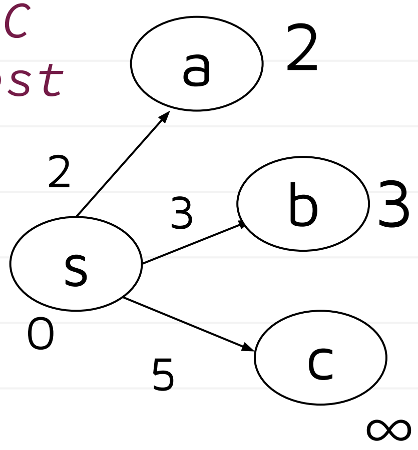
Outline of Dijkstra's Algorithm

```
def dijkstra(graph, weights, source):
    est = {node: float('inf') for node in graph.nodes}
    est[source] = 0
    pred = {node: None for node in graph.nodes}

    # empty set
    C = set()

    # while there are nodes still outside of C
        # find node u outside of C with smallest
        # estimated distance
    C.add(u)

    for v in graph.neighbors(u):
        update(u, v, weights, est, pred)

    return est, pred
```



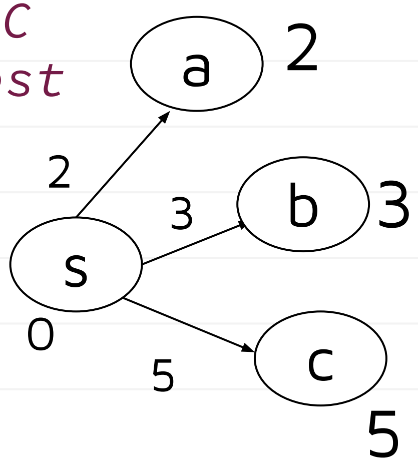
Outline of Dijkstra's Algorithm

```
def dijkstra(graph, weights, source):
    est = {node: float('inf') for node in graph.nodes}
    est[source] = 0
    pred = {node: None for node in graph.nodes}

    # empty set
    C = set()

    # while there are nodes still outside of C
        # find node u outside of C with smallest
        # estimated distance
    C.add(u)

    for v in graph.neighbors(u):
        update(u, v, weights, est, pred)

    return est, pred
```



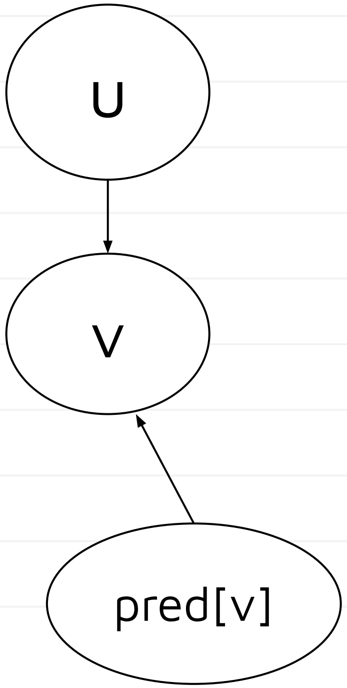
Outline of Dijkstra's Algorithm

```
def dijkstra(graph, weights, source):
    est = {node: float('inf') for node in graph.nodes}
    est[source] = 0
    pred = {node: None for node in graph.nodes}

    # empty set
    C = set()

    # while there are nodes still outside of C
        # find node u outside of C with smallest
        # estimated distance
    C.add(u)

    for v in graph.neighbors(u):
        update(u, v, weights, est, pred)

    return est, pred
```



Updating, in Code

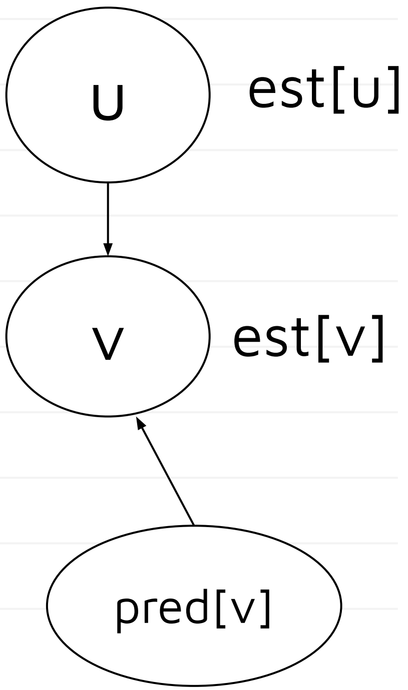
```
def update(u, v, weights, est, predecessor):
"""Update edge (u,v)."""
    if est[v] > est[u] + weights(u,v):
        est[v] = est[u] + weights(u,v)
        predecessor[v] = u
        return True
    else:
        return False
```



Time complexity: ?

Updating, in Code

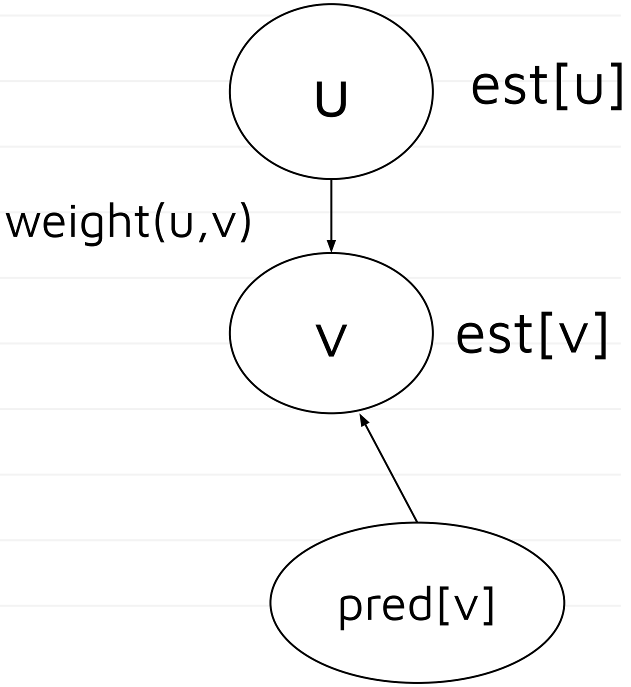
```
def update(u, v, weights, est, predecessor):
"""Update edge (u,v)."""
    if est[v] > est[u] + weights(u,v):
        est[v] = est[u] + weights(u,v)
        predecessor[v] = u
        return True
    else:
        return False
```



Time complexity: ?

Updating, in Code

```
def update(u, v, weights, est, predecessor):
"""Update edge (u,v)."""
    if est[v] > est[u] + weights(u,v):
        est[v] = est[u] + weights(u,v)
        predecessor[v] = u
        return True
    else:
        return False
```



Time complexity: ?

Updating, in Code

```
def update(u, v, weights, est, predecessor):
"""Update edge (u,v)."""
    if est[v] > est[u] + weights(u,v):
        est[v] = est[u] + weights(u,v)
        predecessor[v] = u
        return True
    else:
        return False
```

Time complexity: ?

A: $\Theta(1)$

B: $\Theta(E)$

C: $\Theta(V)$

D: $\Theta(V + E)$

Updating, in Code

```
def update(u, v, weights, est, predecessor):
"""Update edge (u,v)."""
    if est[v] > est[u] + weights(u,v):
        est[v] = est[u] + weights(u,v)
        predecessor[v] = u
        return True
    else:
        return False
```

Time complexity: $\Theta(1)$

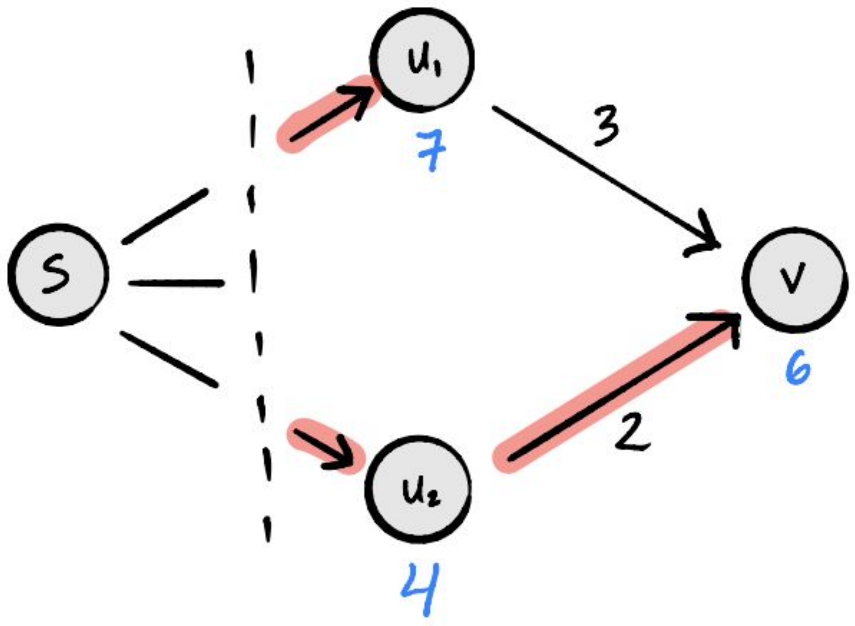
When does an update discover a shortest path?

Suppose updating (u_2, v) finds a shorter path to v.

True or **False**: the actual shortest path must go through u_2.

A: True

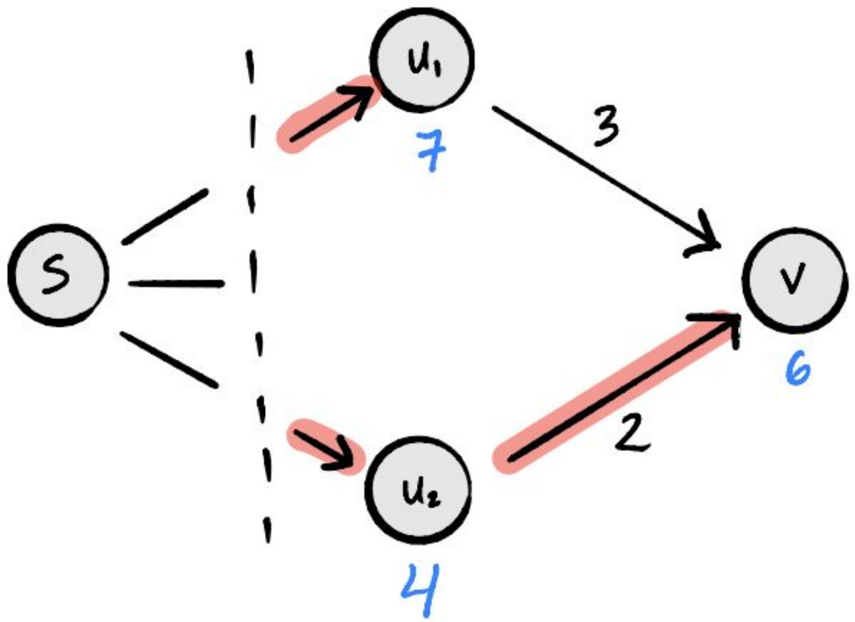
B: False



When does an update discover a shortest path?

Suppose updating (u_2, v) finds a shorter path to v.

