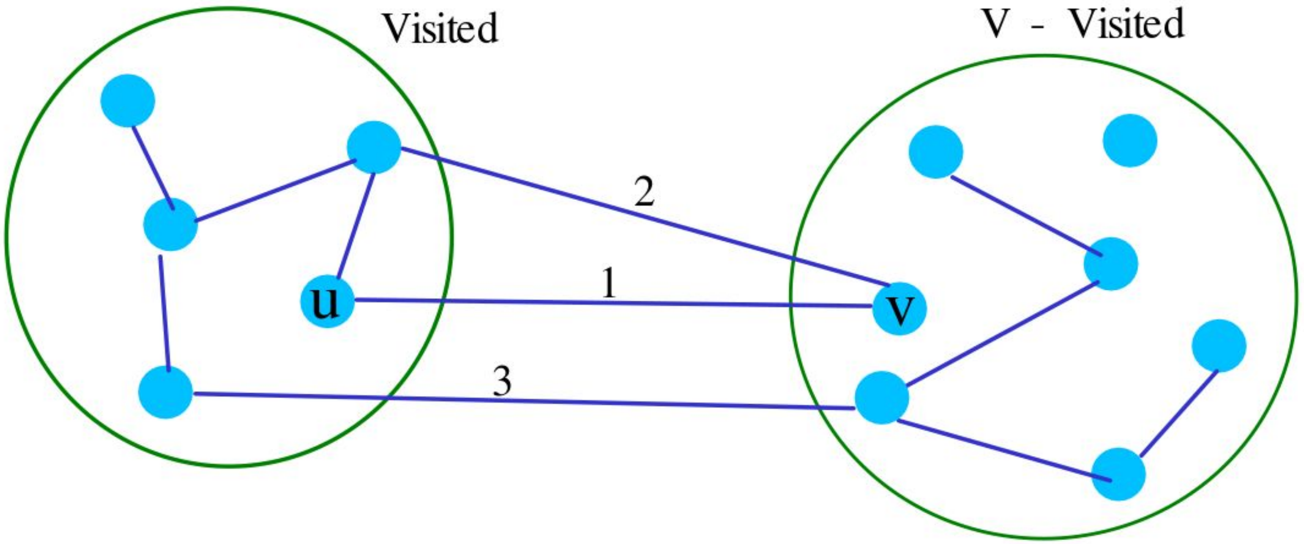
# Graph Algorithms

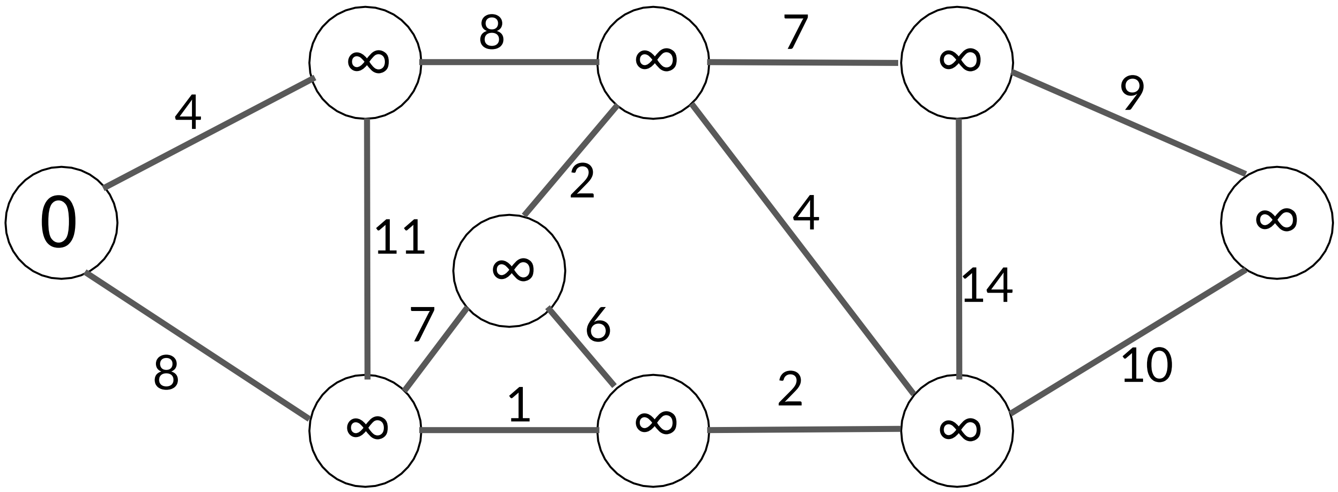
# Prim’s Algorithm

# Prim’s Algorithm

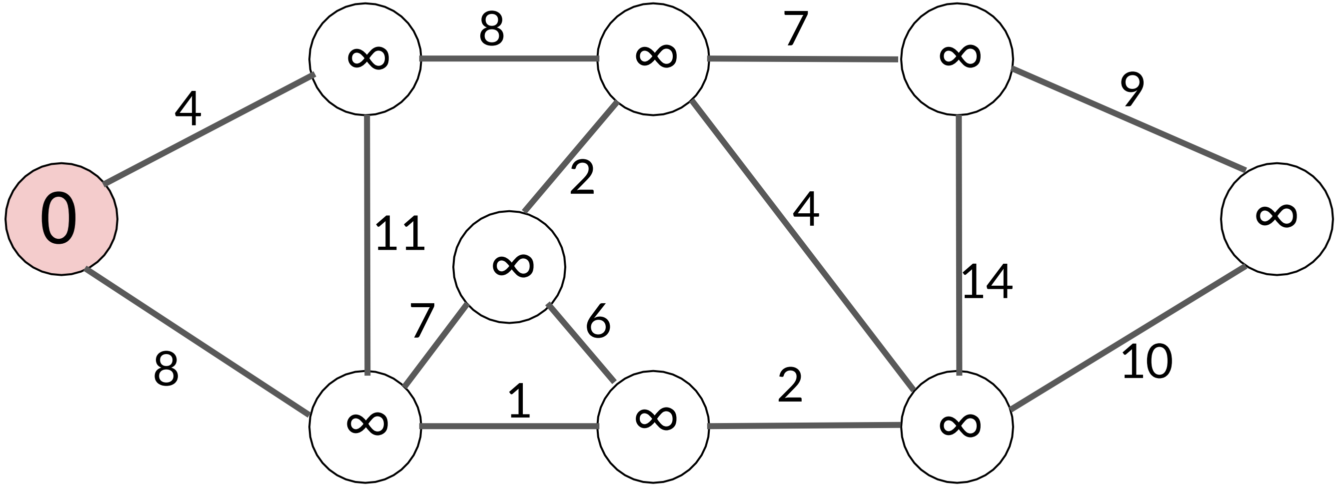
Idea: Grow one connected component in a greedy fashion (i.e., by adding a vertex **v ∈ V − Visited** that is one end of a minimum weight edge leaving Visited).



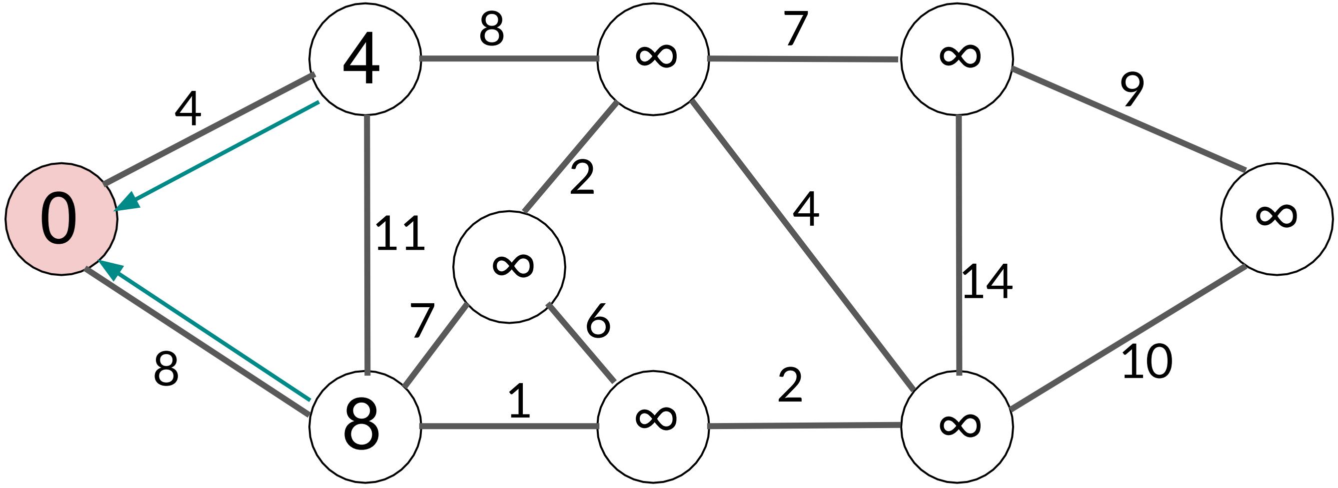
# Prim's algorithm



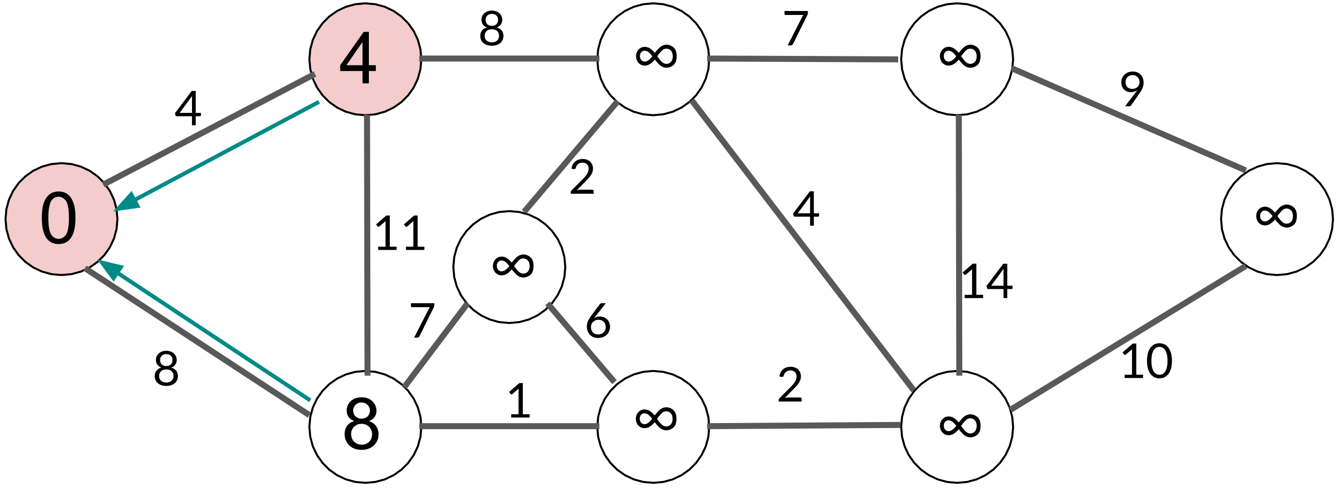
# Prim's algorithm



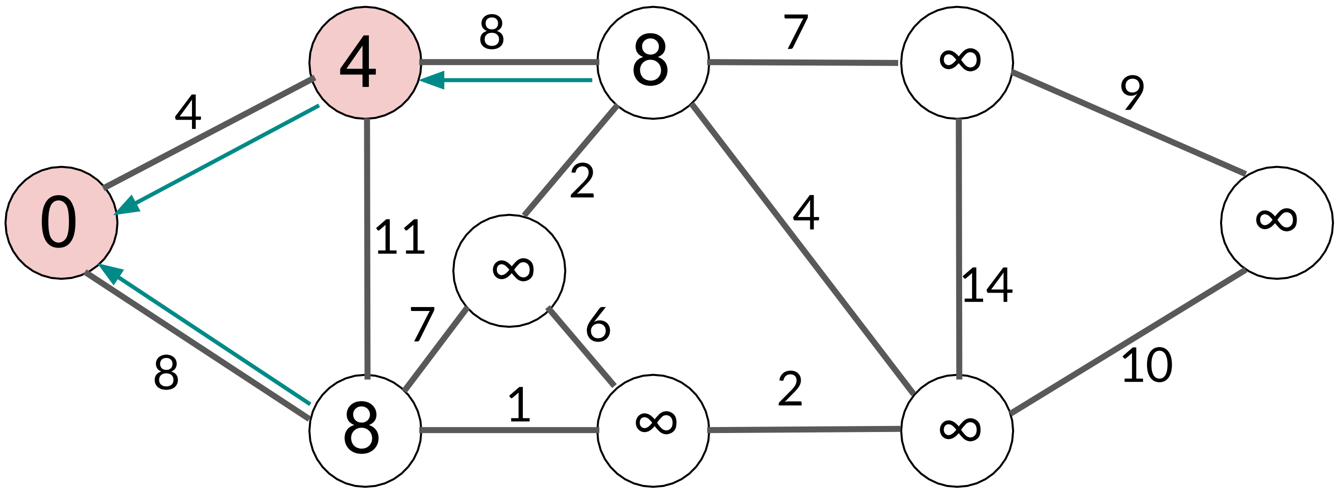
# Prim’s algorithm



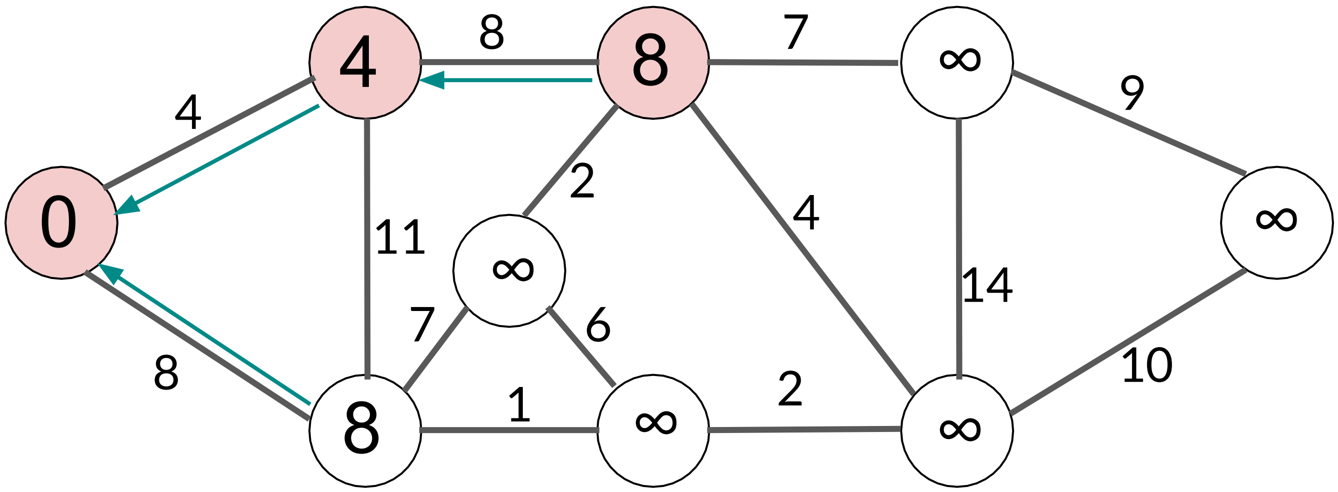
# Prim's algorithm



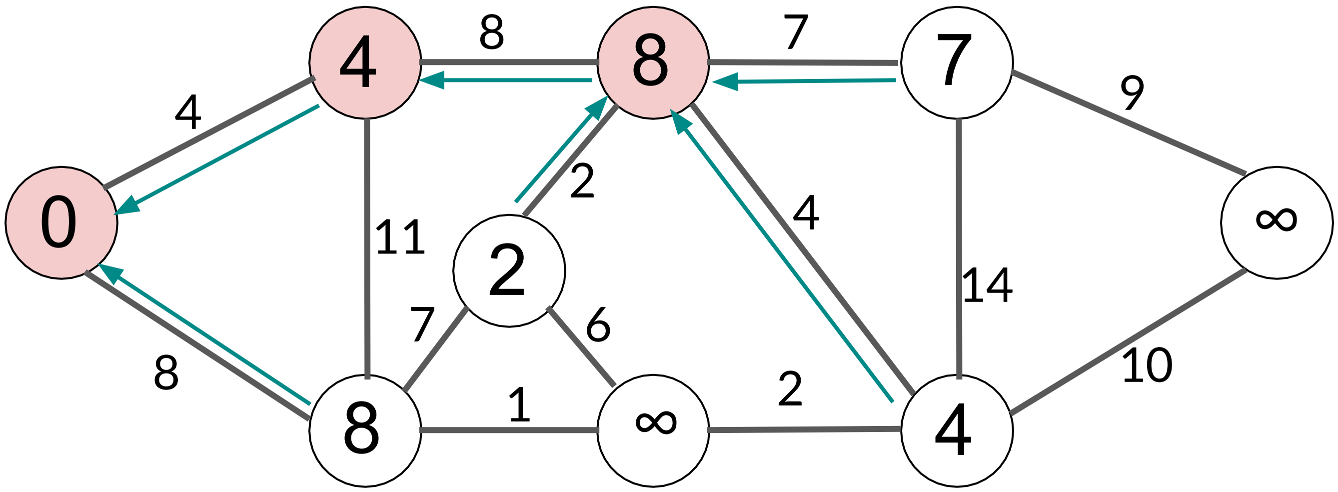
# Prim's algorithm



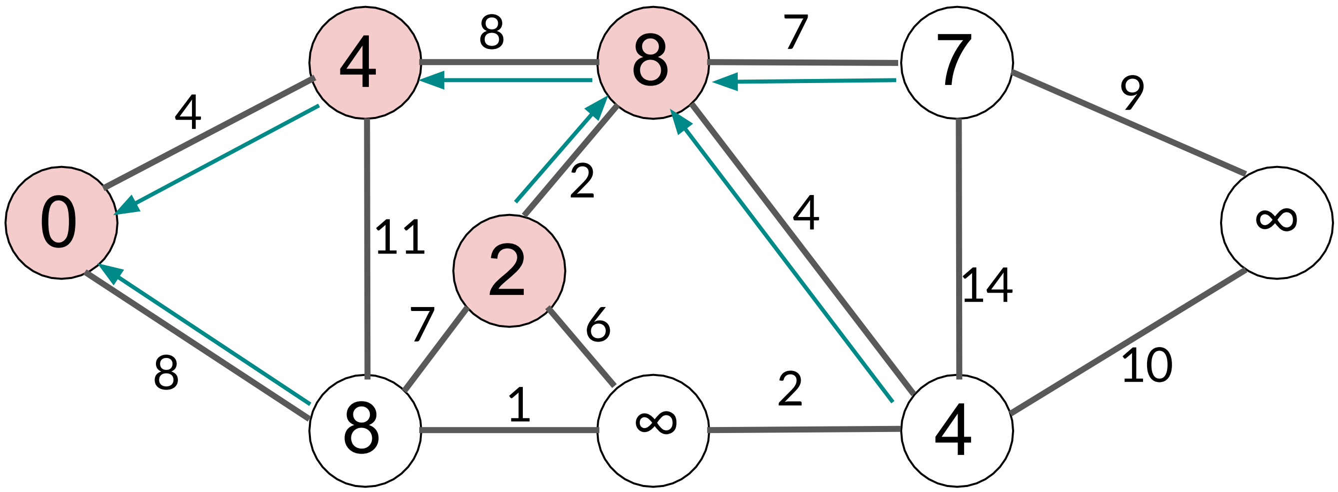
# Prim's algorithm



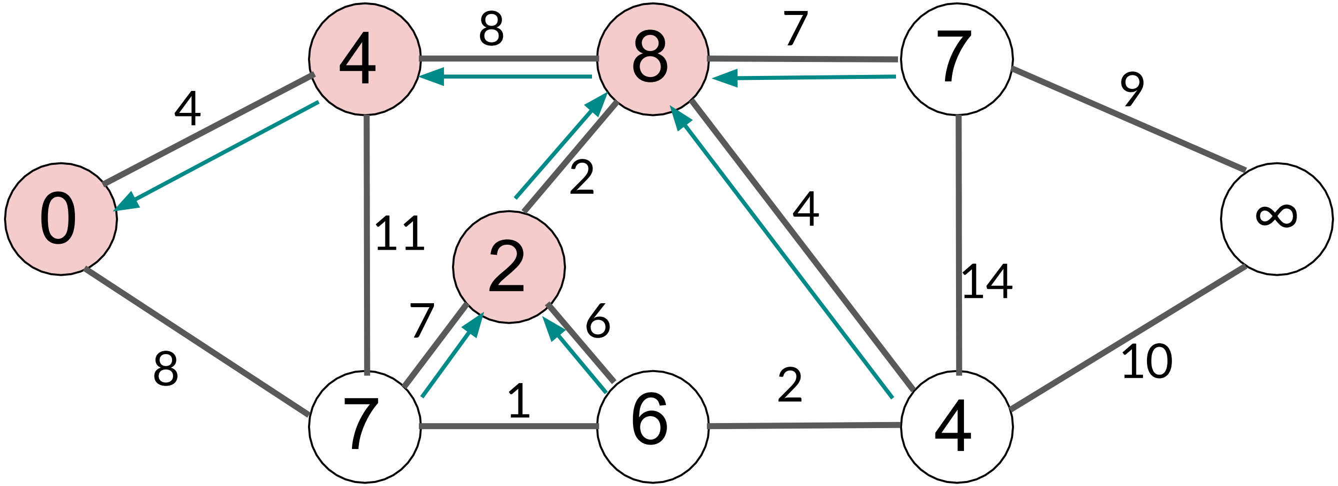
# Prim's algorithm



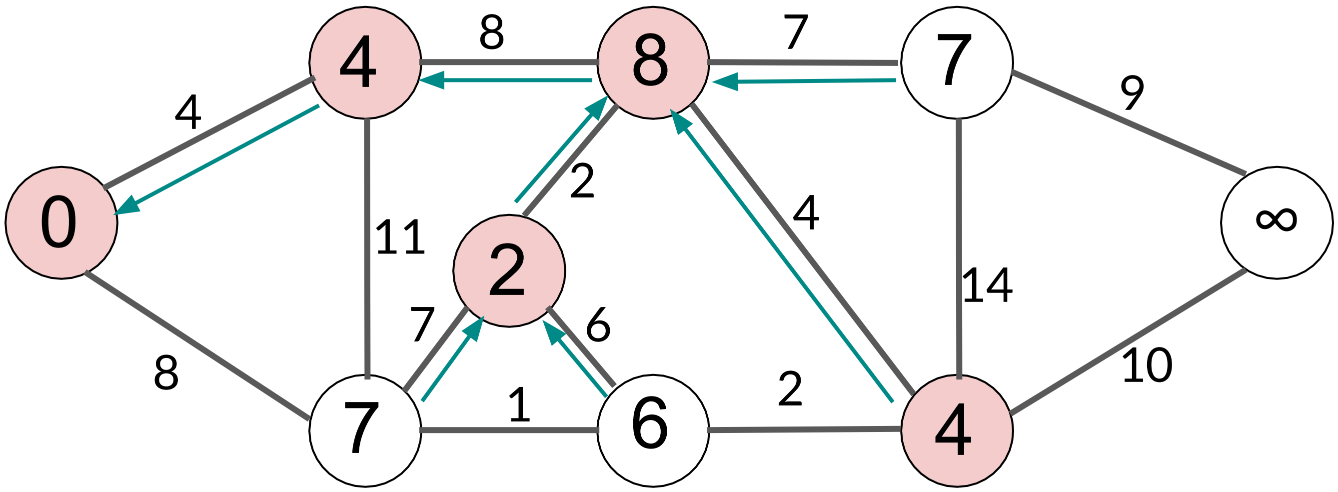
# Prim’s algorithm



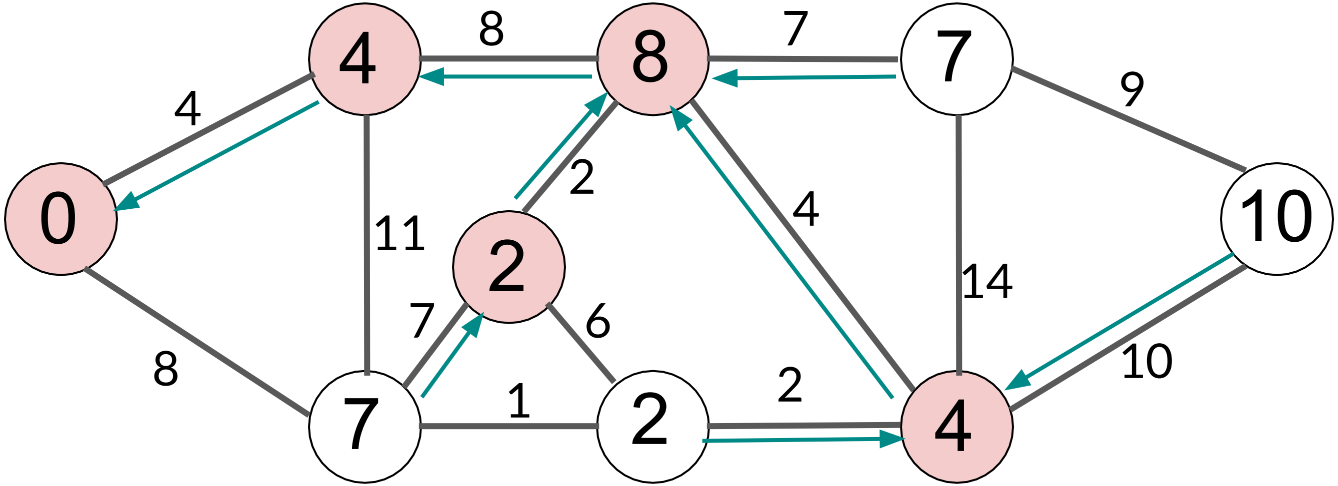
# Prim's algorithm



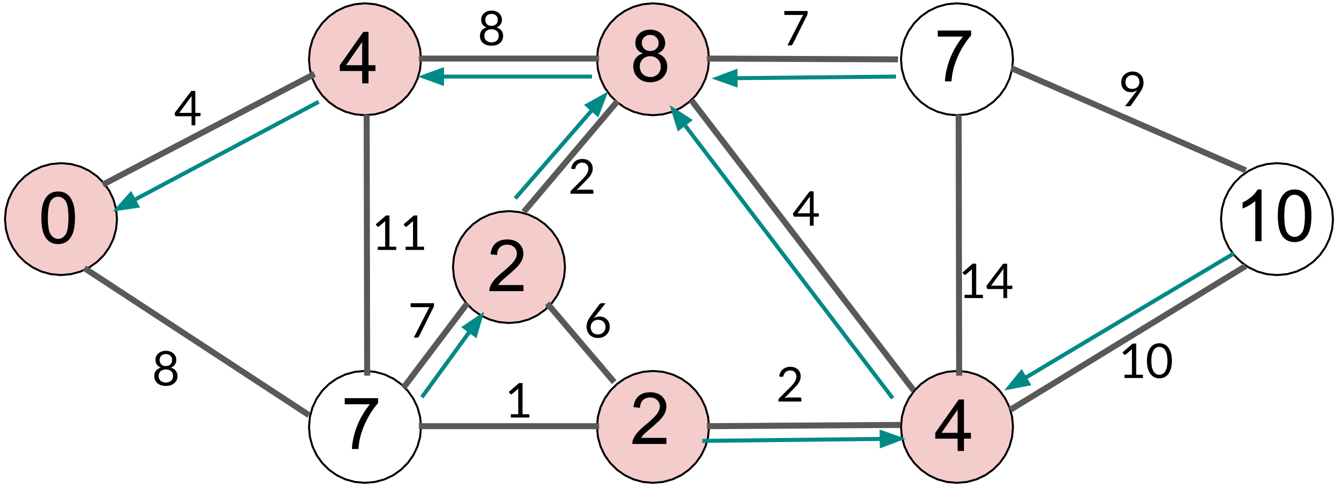
# Prim’s algorithm



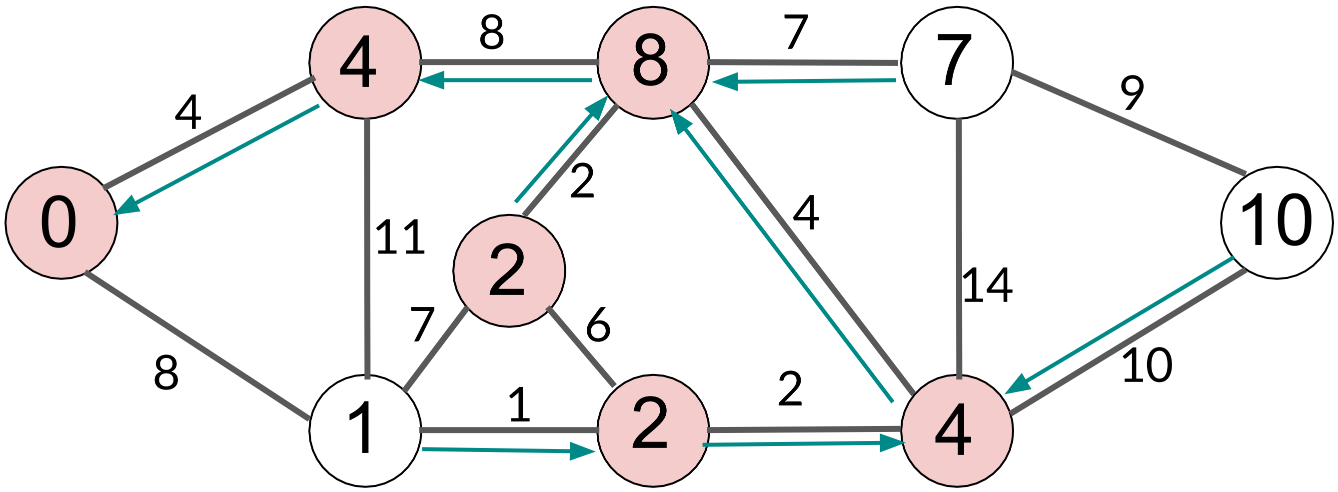
# Prim's algorithm



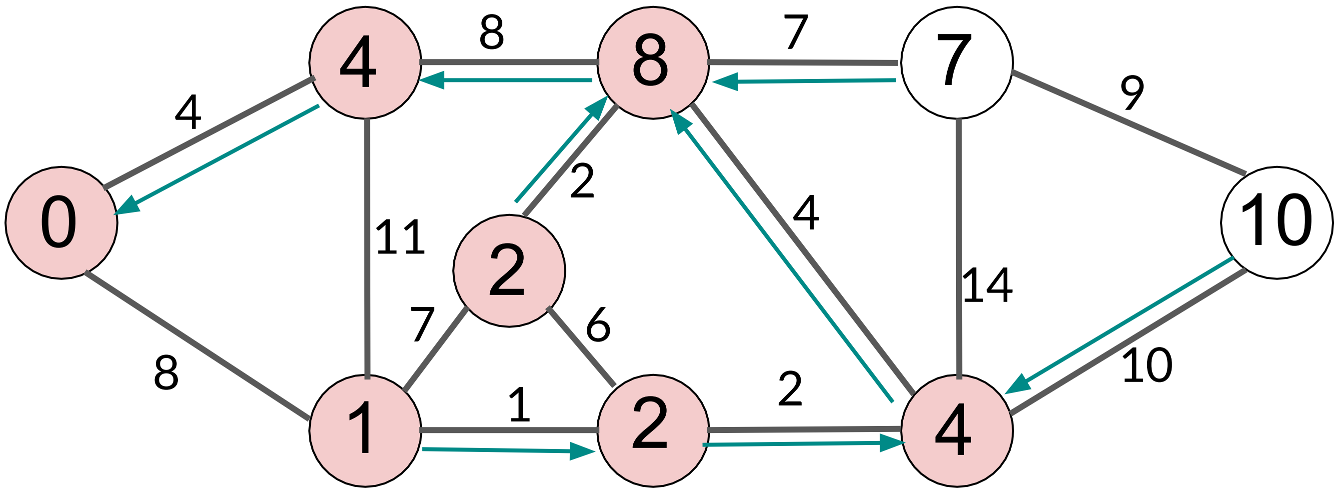
# Prim's algorithm



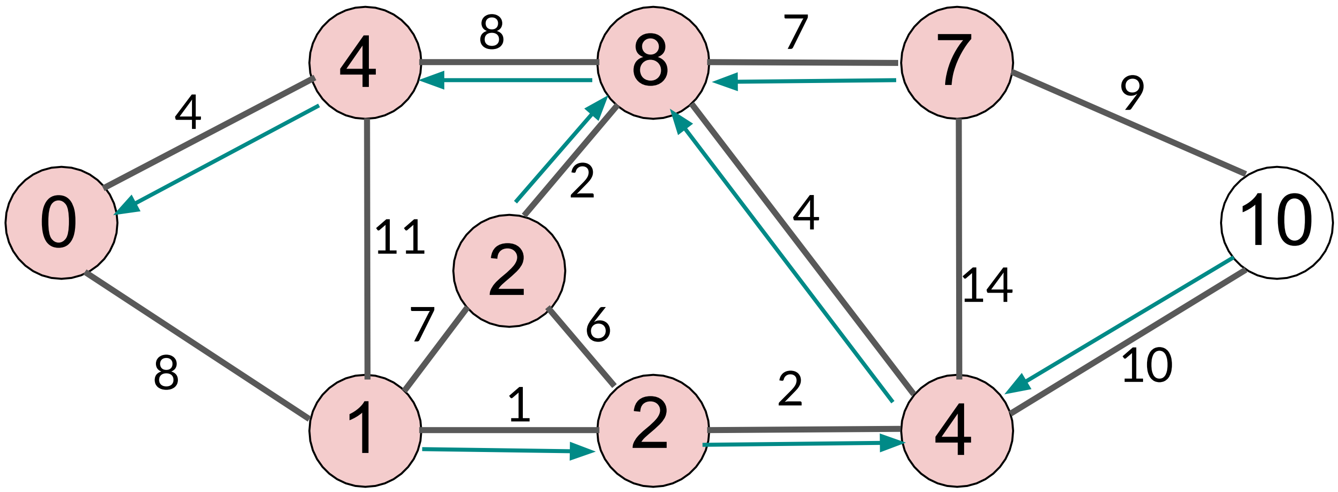
# Prim’s algorithm



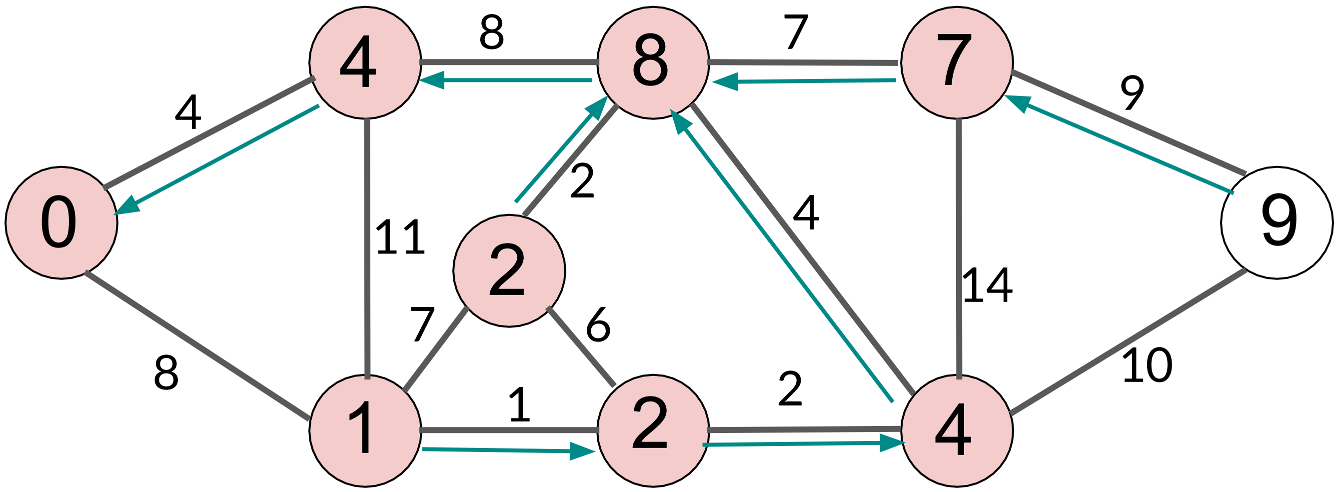
# Prim's algorithm



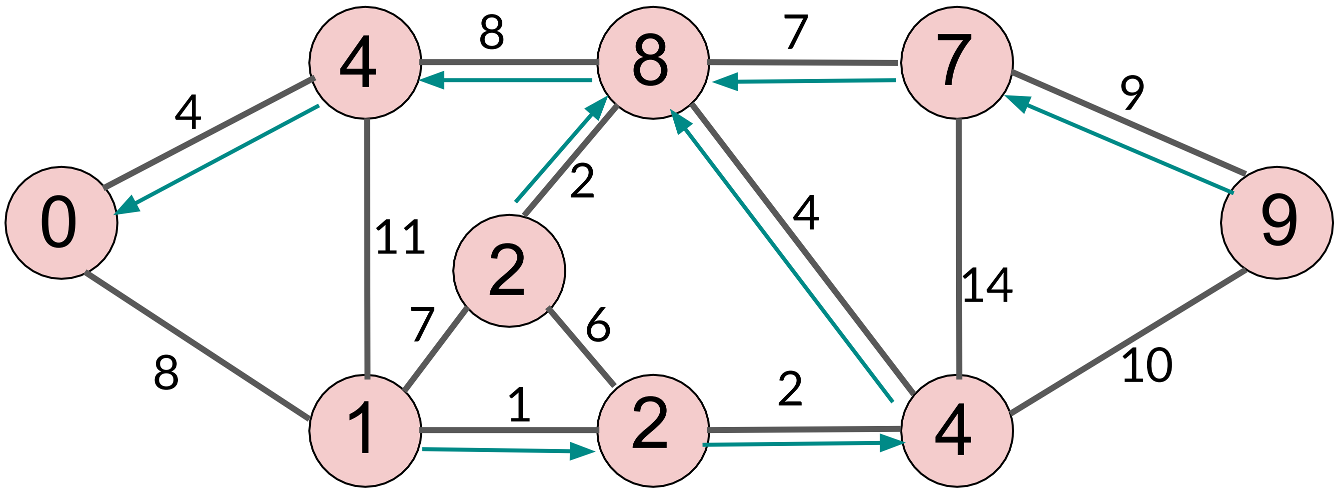
# Prim's algorithm



# Prim's algorithm



# Prim’s algorithm



# Prim’s Algorithm: Correctness: Unique edge weights

- **T**: MST found by Prim’s Algorithm
- **M**: optimal MST

**Proof by contradiction.** Assume T ≠ M →T-M ≠ Ø → Let (u, v) be any edge in T - M.

- When (u,v) was added, it was the least-cost edge crossing the cut (Visited, V-Visited)
  - (u, v) crosses the cut, since u and v were not connected when Priml's algorithm selected (u, v)
  - Prim’s algorithm select the least-cost edge crossing the cut
- **M** is a MST → There must be a path from u to v in **M**. This path begins in visited and ends in V-Visited. → There must be an edge along that path where x ∈ Visited and y ∈ V-Visited. Since (u,v) is the least-code edge crossing (Visited, V-Visited) → **w(u,v) < w(x,y)**
- M’ = M-{(x,y)} U {(u,v)}. M’ is a spanning tree because it connects all vertices. Since (x,y) is on the cycle formed by adding (u, v)
- w(M’) = w(M) - w(x,y) + w(u,v) < w(M) → M’ is a MST → contradiction M was the optimal solution

# Prim’s Algorithm: Correctness: Not unique edge weights

- **T**: MST found by Prim’s Algorithm
- **M**: optimal MST

**Proof.** We will prove w(T) = w(M). If T = M, we are done. Otherwise T ≠ M, so T–M ≠ Ø. Let (u,v) be any edge in T-M.

- When (u,v) was added, it was the least-cost edge crossing the cut (Visited, V-Visited)
  - (u, v) crosses the cut, since u and v were not connected when Prim’'s algorithm selected (u, v)
  - Prim’s algorithm select the least-cost edge crossing the cut
- **M** is a MST → There must be a path from u to v in **M**. This path begins in Visited and ends in V-Visited. → There must be an edge along that path where x in Visited and y in V-Visited. Since (u,v) is the least-code edge crossing (Visited, V-Visited) → **w(u,v) ≤ w(x,y)**
- M’ = M-{(x,y)} U {(u,v)}. M’ is a spanning tree because it connects all vertices. Since (x,y) is on the cycle formed by adding (u, v)