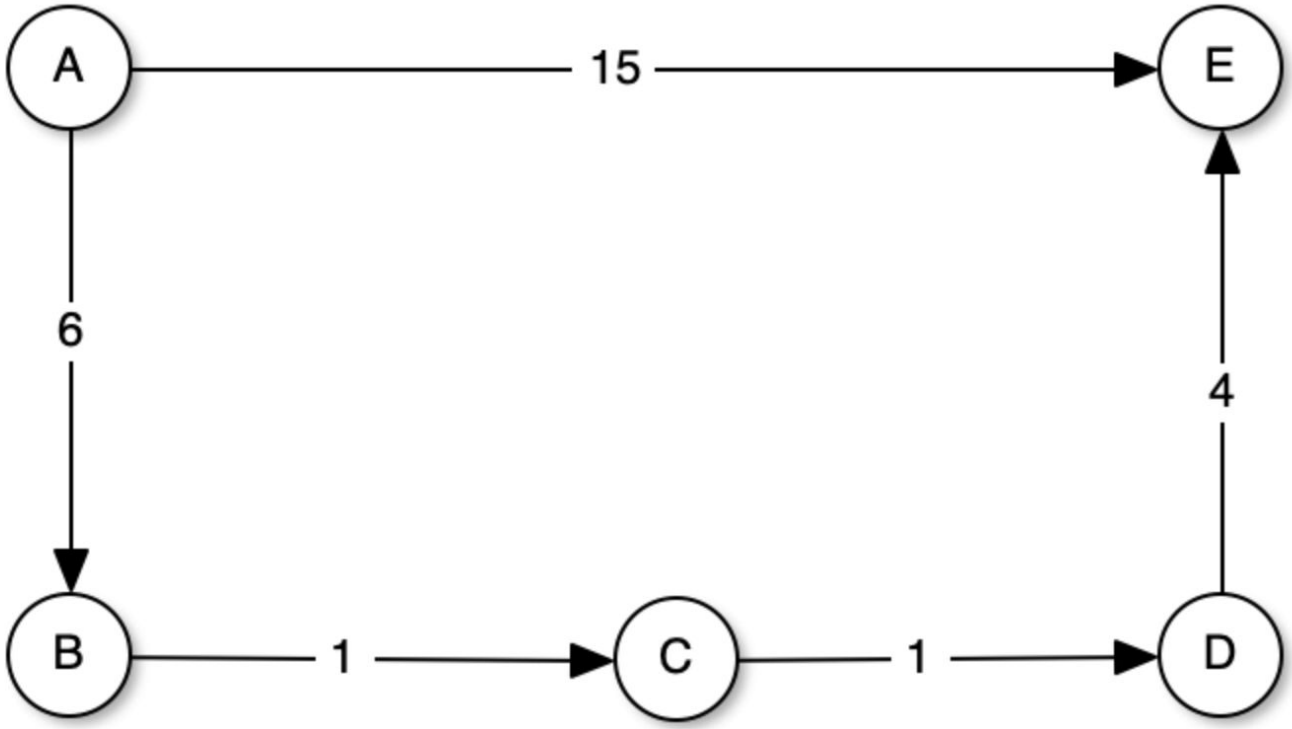
True or **False**: the actual shortest path must go through u_2.

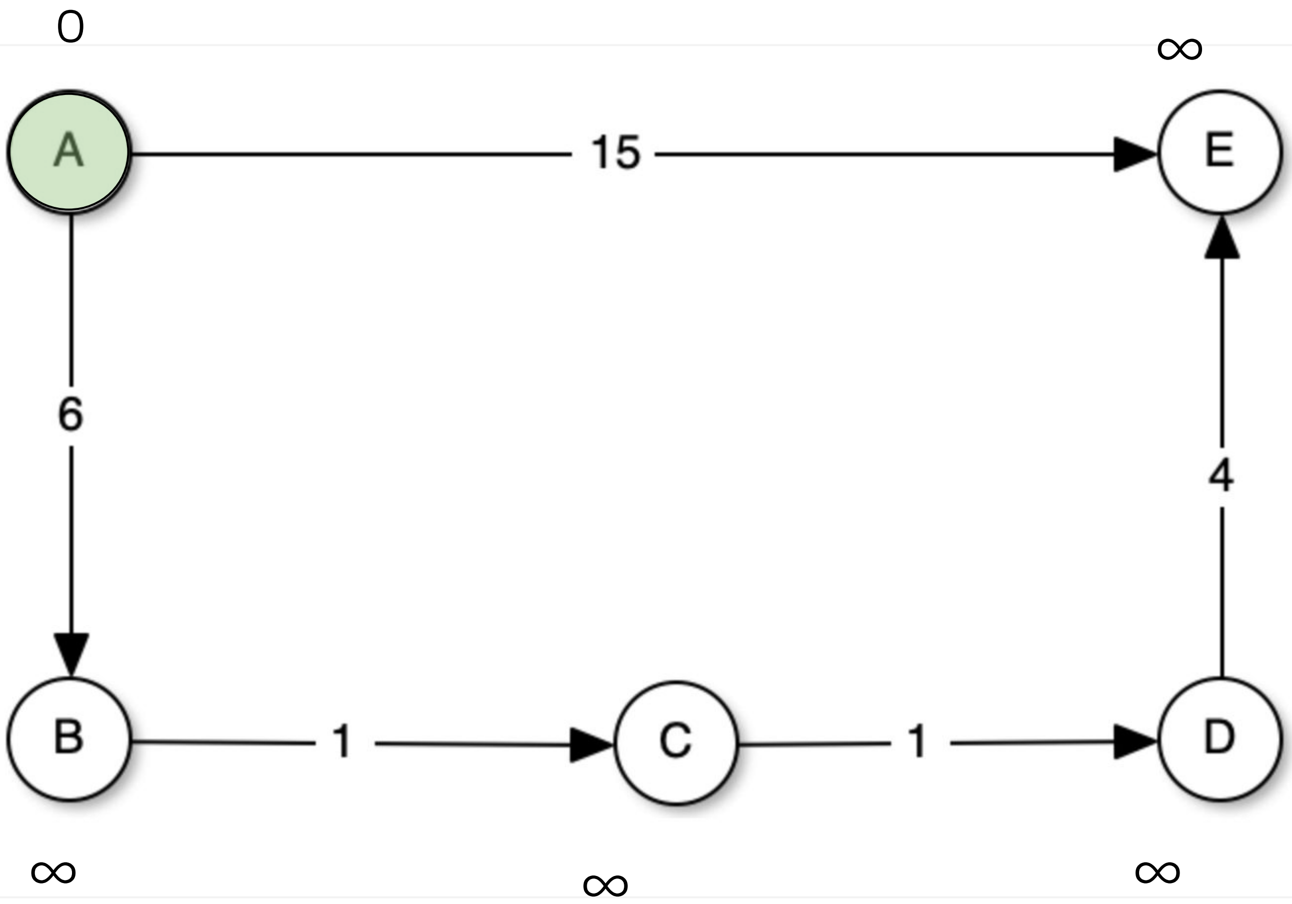


False: we might later discover a better path to u_1

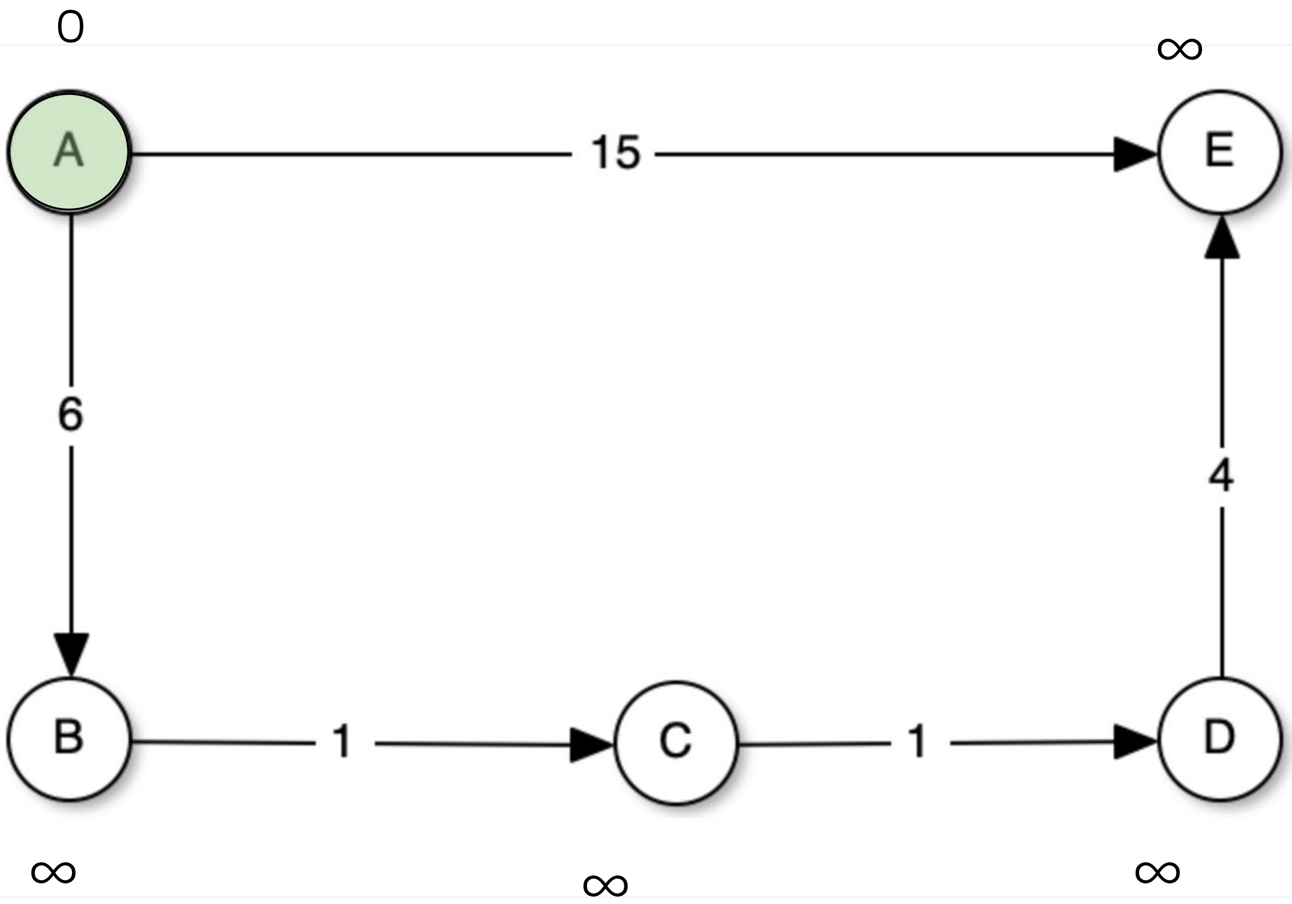
When does an update discover a shortest path?

- Let (u, v) be an edge we update.
- Suppose:
  - the **actual** shortest path to u has been found;
  - the **actual** shortest path to v goes through (u, v).
- Then after *updating* (u, v), the estimated shortest path to v is **correct**.