- w(M’) = w(M) - w(x,y) + w(u,y) → w(M’) **≤** w(M)
- M’ is a MST → w(M) ≤ w(M’) → w(M’) = w(M)
- Note that |T – M’| = |T – M| – 1. Therefore, if we repeat this process once for each edge in T – M, we will have converted M into T while preserving w(M). Thus w(T) = w(M).

# Prim’s Algorithm

```
Prim-simple(G, s)

    T = ∅
    Visited = {s}
    while Visited ≠ V
         find vertex v ∉ Visited such that
         there exists a u ∈ visited and
         (u,v) is a minimum weight edge leaving Visited

         T = T ∪ {(u, v)}
         Visited = Visited ∪ {v}
    return T
```

**Greedy choice:** at each step it adds to the tree an edge that contributes the minimum amount possible to the tree’s weight

Choose vertex v ∈ V − visited connected to a minimum weight edge e = (u, v) between Visited and V − Visited

# Prim's Algorithm Runtime: $O(V^2)$

```
Prim-simple(G, s)

    T = ∅
    Visited = {s}
    while Visited ≠ V
        find vertex v ∉ Visited such that
        there exists a u ∈ visited and
        (u,v) is a minimum weight edge leaving Visited

        T = T ∪ {(u, v)}
        Visited = Visited ∪ {v}
    return T
```

O(V)

O(V)

# Prim's Algorithm: better implementation

- **Idea**: Maintain V – Visited as a priority queue Q.
- For v ∈ V- Visited, we define:

$$\text{weight}(\text{v}) = \begin{cases} \infty \\ \min \text{w}(\text{e}) \mid \text{e} = (\text{u}, \text{v}) \in \text{E and } u \in T \end{cases}$$

- The weight of each vertex in V-Visited is the weight of the least-weight edge connecting it to a vertex in Visited.
- Priority Queue implemented using heap data structure
  - V - Visited is maintained as an array in heap order, and the key of each vertex is its weight defined above
  - ExtractMin(): remove and return vertex with minimum weight
  - Insert(v, weight(v)): insert vertex v with weight(v)
  - DeleteMin(v): delete the vertex with minimum weight
  - decrease-key(v, oldWeight, newWeight)
    - deletes vertex V with oldWeight and inserts vertex V with newWeight
  - The runtime of all operations are O(log k) where k is the size of heap

# Prim’s Algorithm: better implementation

**Idea**: Maintain V – Visited as a priority queue Q. The key of each vertex in Q is the weight of the least-weight edge connecting it to a vertex in T.

```
Prim(G, s)
    T = ∅
    Q = V
    Key[s] = 0
    Key[u] = ∞ for all u ∈ V
    while Q ≠ ∅
        u = EXTRACT-MIN(Q)
        for each v ∈ Adj[u]
            if v ∈ Q and w(u,v) < key[v]
                Key[v] = w(u, v)
                pred[v] = u      # T = T ∪ {(u, v)}
    return T
```

At the end, {(v, pred[v])} forms the MST.

# Prim's Algorithm: Runtime

**Idea**: Maintain V – T as a priority queue Q. The key of each vertex in Q is the weight of the least-weight edge connecting it to a vertex in T.

$\Theta(V)$

$|V|$

$degree(u)$

```
Prim(G, s)
    T = ∅
    Q = V
    Key[s] = 0
    Key[u] = ∞ for all u ∈ V
    while Q ≠ ∅
        u = EXTRACT-MIN(Q)
        for each v ∈ Adj[u]
            if v ∈ Q and w(u,v) < key[v]
                Key[v] = w(u, v)
                pred[v] = u        # T = T ∪ {(u, v)}
    return T
```

Decrease-key

Runtime = $\Theta(V)\cdot(T_{\text{EXTRACT-MIN}}) + \Theta(E)\cdot(T_{\text{DECREASE-KEY}})$

# Prim's Algorithm: Runtime

Runtime = $\Theta(V)\cdot(T_{\text{EXTRACT-MIN}}) + \Theta(E)\cdot(T_{\text{DECREASE-KEY}})$

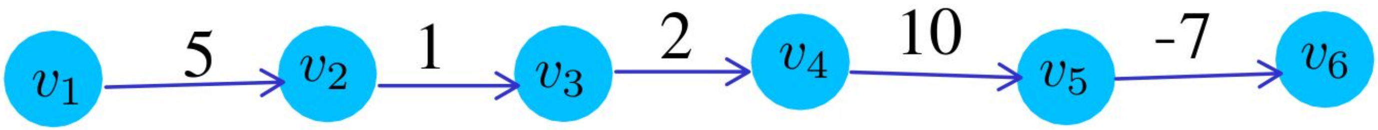
| Q | $T_{\text{EXTRACT-MIN}}$ | $T_{\text{DECREASE-KEY}}$ | Total Time |
|---|---|---|---|
| array | $O(V)$ | $O(1)$ | $O(V^2)$ |
| Binary heap | $O(\lg V)$ | $O(\lg V)$ | $O(E \lg V)$ |
| Fibonacci heap | $O(\lg V)$ amortized | $O(1)$ amortized | $O(E + V \lg V)$ amortized |

# Shortest Path

# Shortest path

- Consider a digraph $G = (V, E)$ with edge-weight function $w : E \to R$. The weight of path $P = (v_1, v_2, \ldots, v_k)$ is defined to be

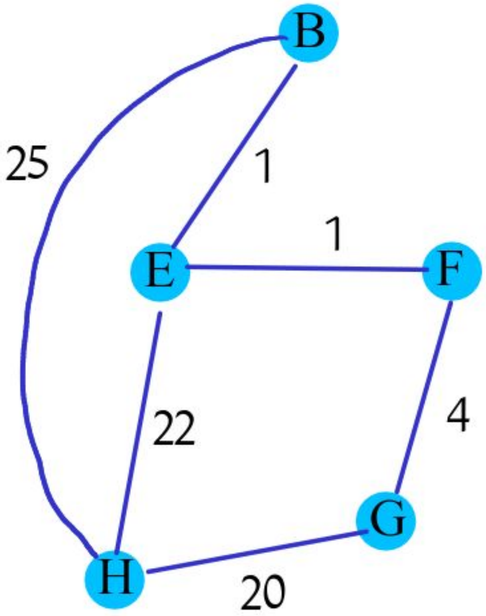
$$w(P) = \sum_{i=1}^{k-1} w(v_i, v_{i+1})$$



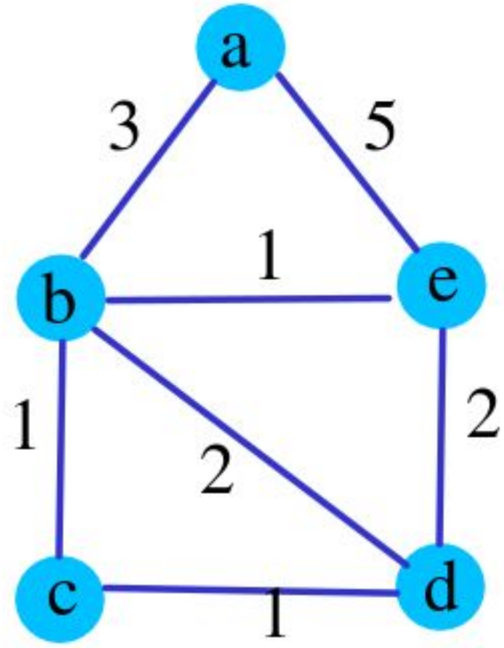

- A **shortest path** from u to v is a path of minimum weight from u to v.
- Shortest path from u to v **= δ(u, v)** = min { w(P): P is a path from u to v}
- δ(u, v) = ∞ if no path from u to v exists.

# Why BFS is not enough for finding the shortest path?

What is the shortest path from B to G?

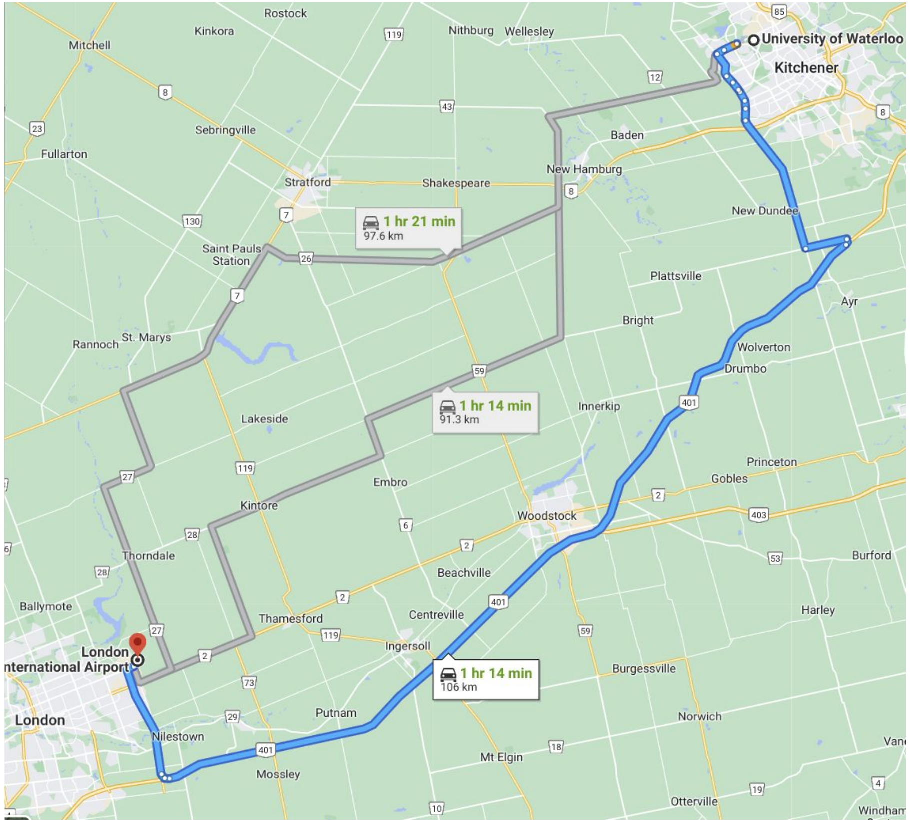


# Why MST algorithms are not enough?



# Finding Shortest paths in graphs

- **Input**: a graph (directed/undirected) G=(V, E) with **non-negative** edge weights( $w(e) \geq 0$), and a starting node **s**
- **Output:** A shortest path from s to each vertex in the graph
- Single-Source-Shortest-Path problem
- The length of the shortest path and then find the shortest path



# Applications of Shortest Path

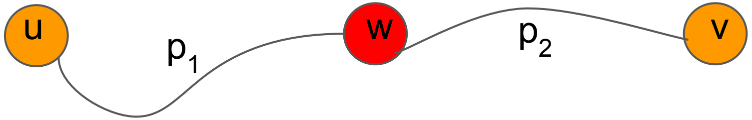
Map routing

Robot navigation

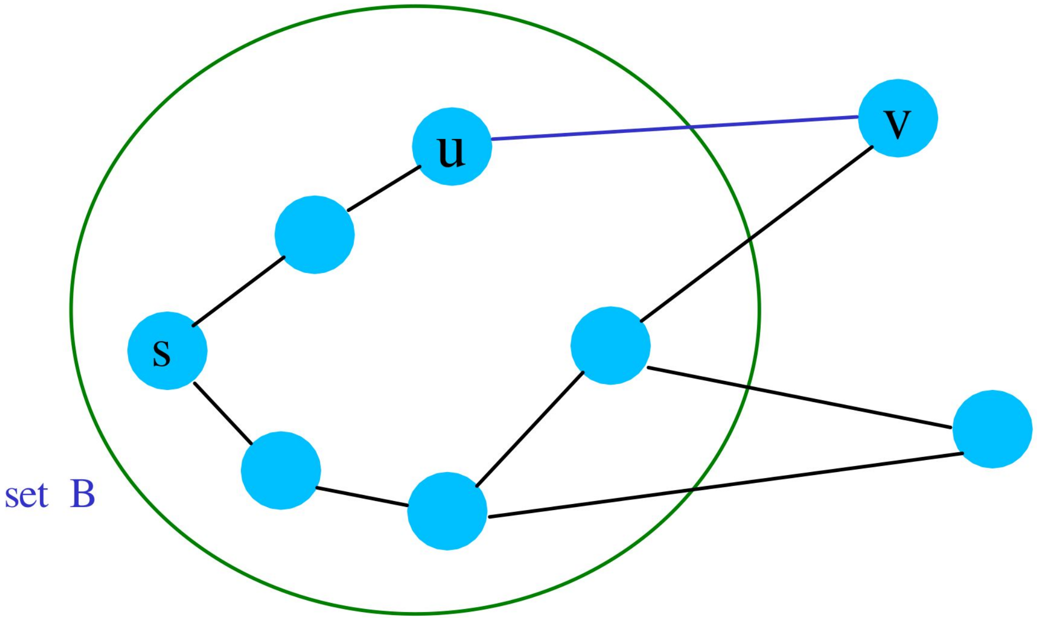
Network routing protocols (OSPF, BGP, RIP)

# Shortest path: Optimal Substructure property

- **Optimal Substructure property:**
  - **Optimal solution to the problem contains optimal solution to the subproblems**
- **Example: Shortest path in graphs**
  - **P**: the shortest path between u and v.
  - **Claim**: $p_1$ is a shortest path from u to w
    - If there were another path, say $p'_1$ from u to w with less weight, we could cut out $p_1$ and paste in $p'_1$ to produce a path $p' = p'_1 + p_2$ with fewer edges → contradiction: $p_1$ is an optimal solution or the shortest path
    - Similarly we can show $p_2$ is the shortest path from w to v



# Shortest path: Greedy Algorithm: Dijkstra’s Algorithm: Idea




Greedy choice: add the vertex with the minimum distance from s

# Shortest path: Greedy Algorithm: Dijkstra's Algorithm: Idea

dist[v] = $\infty$ for all v $\in$ V

dist[s] = 0

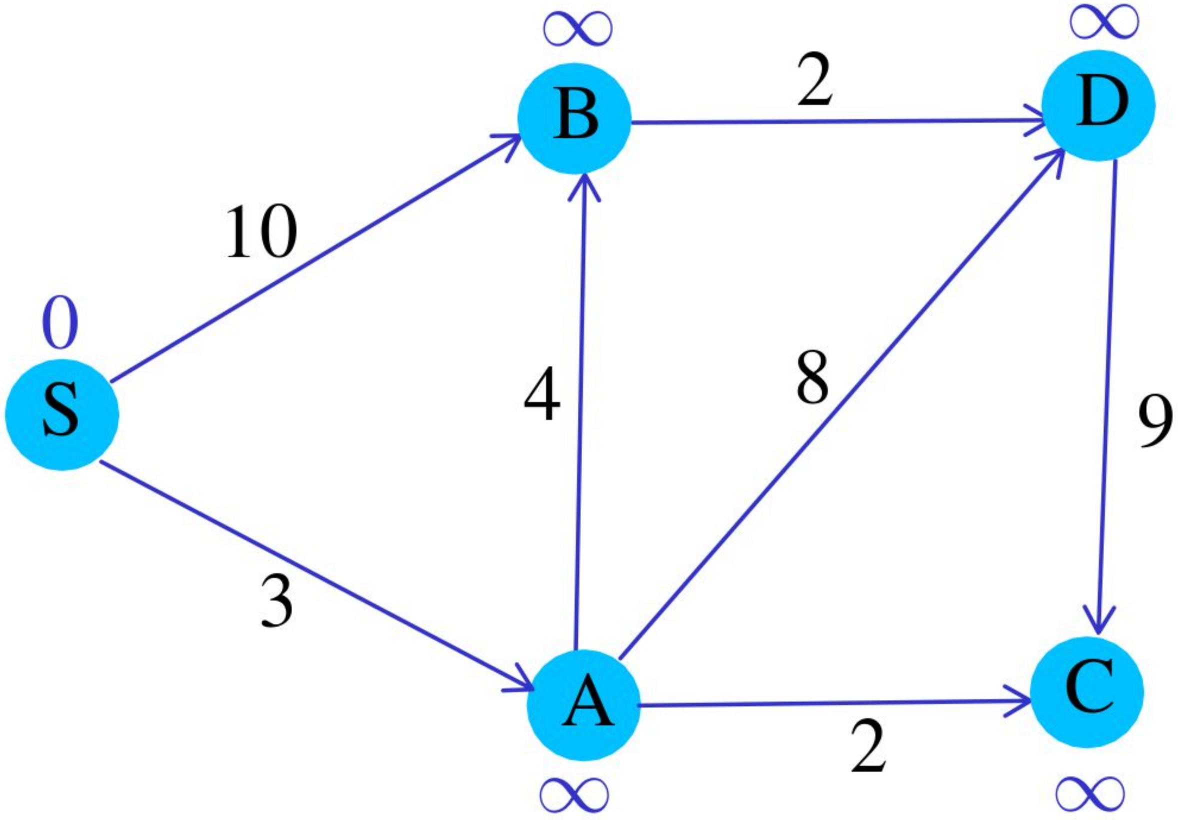
B = $\varnothing$ # B is a set of vertices with known shortest distance to s

While B $\neq$ V

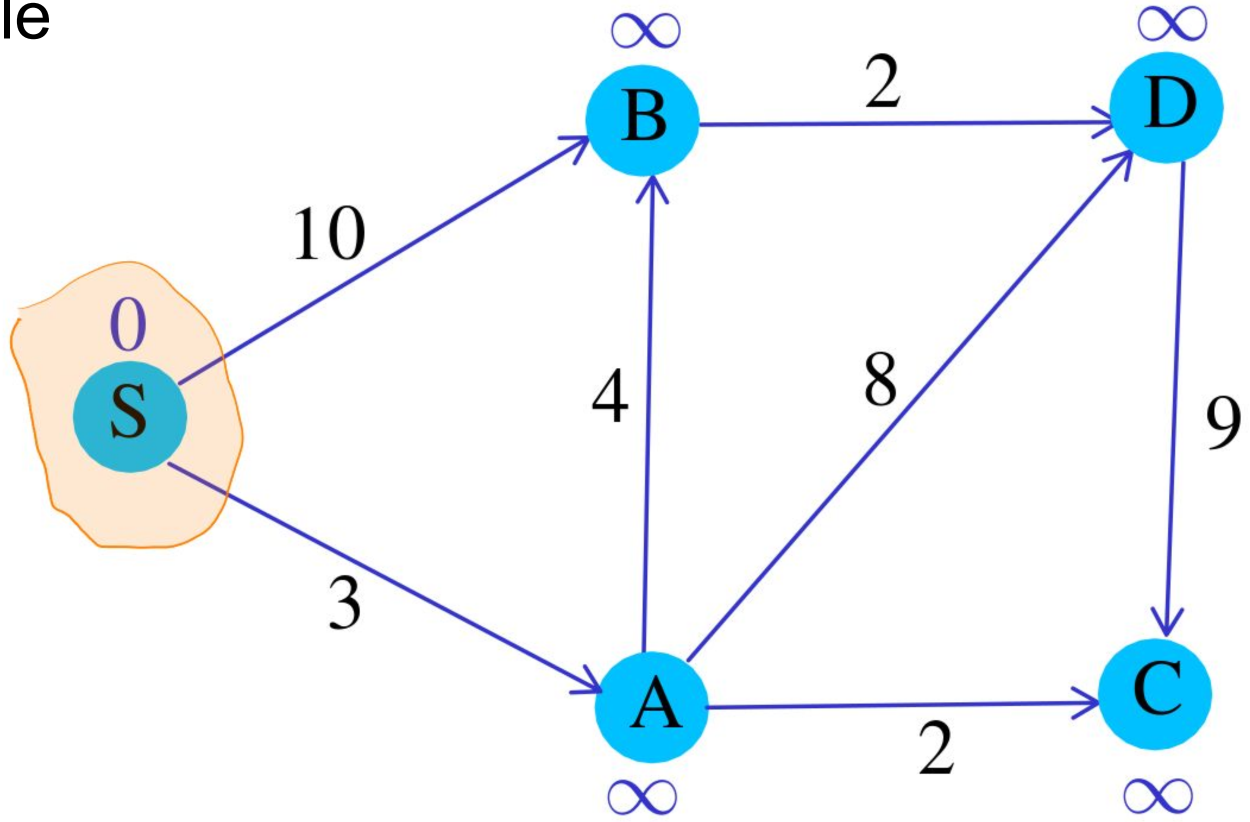
Choose edge (u, v), u $\in$ B, v $\notin$ B to minimize d(s, u) + w(u, v)