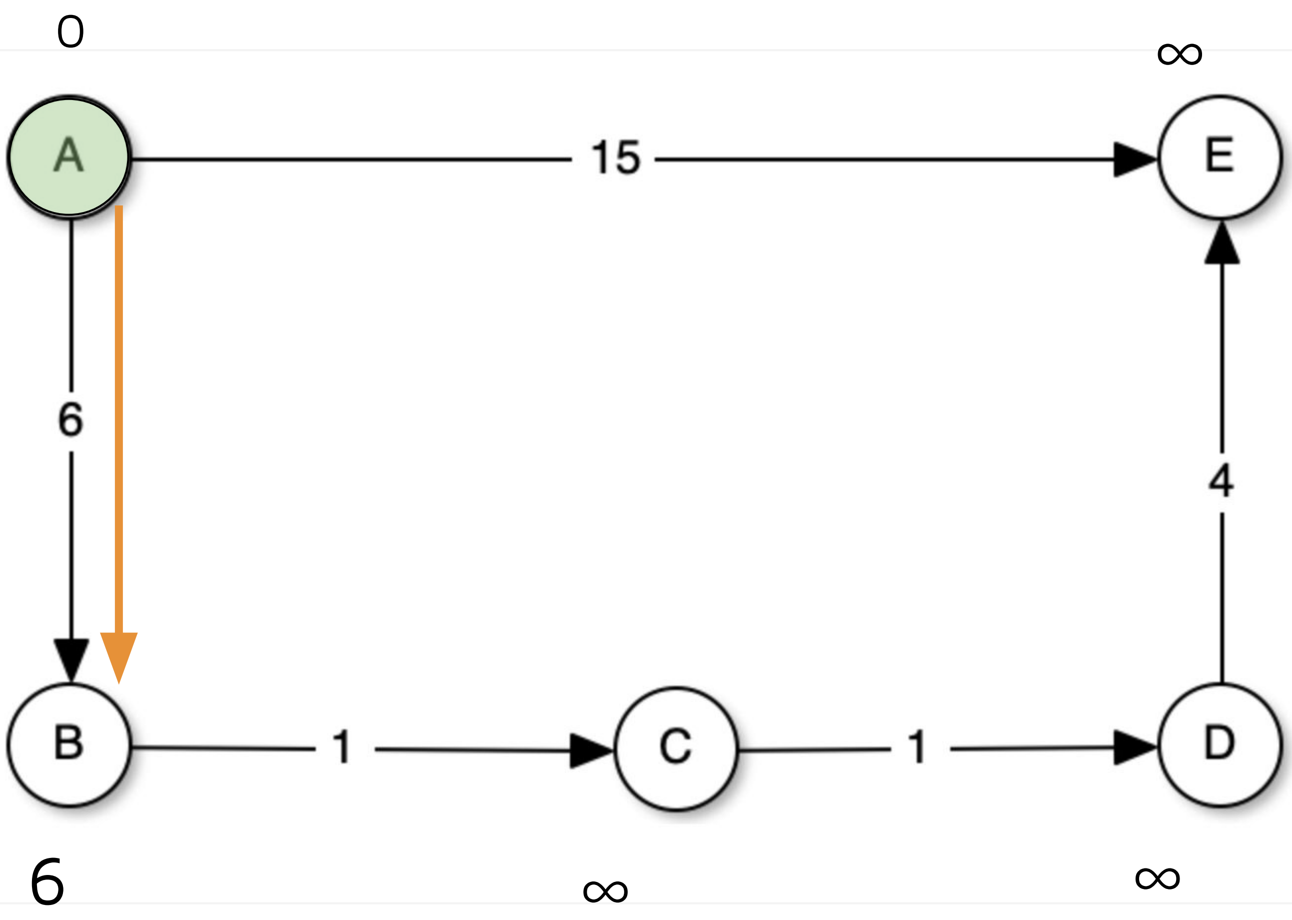

A
E
15
6
4
B
C
D
1
1



C = { }

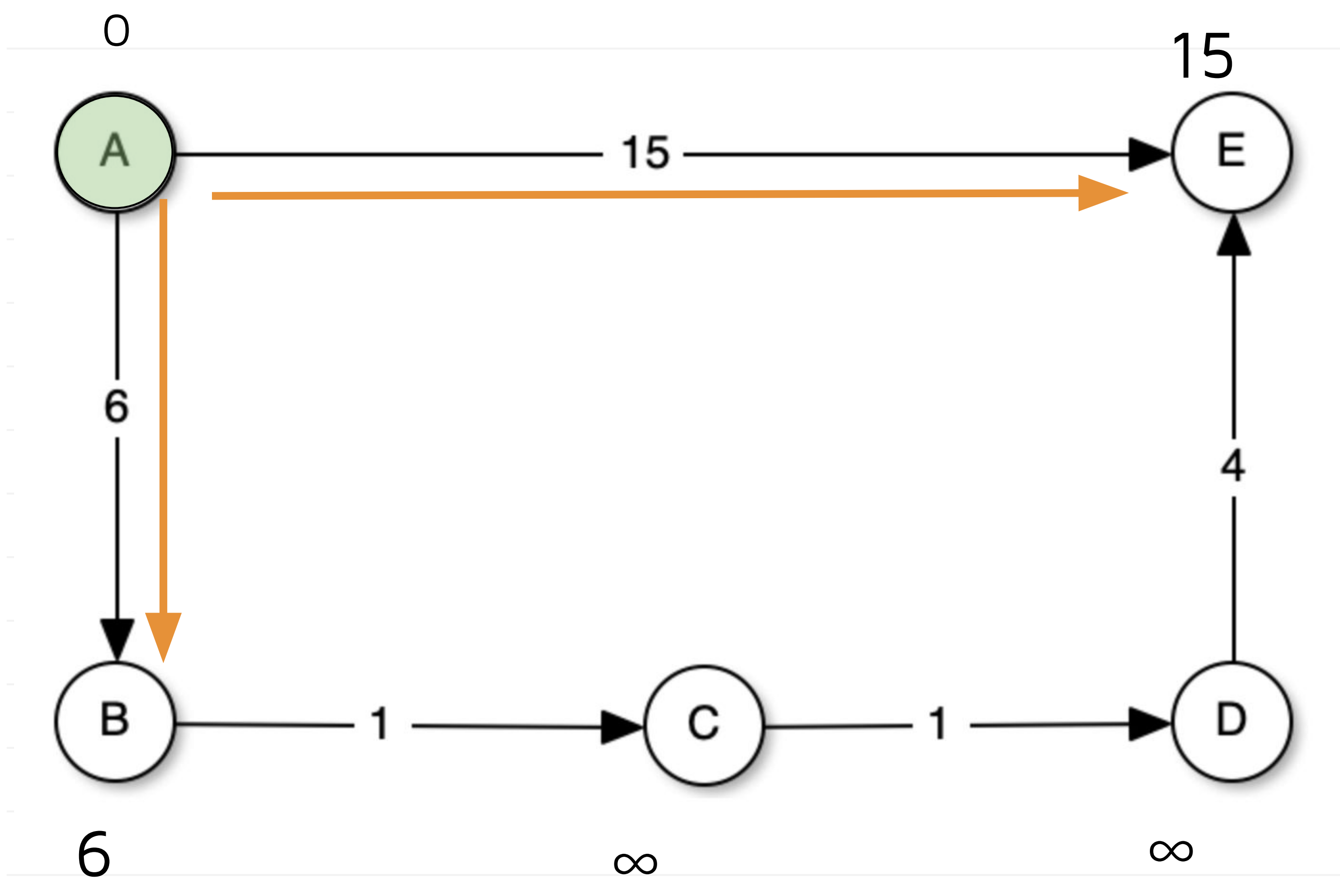


C = { A }

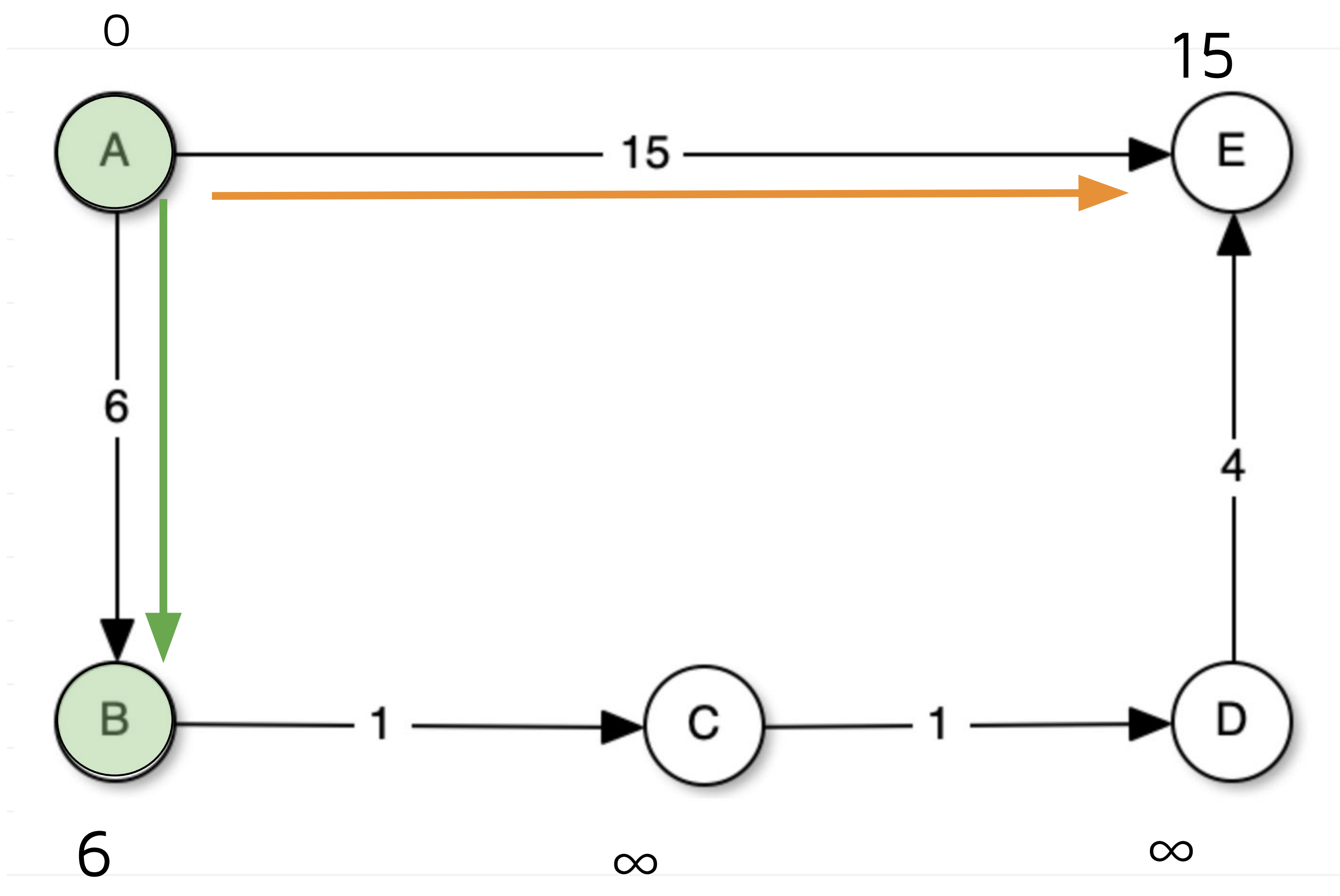


C = { A }

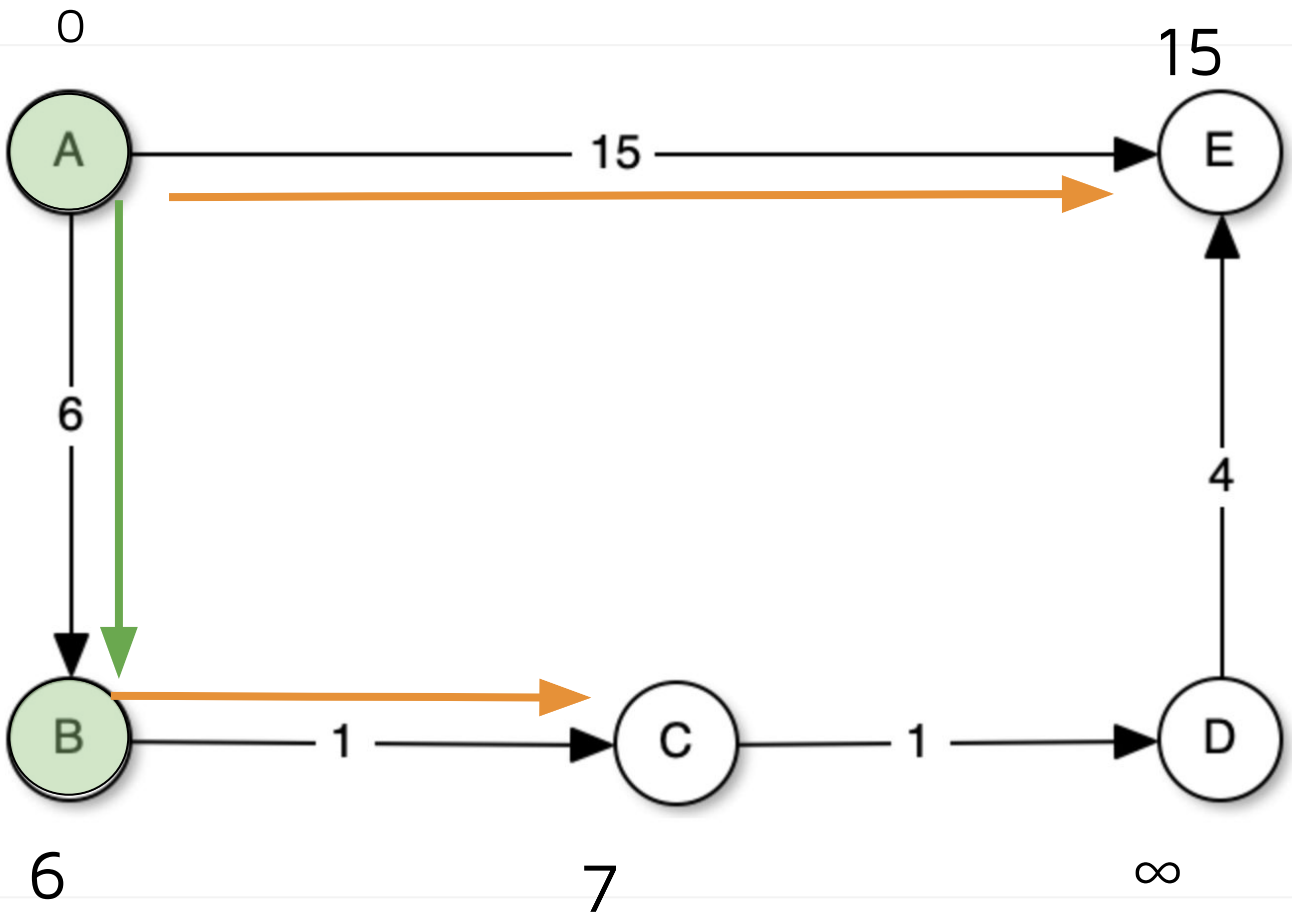
pred =
{ B:A,



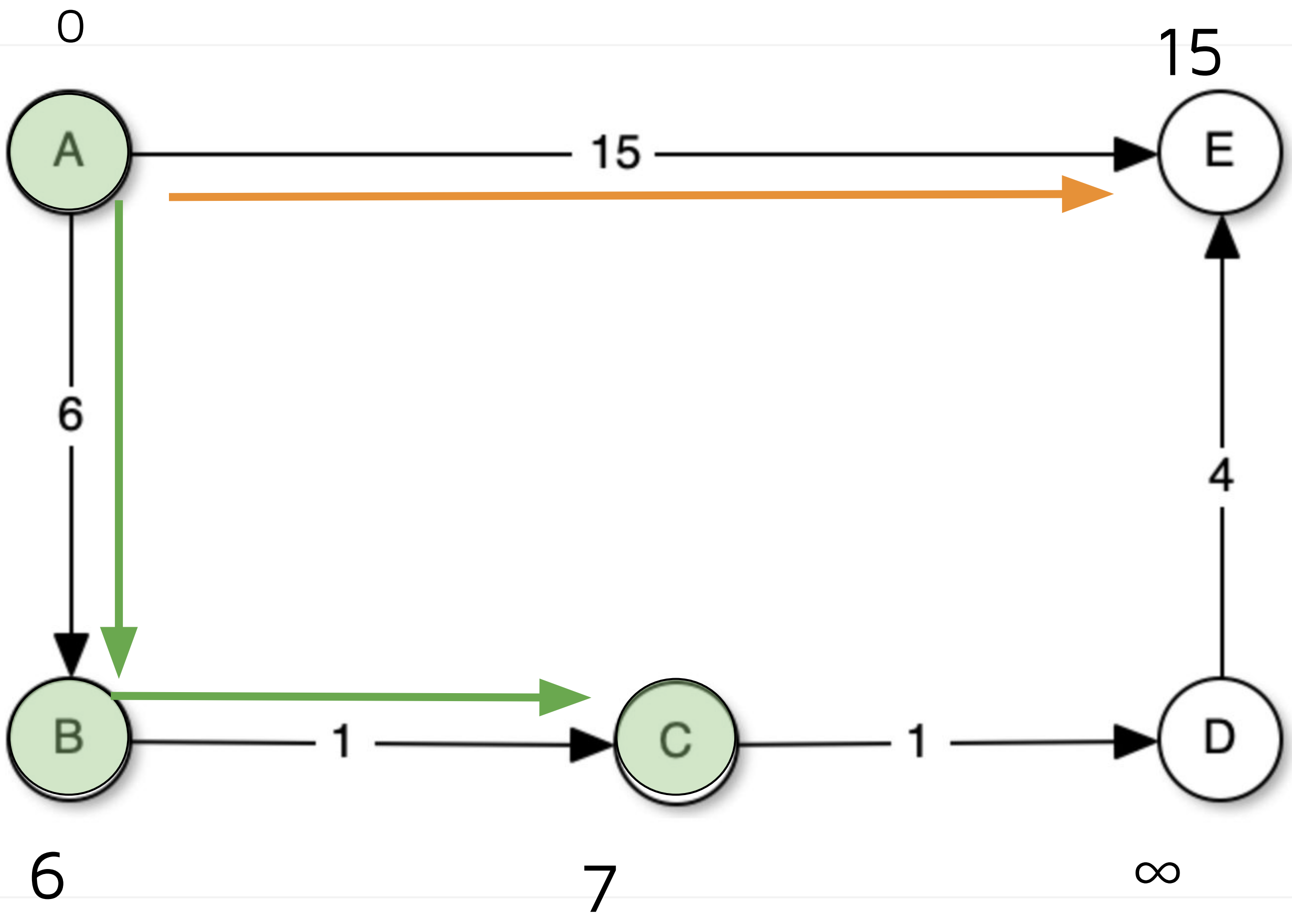

```
C = { A }

pred =
    { B:A,
      E:A,#fornow
```




```
C = { A, B }

pred =
    { B:A,
      E:A,#fornow
```

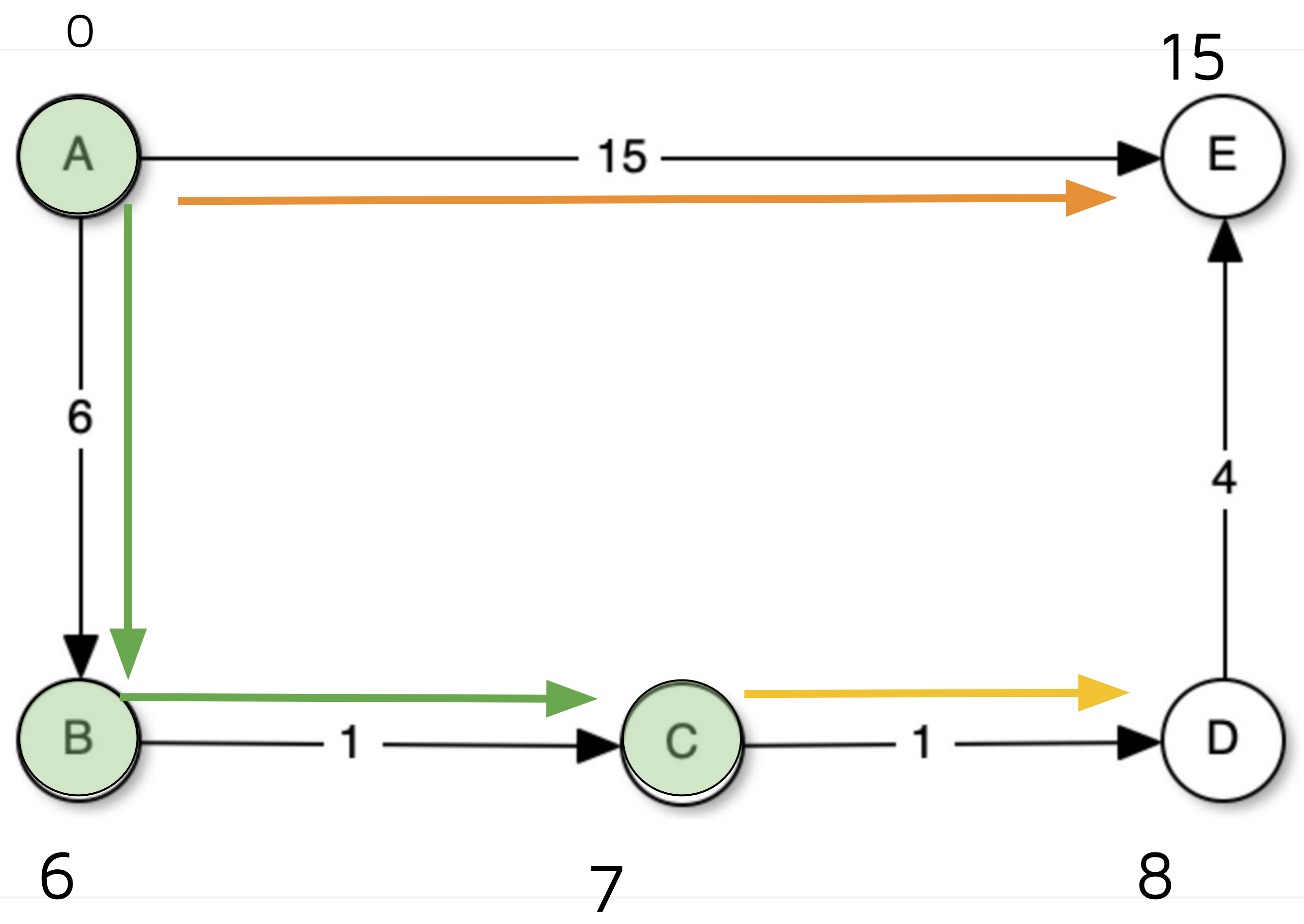



```
C = { A, B }

pred =
    { B:A,
      E:A, #fornow
      C:B,
```



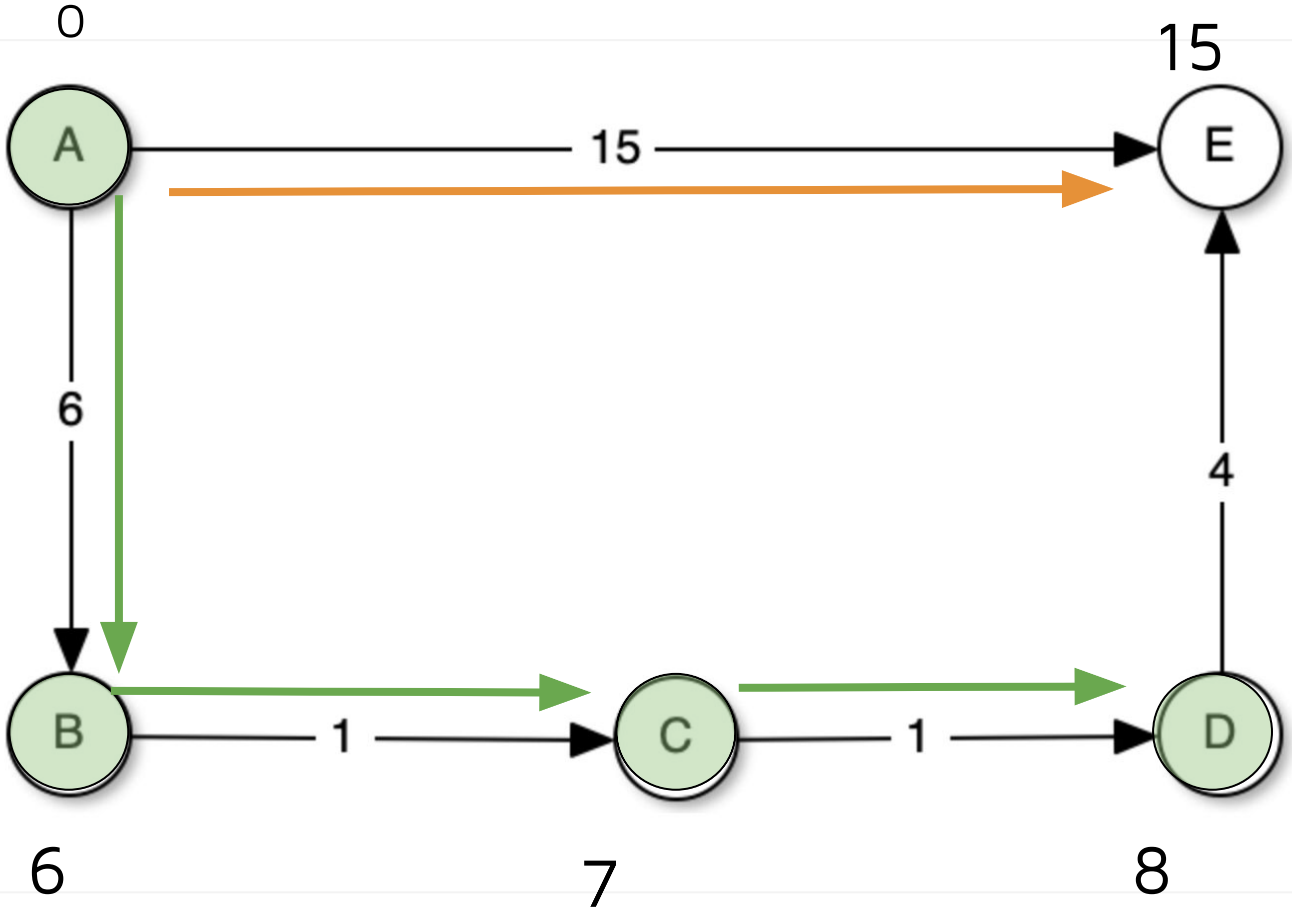
C = { A, B, C }

```
pred =
    { B:A,
      E:A,#fornow
      C:B,
```