Update d[v]: distance of S to v

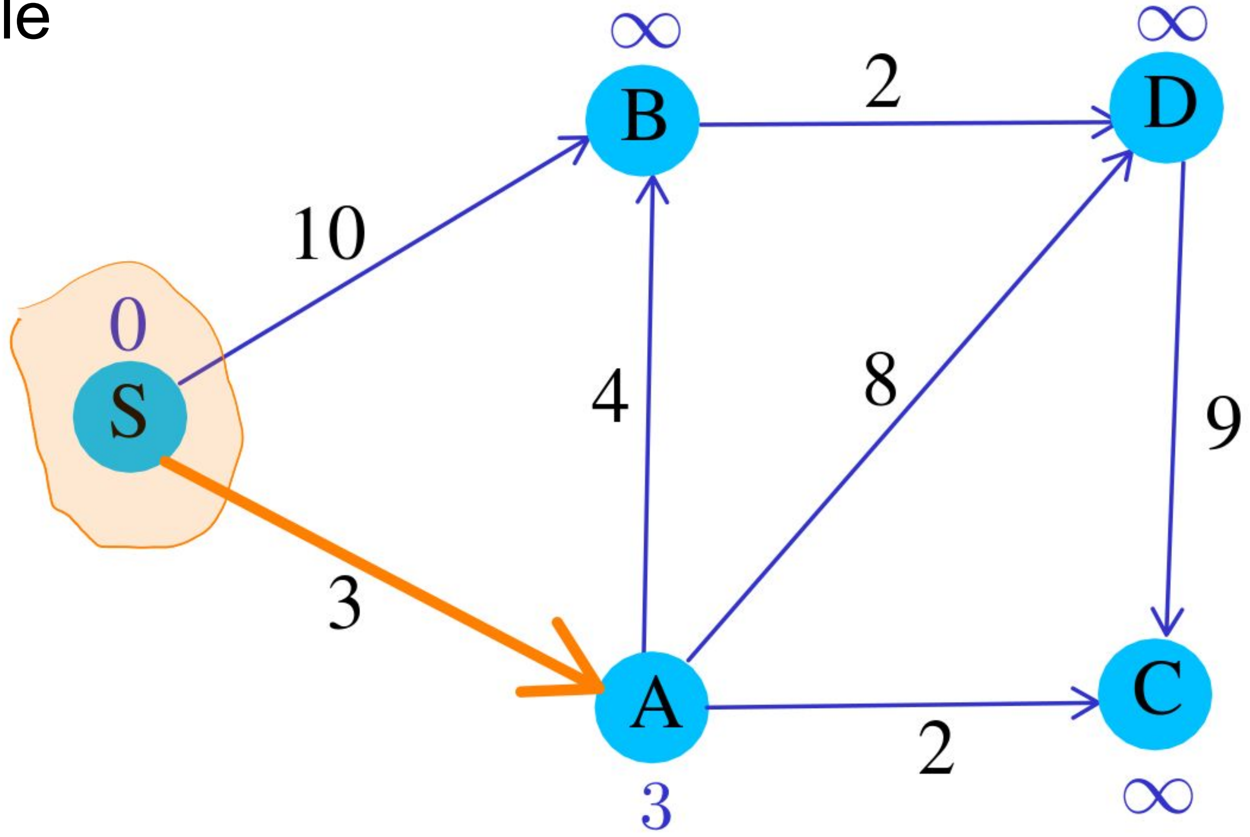
# Example



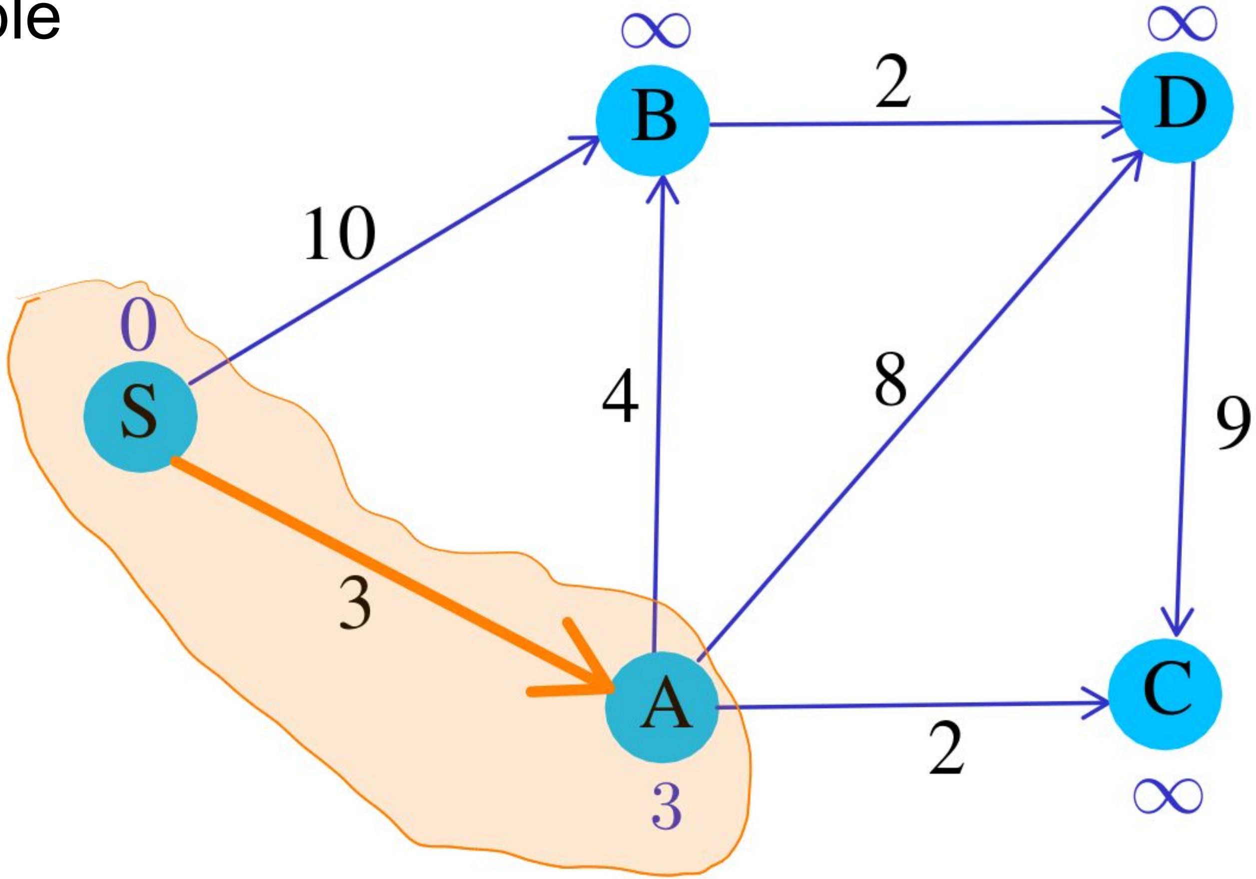
# Example



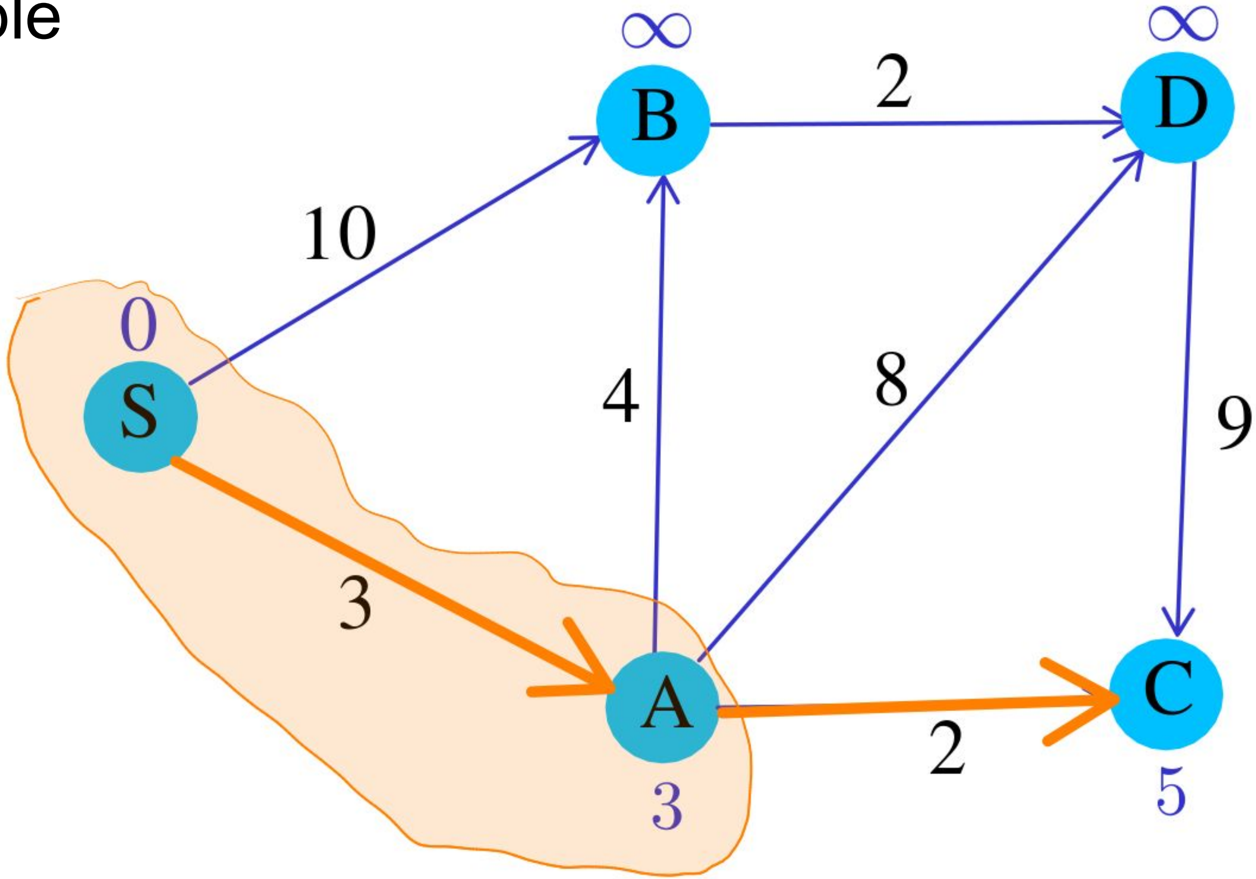
Example



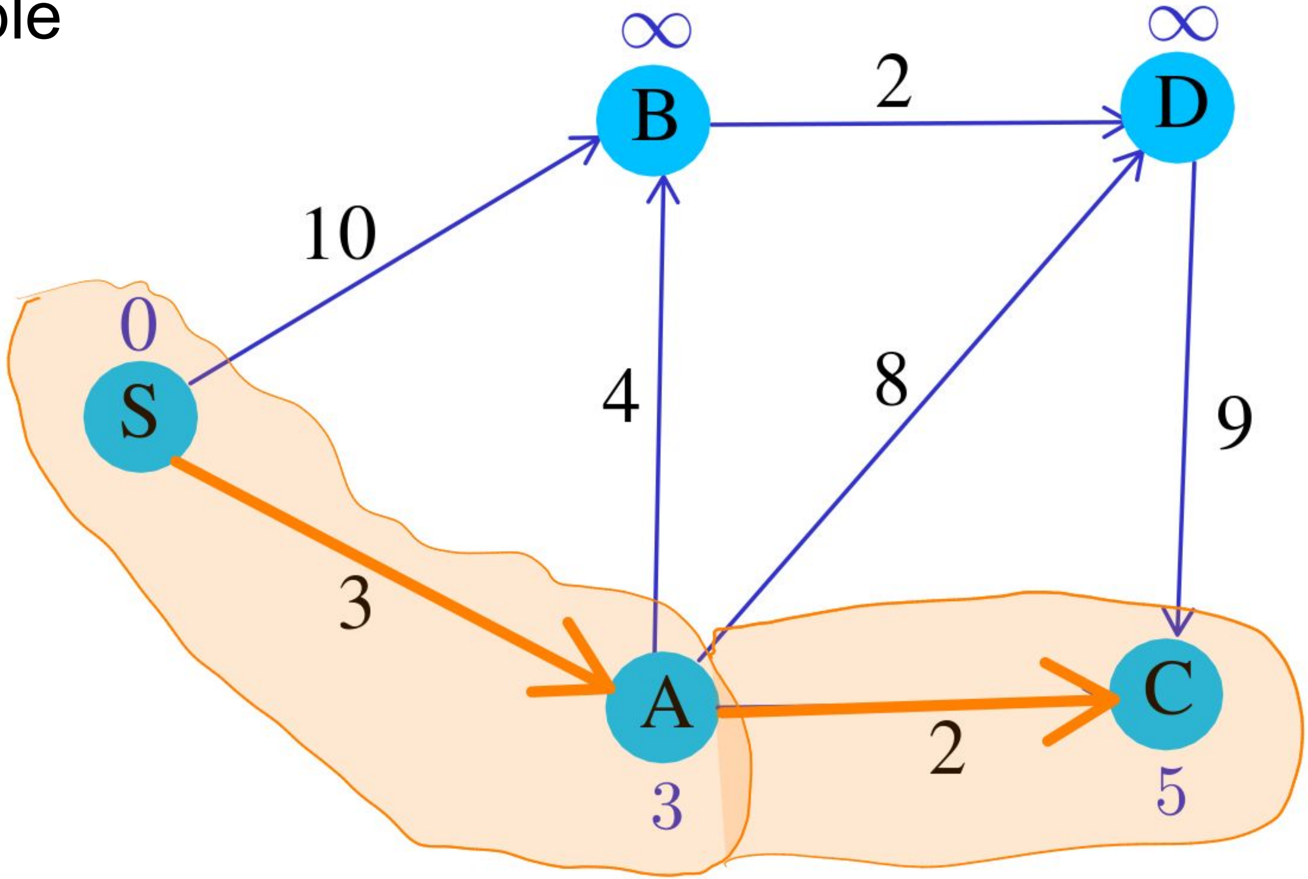