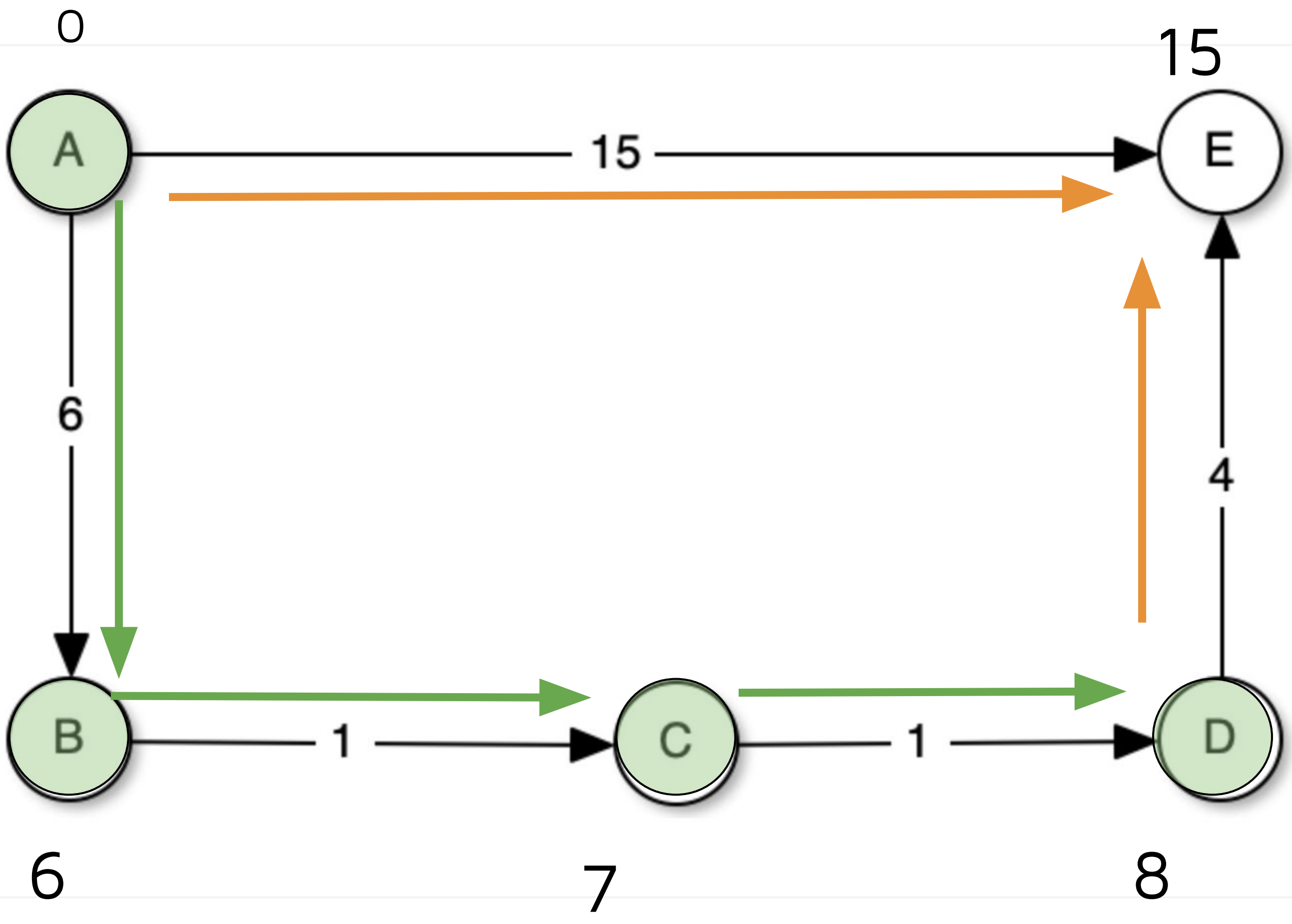
C = { A, B, C }

```
pred =
    { B:A,
      E:A,#fornow
      C:B,
      D:C
```



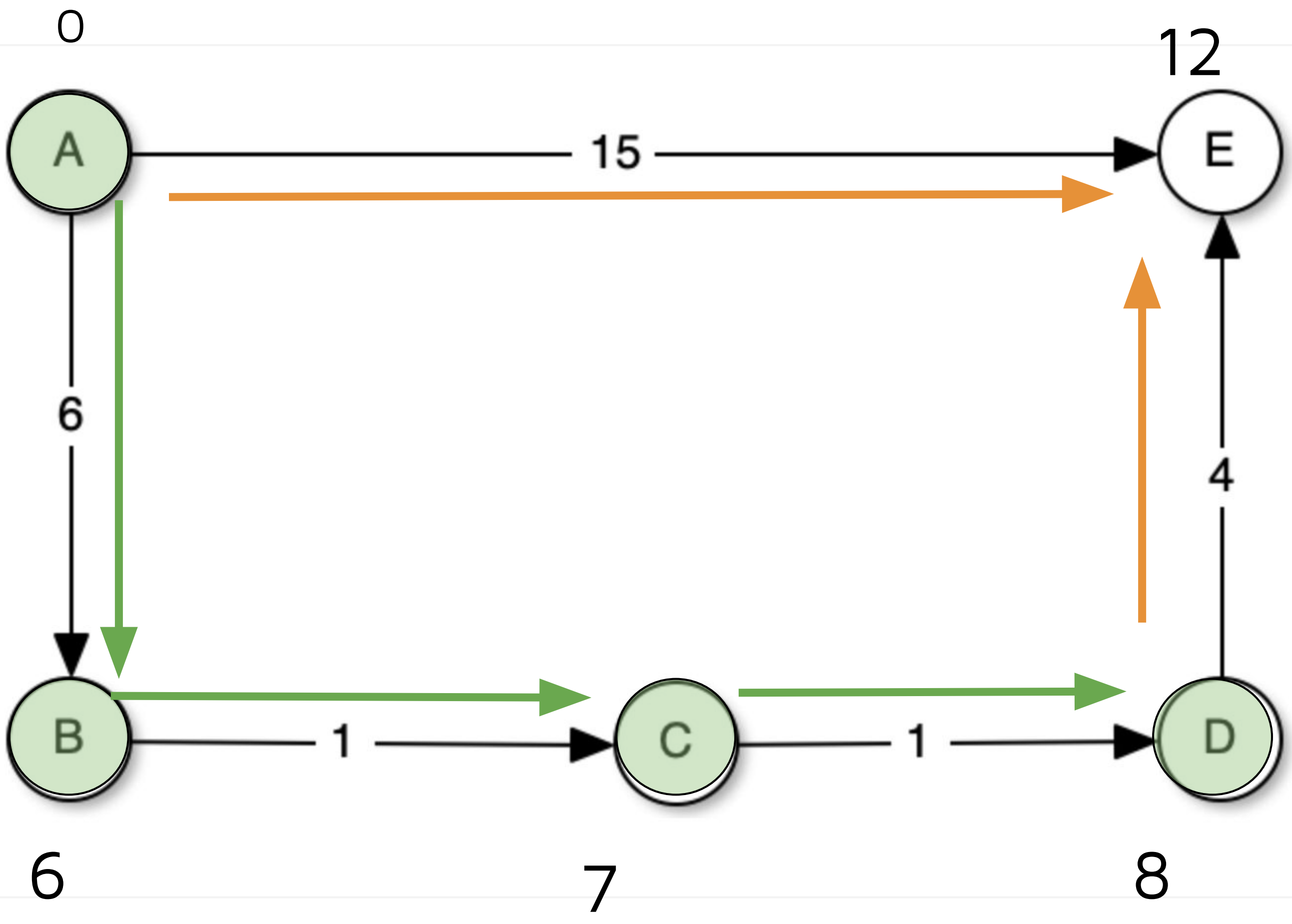
C = { A, B, C, D }

```
pred =
    { B:A,
      E:A,#fornow
      C:B,
      D:C
```



C = { A, B, C, D }

```
pred =
    { B:A,
      E:A,#fornow
      C:B,
      D:C
```



C = { A, B, C, D }

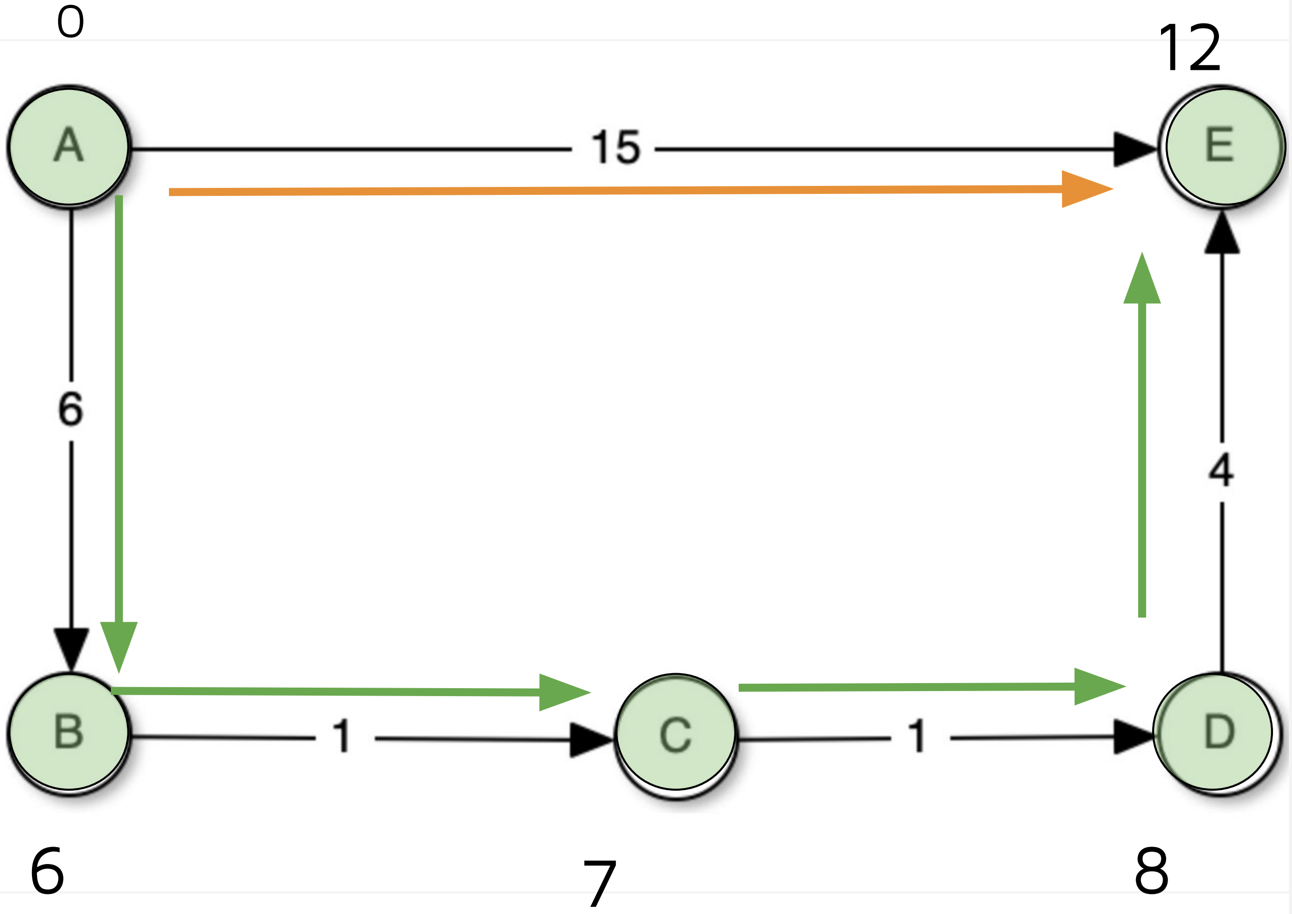
pred =

{ B:A,

E:D,

C:B,

D:C



How many times do we update ("touch") each edge?

A: Once

B: Twice

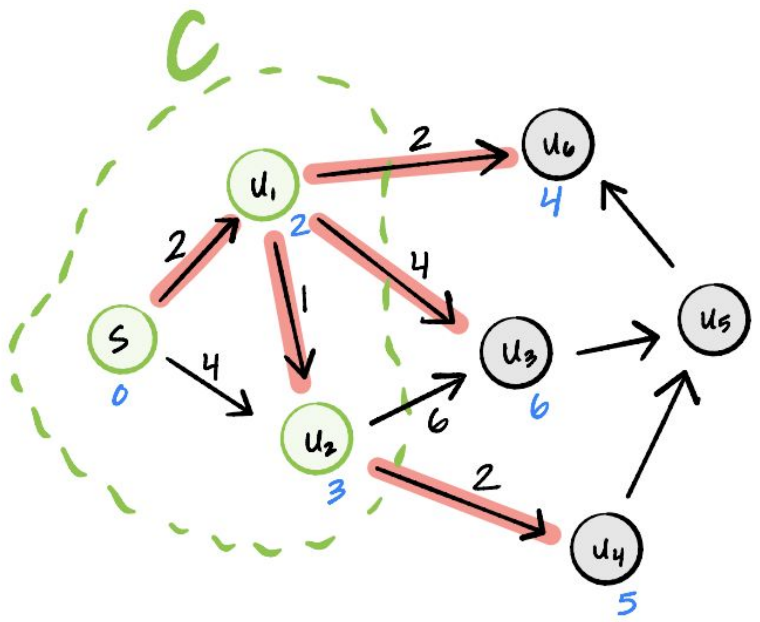
C: |E|

D: |V|

E: Something else

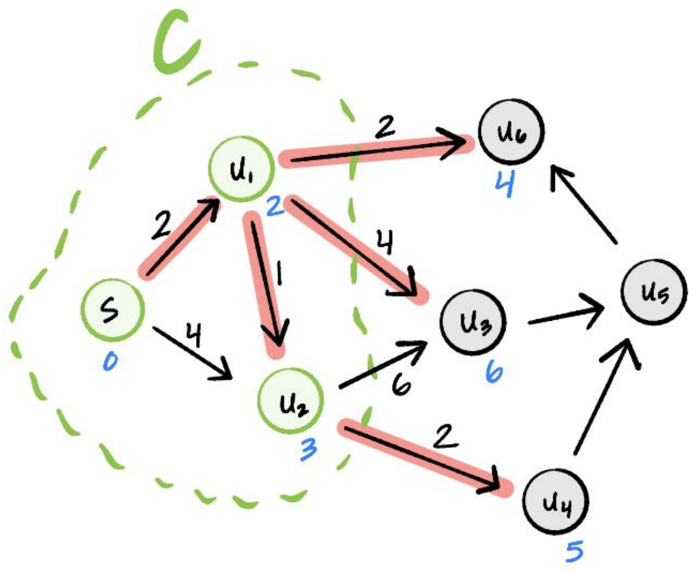
Proof Idea

Claim: at beginning of any iteration of Dijkstra's, if u is node $\notin C$ with smallest estimated distance, the shortest path to u has been correctly discovered.



Proof Idea

- Let u be node outside of C for which est[u] is smallest.
- We've discovered a path from s to u of length est[u].
- Any path from s to u has to exit C somewhere.
- Any path from s to u will cost at least est[u] just to exit C.



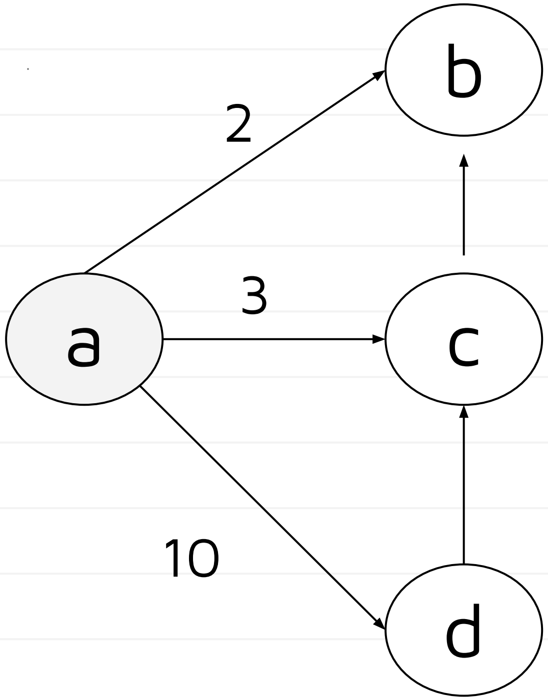
Question

- Is there a limitation for Dijkstra’s algorithm?

Exercise

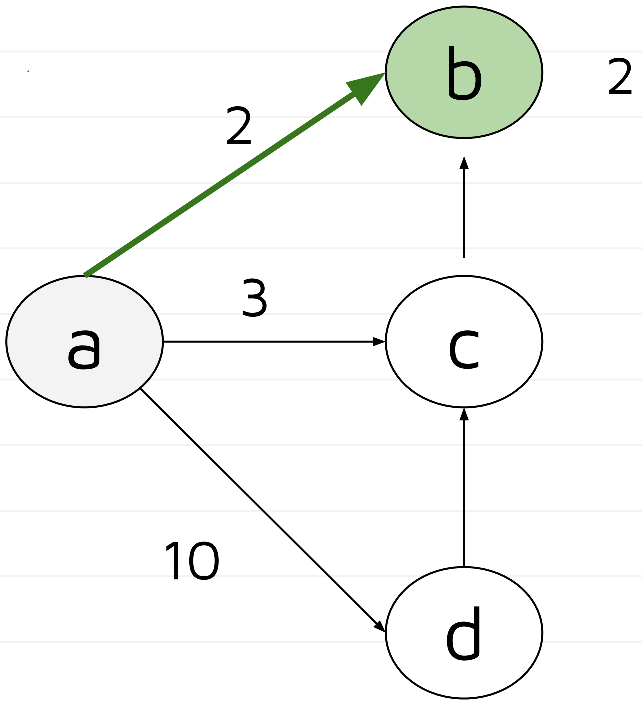
- Why do the edge weights need to be positive?
- Come up with a simple example graph with some negative edge weights where Dijkstra's **fails** to compute the correct shortest path.

Example



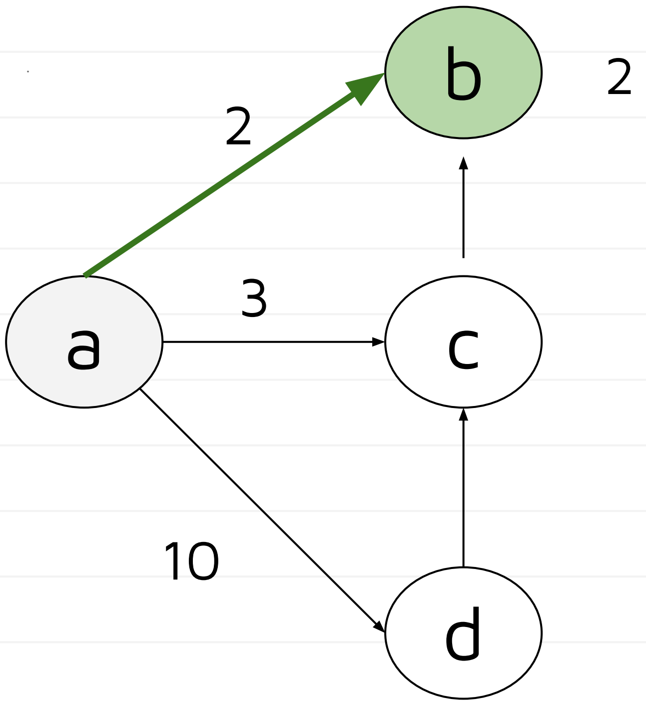
What is the fastest way to get to **b**?

Example



What is the fastest way to get to **b**?

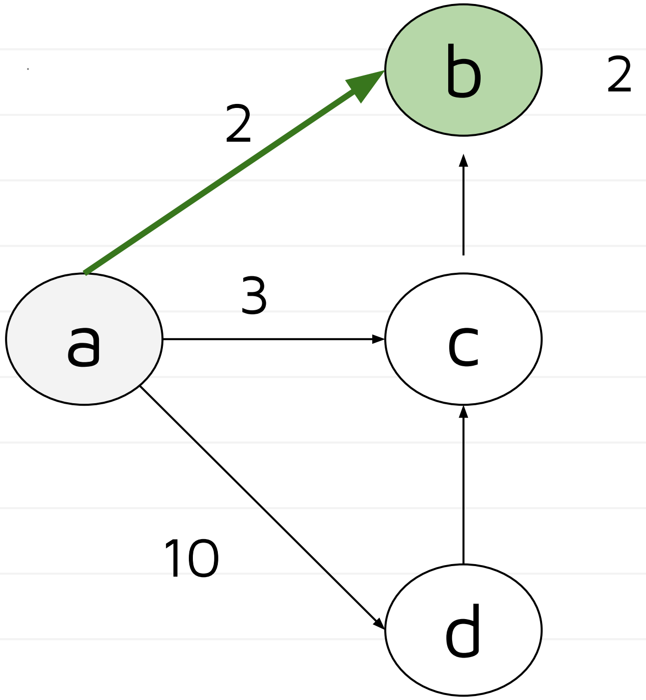
Example



What is the fastest way to get to **b**?

Do we ever have to consider another path to **b**?

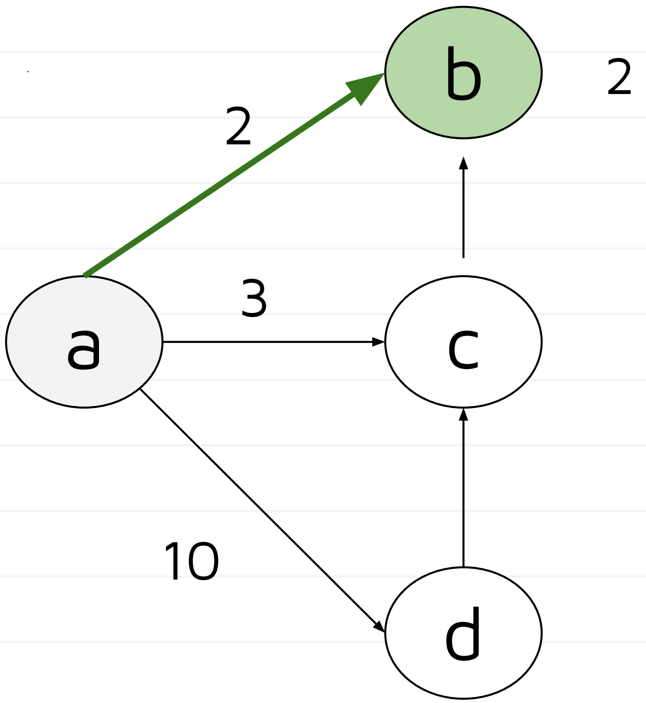
Example



What is the fastest way to get to **b**?

Do we ever have to consider another path to **b**? No.

Example

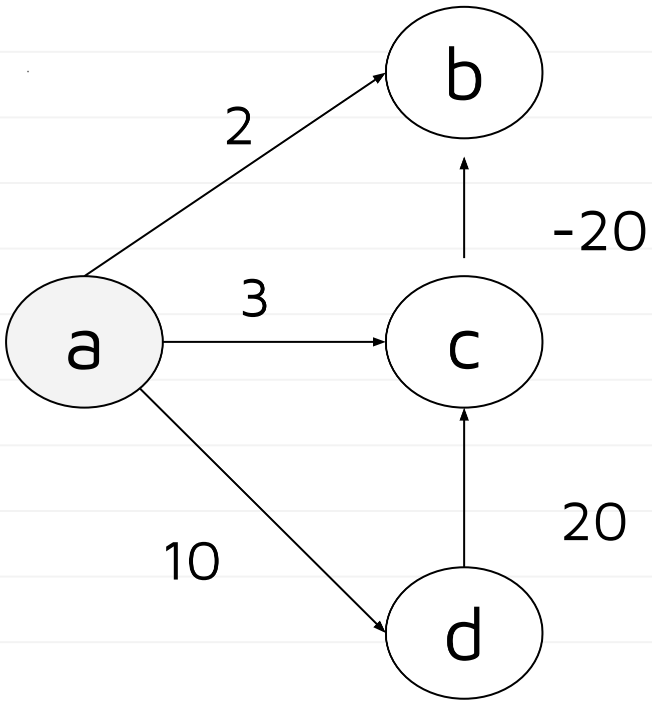


What is the fastest way to get to **b**?