# Example



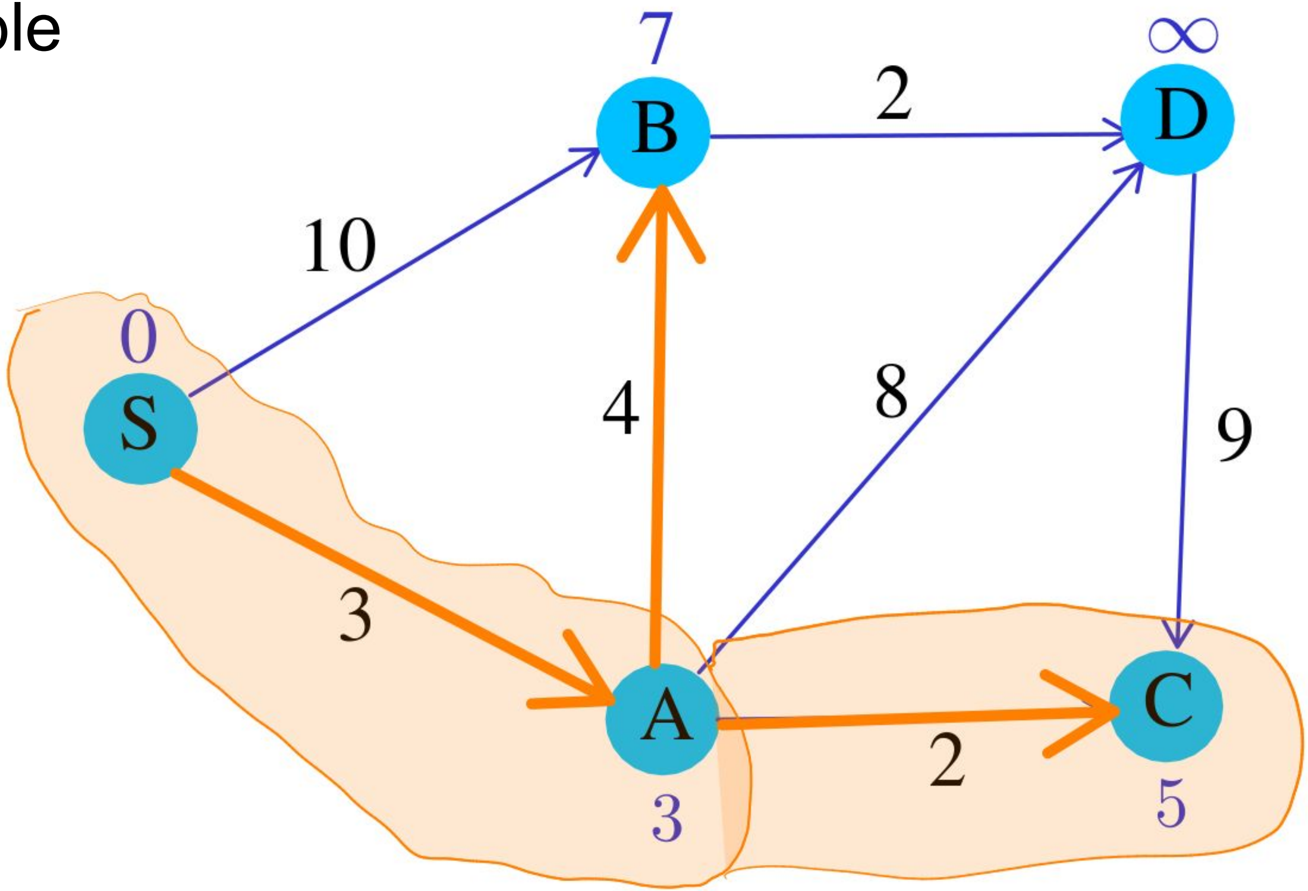
# Example



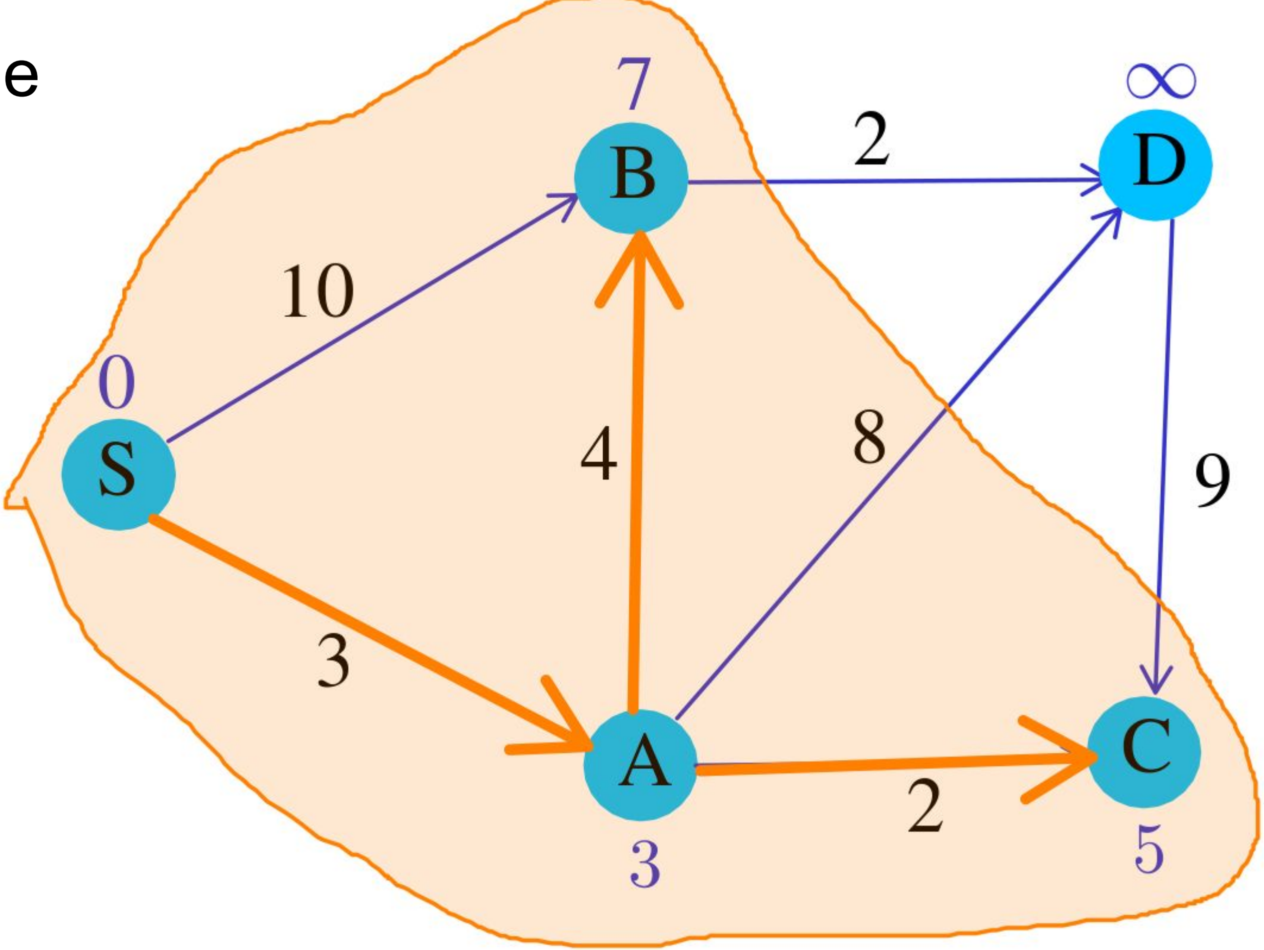
Example



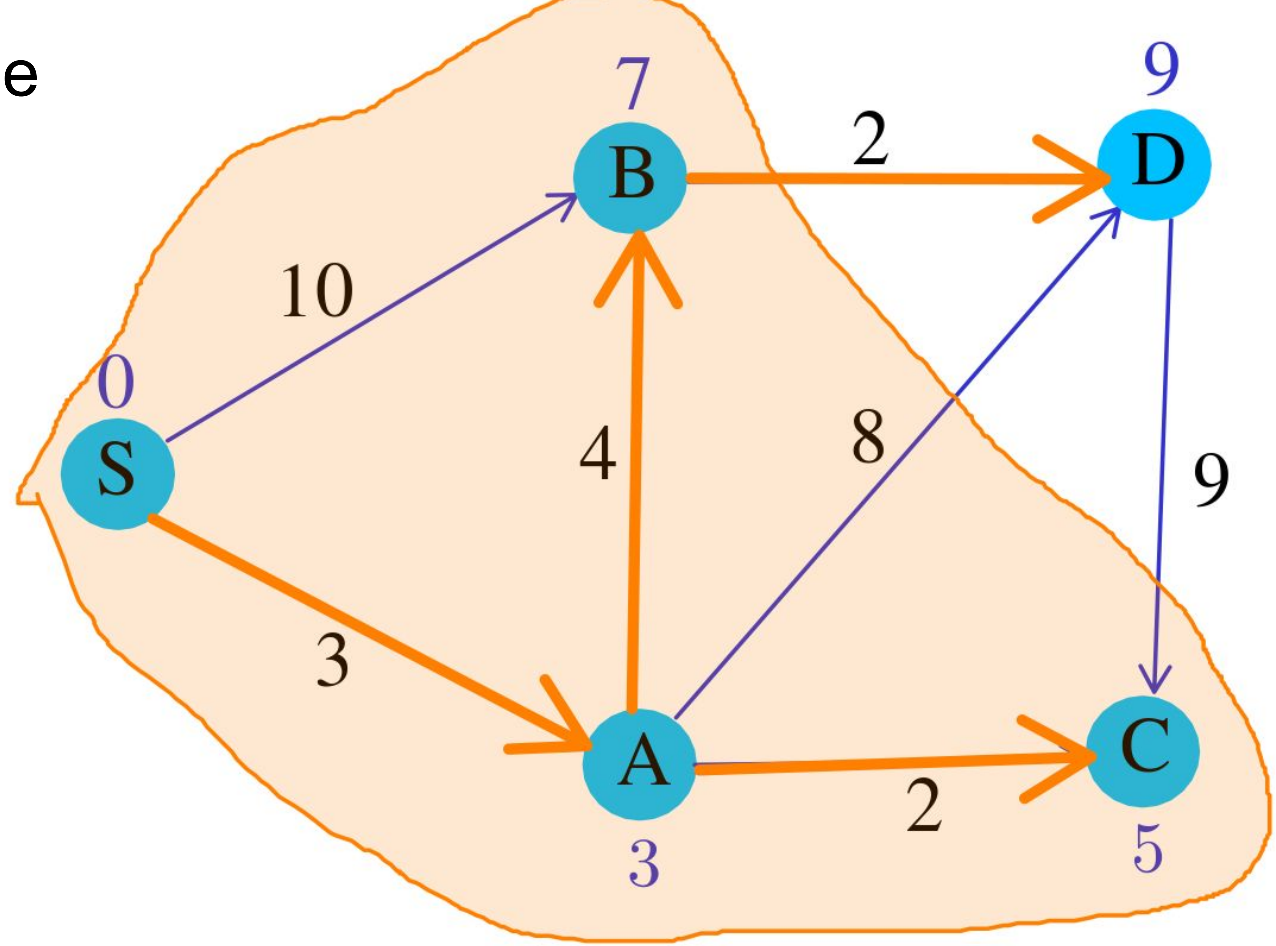
# Example



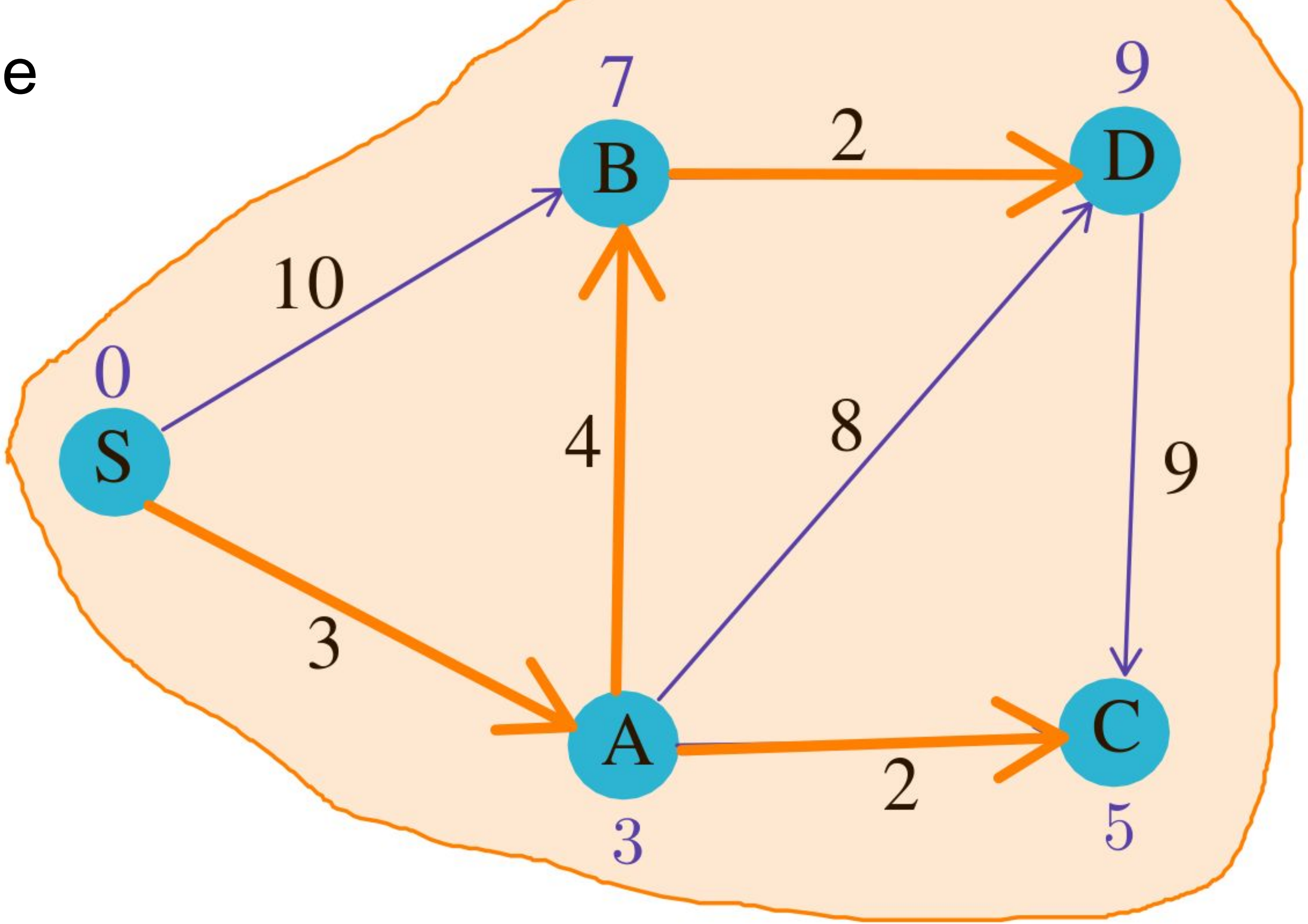
# Example



Example