Do we ever have to consider another path to **b**? No.

Let's make another path that is **better** than 2.

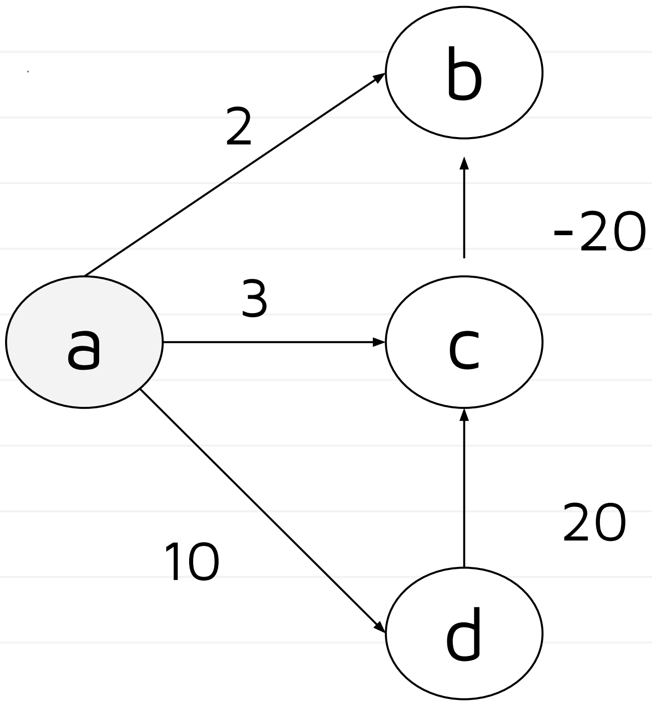
Example



Let's make another path that is **better** than 2.

Will the algorithm work for these weights?

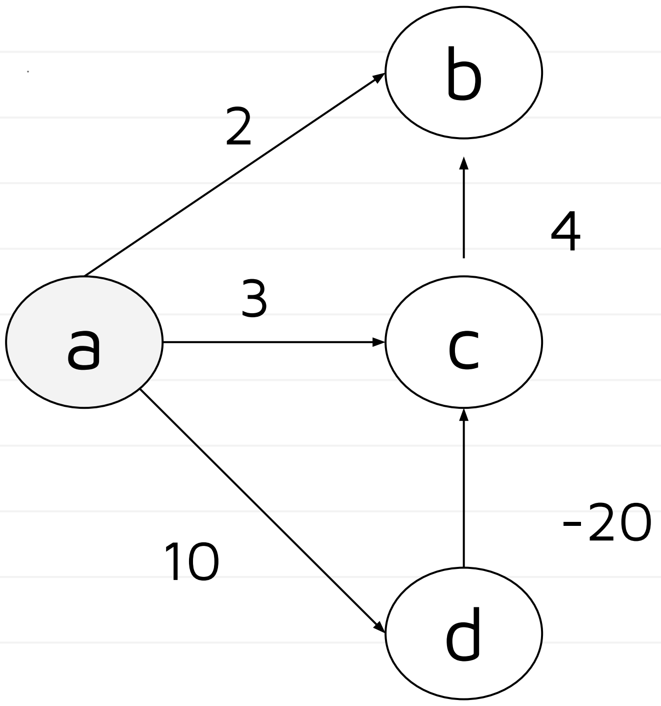
Example



Let's make another path that is **better** than 2.

Will the algorithm work for these weights? **Yes.**

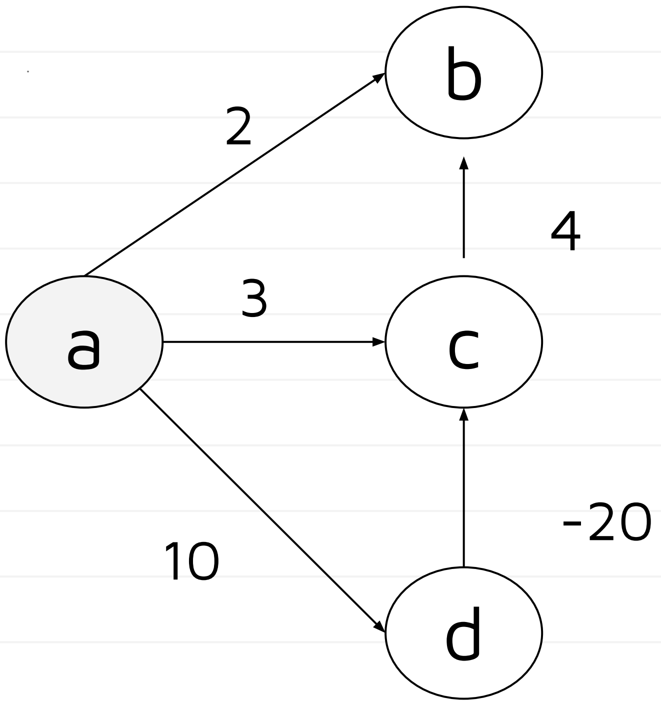
Example



Let's make another path that is **better** than 2.

Will the algorithm work for these weights?

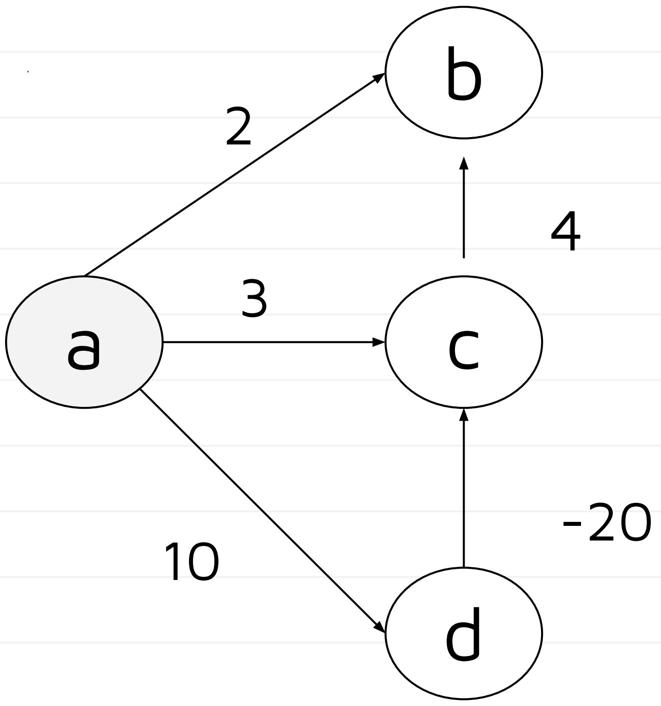
Example



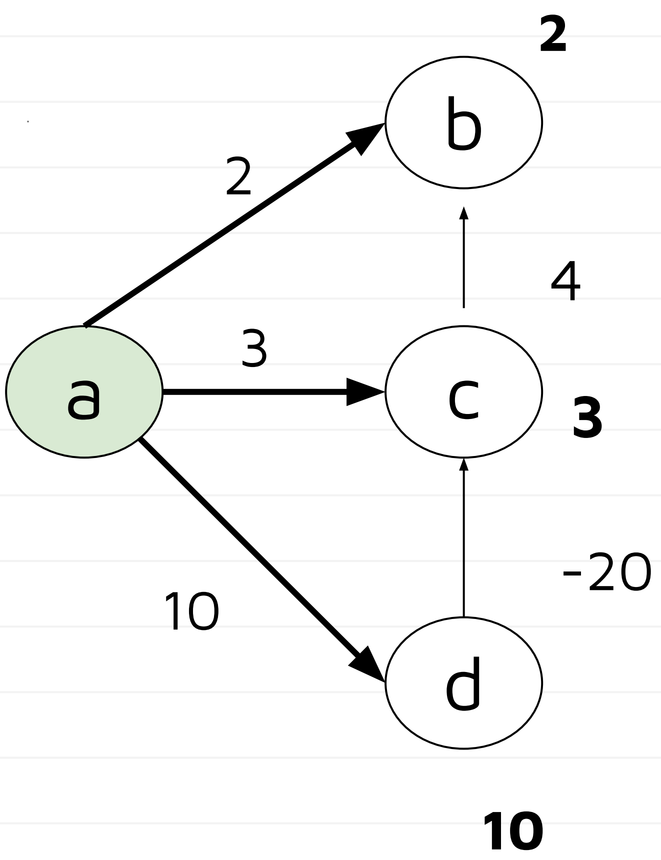
Let's make another path that is **better** than 2.