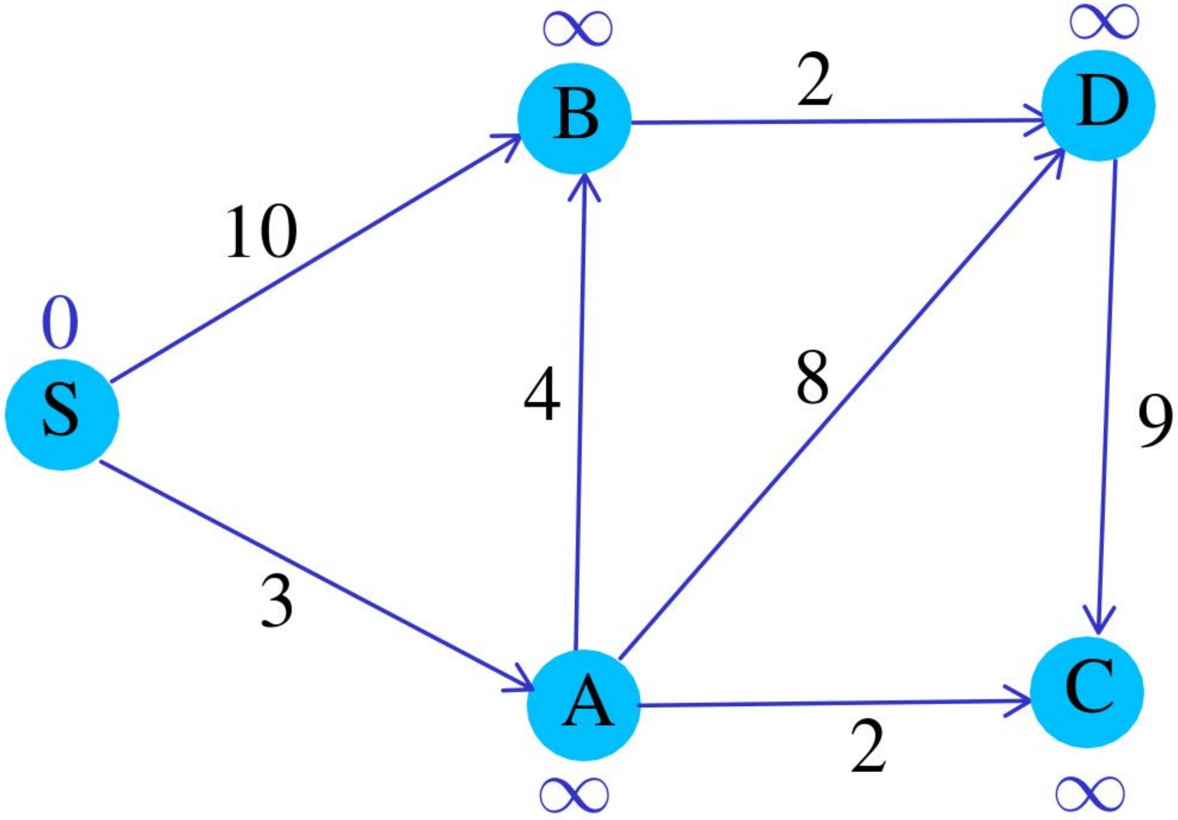
# Example



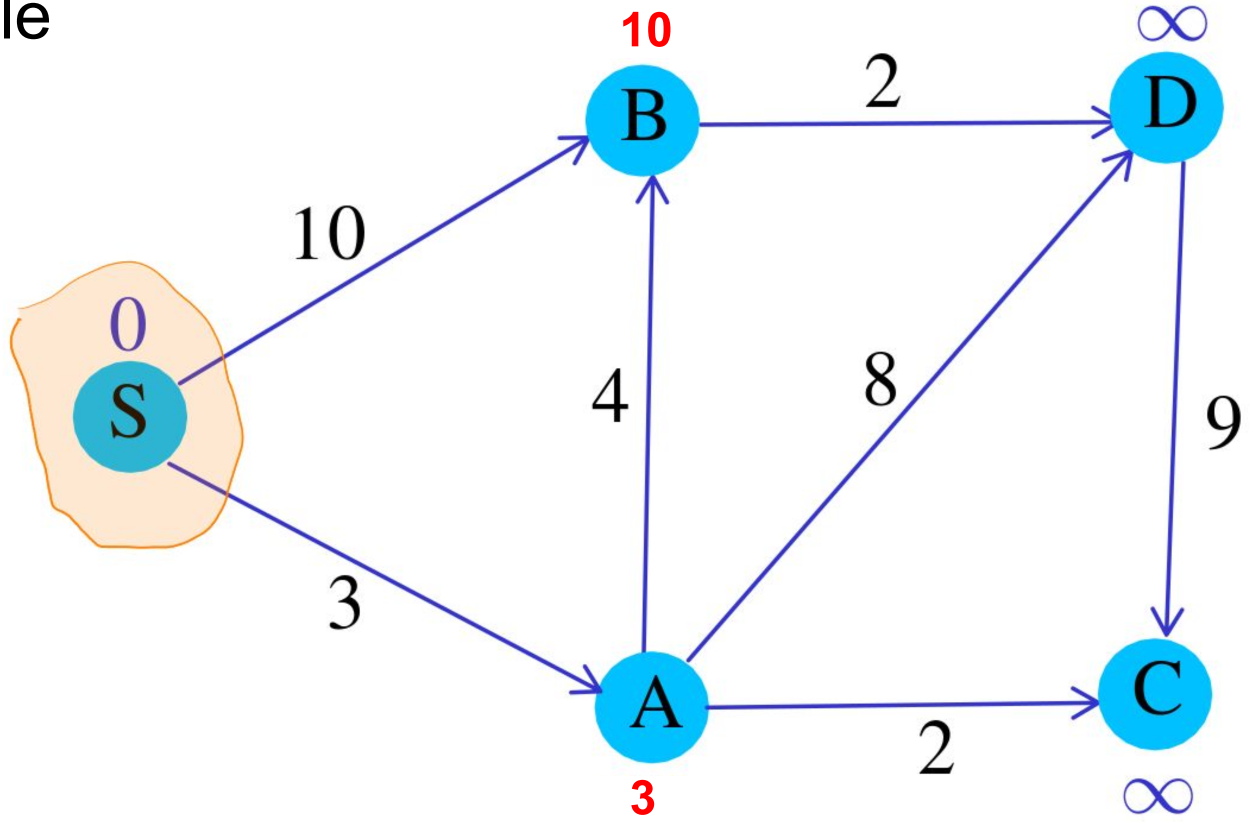
# Example

Add the relaxation step

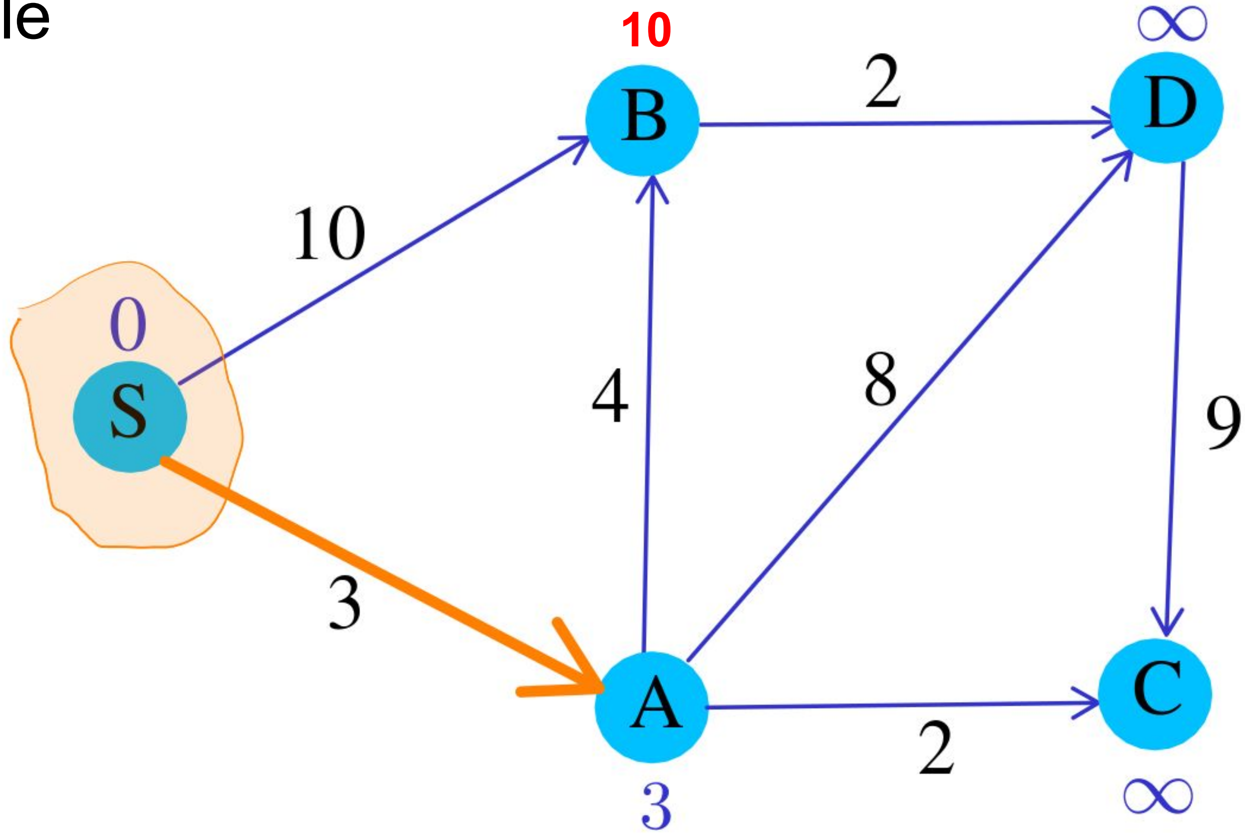
# Example



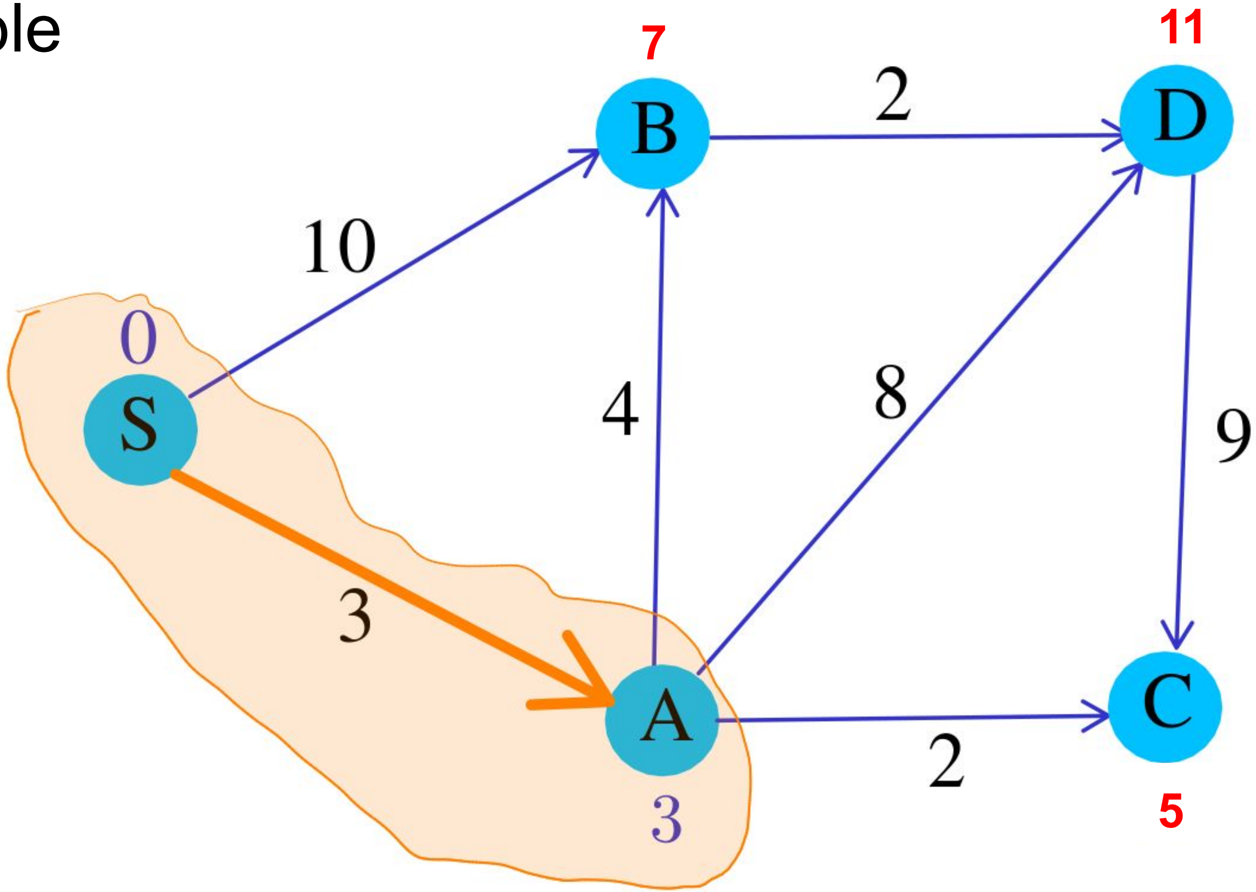
# Example



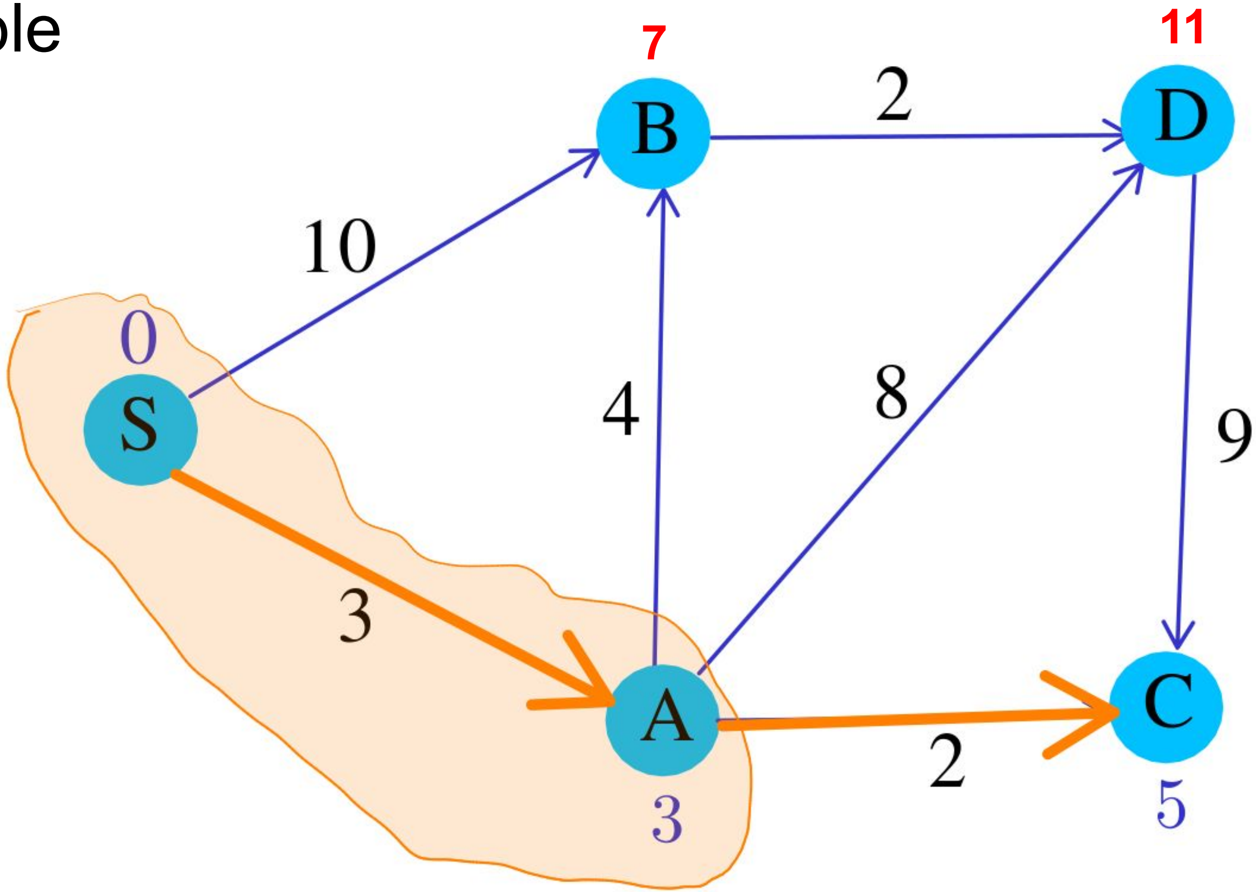