Will the algorithm work for these weights? **No.**

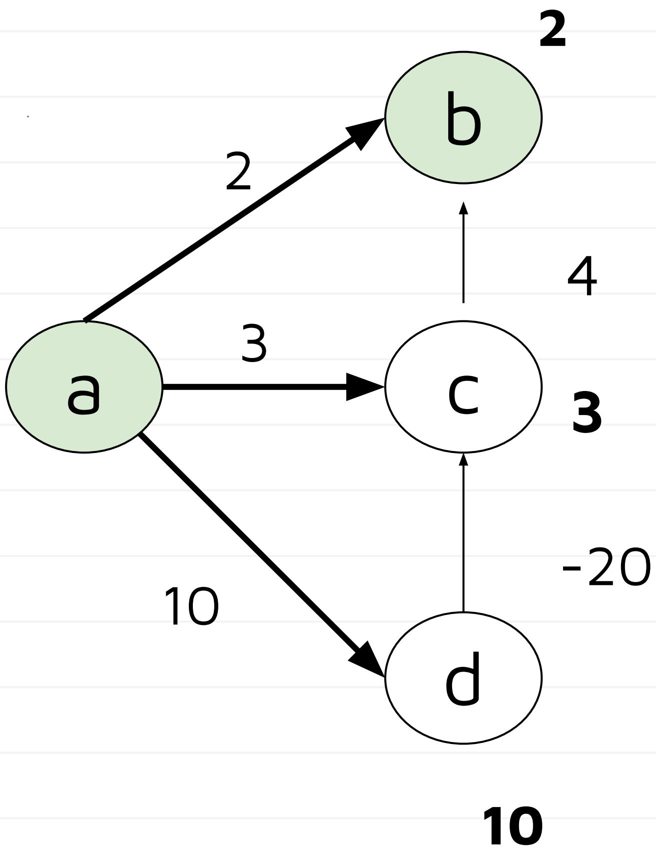
Example



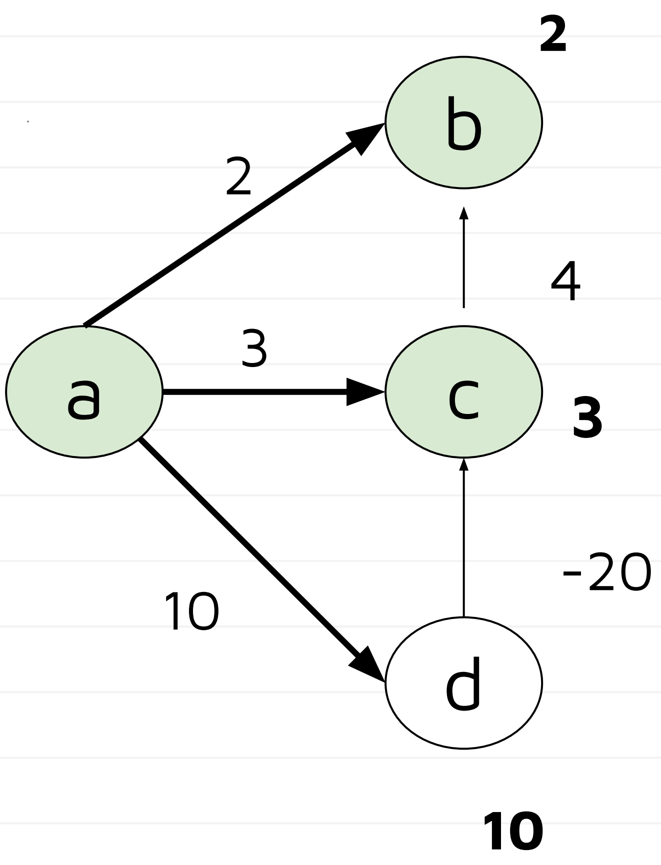
Example



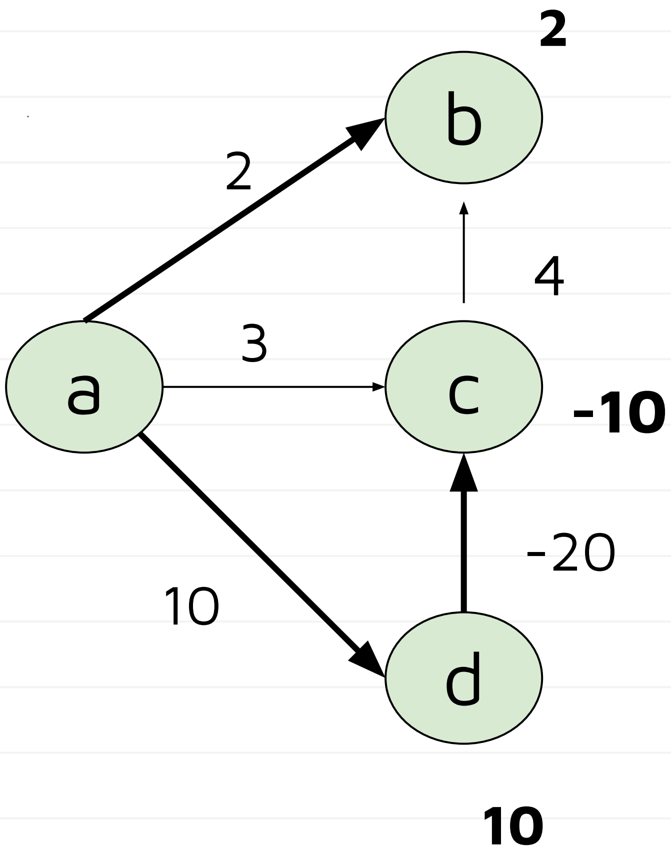
Example



Example



Example



Implementation

Outline of Dijkstra's Algorithm

```python
def dijkstra(graph, weights, source):
    est = {node: float('inf') for node in graph.nodes}
    est[source] = 0
    pred = {node: None for node in graph.nodes}

    # empty set
    C = set()

    # while there are nodes still outside of C
        # find node u outside of C with smallest
        # estimated distance
    C.add(u)

    for v in graph.neighbors(u):
        update(u, v, weights, est, pred)

    return est, pred
```

Dijkstra's Algorithm: Naïve Implementation

```
def dijkstra(graph, weights, source):
    est = {node: float('inf') for node in graph.nodes}
    est[source] = 0
    pred = {node: None for node in graph.nodes}

    outside = set(graph.nodes)

    while outside:
        # find smallest with linear search
        u = min(outside,  key = est.get)
        outside.remove(u)
        for v in graph.neighbors(u):
            update(u, v, weights, est, pred)

    return est, pred
```

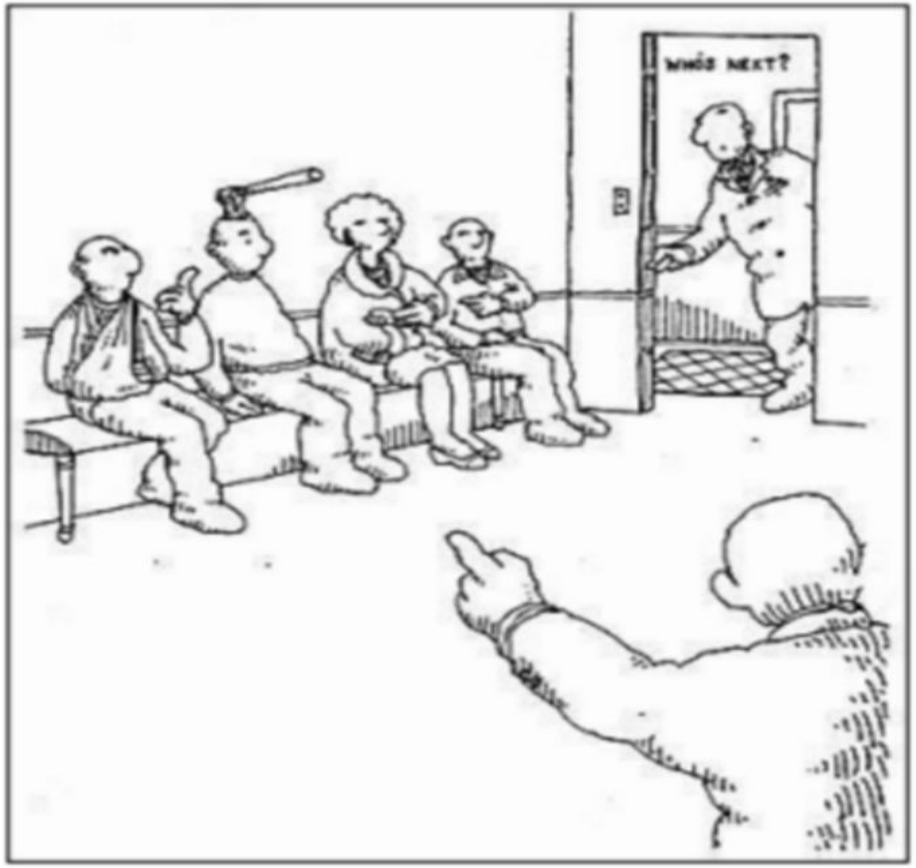
Dijkstra's Algorithm: Naïve Implementation

```
def dijkstra(graph, weights, source):
    est = {node: float('inf') for node in graph.nodes}
    est[source] = 0
    pred = {node: None for node in graph.nodes}

    outside = set(graph.nodes)

    while outside:
        # find smallest with linear search
        u = min(outside, key = est.get)
        outside.remove(u)
        for v in graph.neighbors(u):
            update(u, v, weights, est, pred)

    return est, pred
```

inefficient

Priority Queue ADT

- Emergency Department waiting room operates as a priority queue
- Patients sorted according to seriousness, NOT how long they have waited



Priority Queues

- A **priority queue** allows us to store (key, value) pairs, efficiently return key with lowest value.
- Suppose we have a priority queue class:
  - `PriorityQueue(priorities)` will create a priority queue from a dictionary whose values are priorities.
  - The `.extract_min()` method removes and returns key with smallest value.
  - The `.change_priority(key, value)` method changes key's value.

Example

```
>>> pq = PriorityQueue({
          'w': 5,
          'x': 4,
          'y': 1,
          'z': 3
})
>>> pq.extract_min()
'Y'
>>> pq.change_priority('w', 2)
>>> pq.extract_min()
?
```

Example

```
>>> pq = PriorityQueue({
            'w': 5,
            'x': 4,
            'y': 1,
            'z': 3
})
>>> pq.extract_min()
'Y'
>>> pq.change_priority('w', 2)
>>> pq.extract_min()
'w'
```

Dijkstra's Algorithm: Priority Queue

```python
def dijkstra(graph, weights, source):
    est = {node: float('inf') for node in graph.nodes}
    est[source] = 0
    pred = {node: None for node in graph.nodes}

    priority_queue = PriorityQueue(est)
    while priority_queue:
        u = priority_queue.extract_min()
        for v in graph.neighbors(u):
            changed = update(u, v, weights, est, pred)
            if changed:
                priority_queue.change_priority(v, est[v])

    return est, pred
```

Heaps

- A priority queue can be implemented using a **heap**.
- If a **binary min-heap** is used:
  - `PriorityQueue(est)` takes $\Theta(V)$ time.
  - `.extract_min()` takes $O(\log V)$ time.
  - `.change_priority()` takes $O(\log V)$ time.

Time Complexity Using Min Heap

```
def dijkstra(graph, weights, source):
    est = {node: float('inf') for node in graph.nodes}
    est[source] = 0
    pred = {node: None for node in graph.nodes}

    priority_queue = PriorityQueue(est)
    while priority_queue:
        u = priority_queue.extract_min()
        for v in graph.neighbors(u):
            changed = update(u, v, weights, est, pred)
            if changed:
                priority_queue.change_priority(v, est[v])

    return est, pred
```

$\Theta(V)$

Time Complexity Using Min Heap

```
def dijkstra(graph, weights, source):
    est = {node: float('inf') for node in graph.nodes}
    est[source] = 0
    pred = {node: None for node in graph.nodes}

    priority_queue = PriorityQueue(est)
    while priority_queue:
        u = priority_queue.extract_min()
        for v in graph.neighbors(u):
            changed = update(u, v, weights, est, pred)
            if changed:
                priority_queue.change_priority(v, est[v])

    return est, pred
```

$\Theta(V)$

$\Theta(1)$

Time Complexity Using Min Heap

```
def dijkstra(graph, weights, source):
    est = {node: float('inf') for node in graph.nodes}
    est[source] = 0
    pred = {node: None for node in graph.nodes}

    priority_queue = PriorityQueue(est)
    while priority_queue:
        u = priority_queue.extract_min()
        for v in graph.neighbors(u):
            changed = update(u, v, weights, est, pred)
            if changed:
                priority_queue.change_priority(v, est[v])

    return est, pred
```

$\Theta(V)$

O(log v)

$\Theta(1)$

Time Complexity Using Min Heap

```
def dijkstra(graph, weights, source):
    est = {node: float('inf') for node in graph.nodes}
    est[source] = 0
    pred = {node: None for node in graph.nodes}

    priority_queue = PriorityQueue(est)
    while priority_queue:
        u = priority_queue.extract_min()
        for v in graph.neighbors(u):
            changed = update(u, v, weights, est, pred)
            if changed:
                priority_queue.change_priority(v, est[v])

    return est, pred
```

$\Theta(V)$

O(log v)

$\Theta(1)$

O(log v)

Time Complexity Using Min Heap

```python
def dijkstra(graph, weights, source):
    est = {node: float('inf') for node in graph.nodes}
    est[source] = 0
    pred = {node: None for node in graph.nodes}          Θ(V )

    priority_queue = PriorityQueue(est)
    while priority_queue:
        u = priority_queue.extract_min()                 O(log v )
        for v in graph.neighbors(u):
            changed = update(u, v, weights, est, pred)   Θ(1 )
            if changed:
                priority_queue.change_priority(v, est[v])  O(log v )

    return est, pred
```

Time Complexity:

Time Complexity Using Min Heap

```
def dijkstra(graph, weights, source):
    est = {node: float('inf') for node in graph.nodes}
    est[source] = 0
    pred = {node: None for node in graph.nodes}

    priority_queue = PriorityQueue(est)
    while priority_queue:
        u = priority_queue.extract_min()
        for v in graph.neighbors(u):
            changed = update(u, v, weights, est, pred)
            if changed:
                priority_queue.change_priority(v, est[v])

    return est, pred
```

$\Theta(V)$

V

O(log v)

Θ(1)

O(log v)

Time Complexity: O(V log V +)

Time Complexity Using Min Heap

```
def dijkstra(graph, weights, source):
    est = {node: float('inf') for node in graph.nodes}
    est[source] = 0
    pred = {node: None for node in graph.nodes}

    priority_queue = PriorityQueue(est)
    while priority_queue:
        u = priority_queue.extract_min()
        for v in graph.neighbors(u):
            changed = update(u, v, weights, est, pred)
            if changed:
                priority_queue.change_priority(v, est[v])

    return est, pred
```

$\Theta(V)$

O(log v)

$\Theta(1)$

O(log v)

Time Complexity: O(V log V + E Log V)

Loop Invariant

- Assume that edge weights are **positive**.
- Before each iteration of Dijkstra's algorithm, both the *distance estimate* and the *predecessor* of the **first** node in the priority queue are **correct**.

Exercise

True or **False**: in Dijkstra's algorithm, a node's predecessor can be changed *after* it is *first* set.

A: True

B: False

C: Not sure yet

Exercise

True or **False**: in Dijkstra's algorithm, a node's predecessor can be changed *after* it is *popped* from the **priority queue**.

A: True

B: False

C: Not sure yet

Thank you!

Do you have any questions?

CampusWire!