# Example



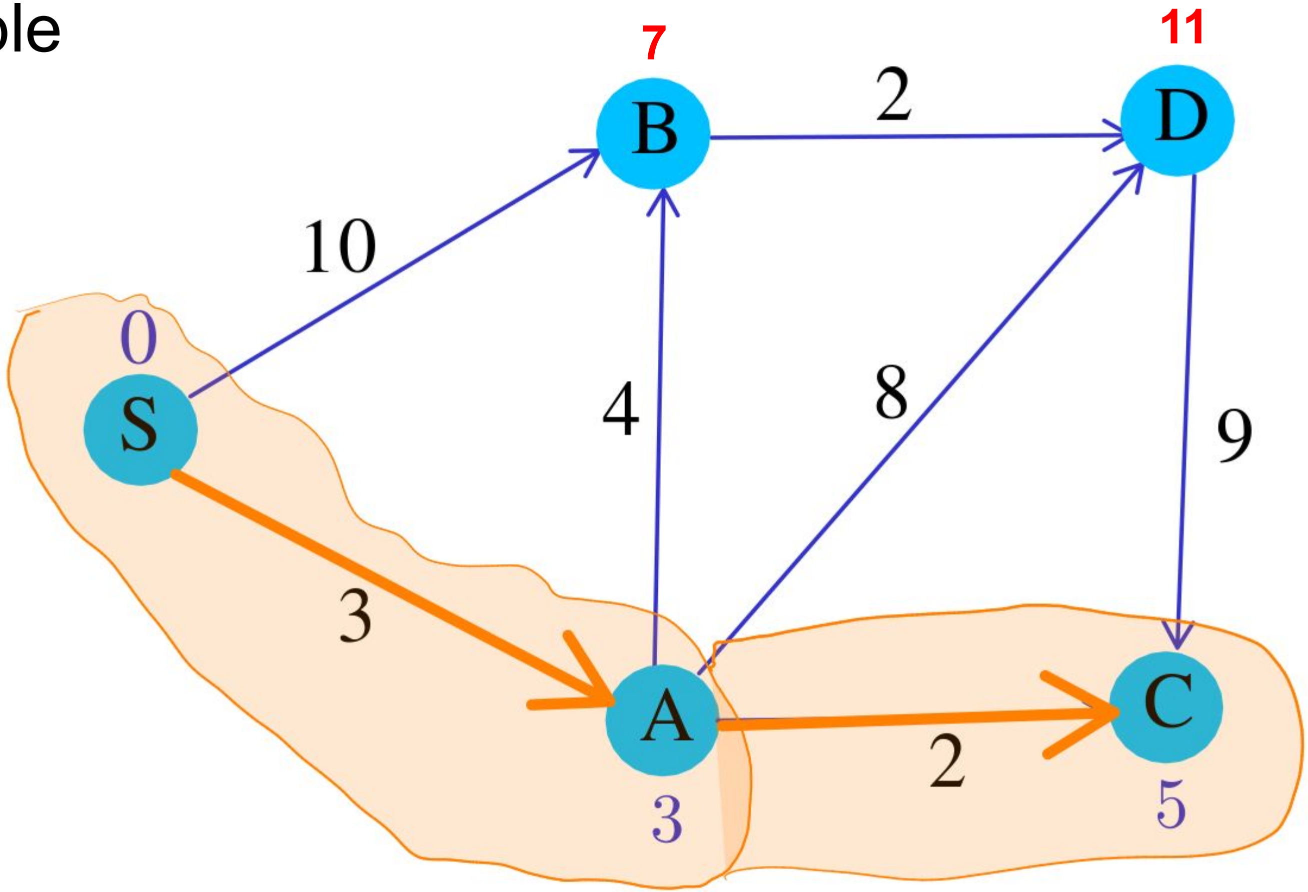
# Example



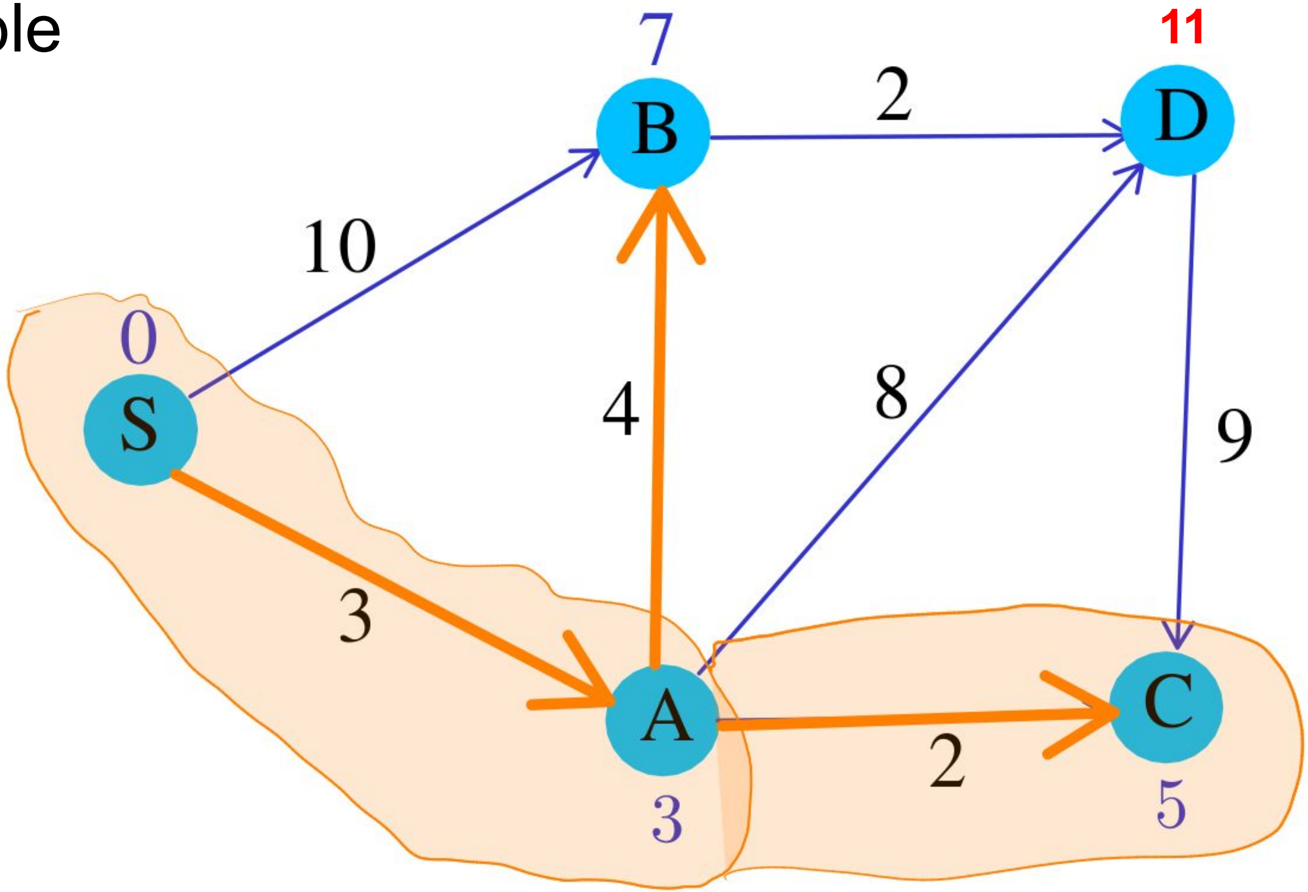
# Example



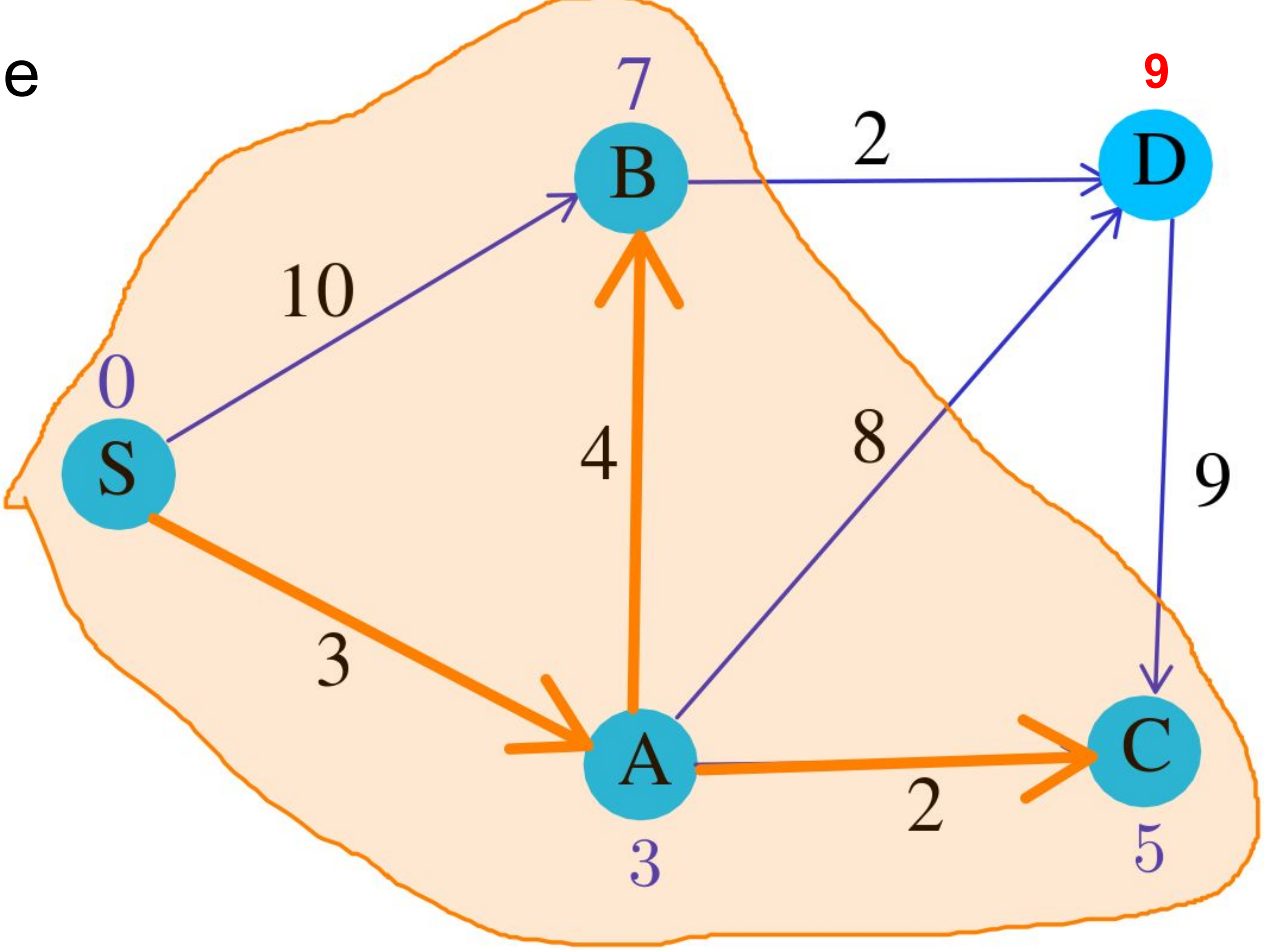
# Example



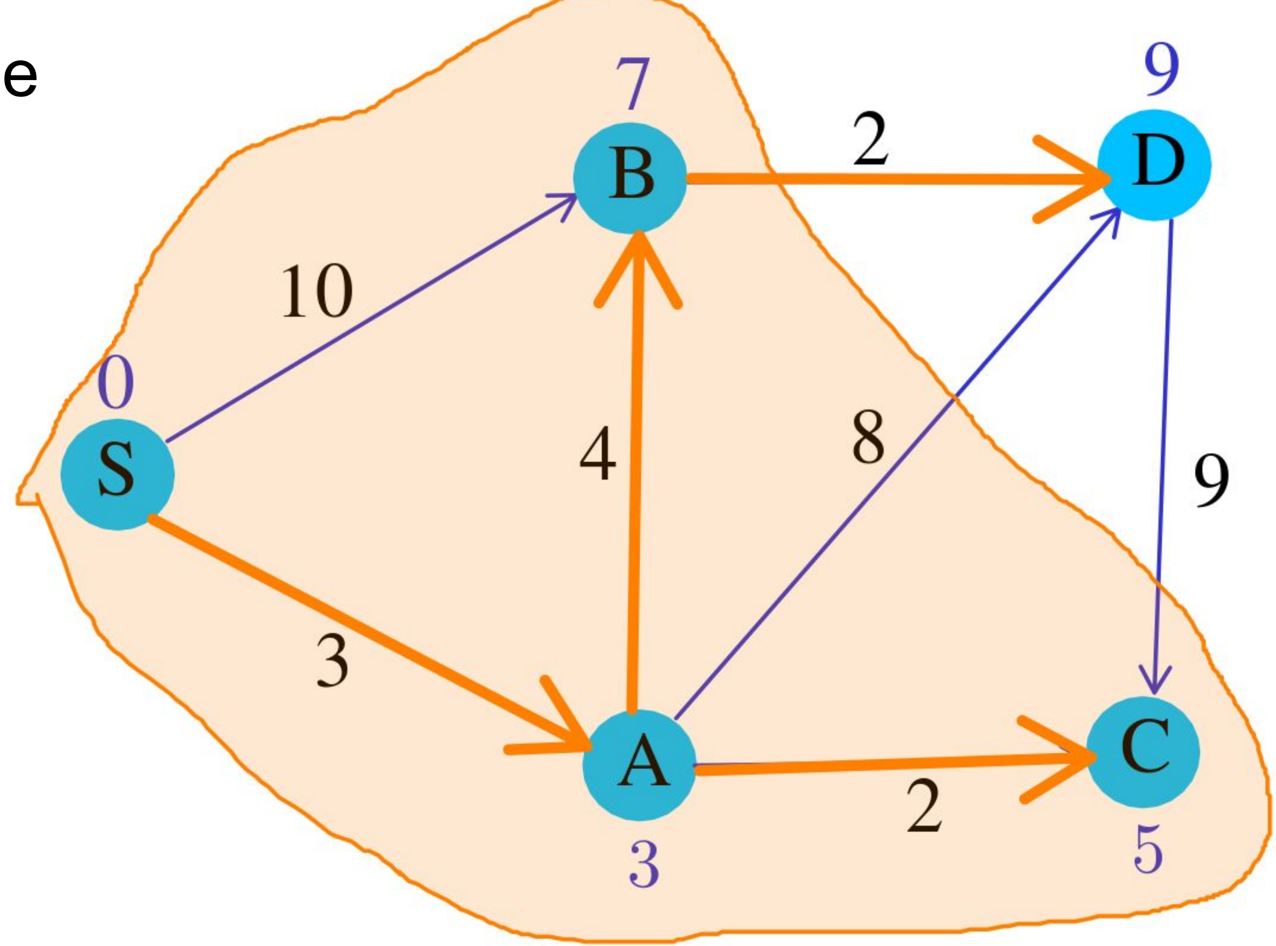
# Example



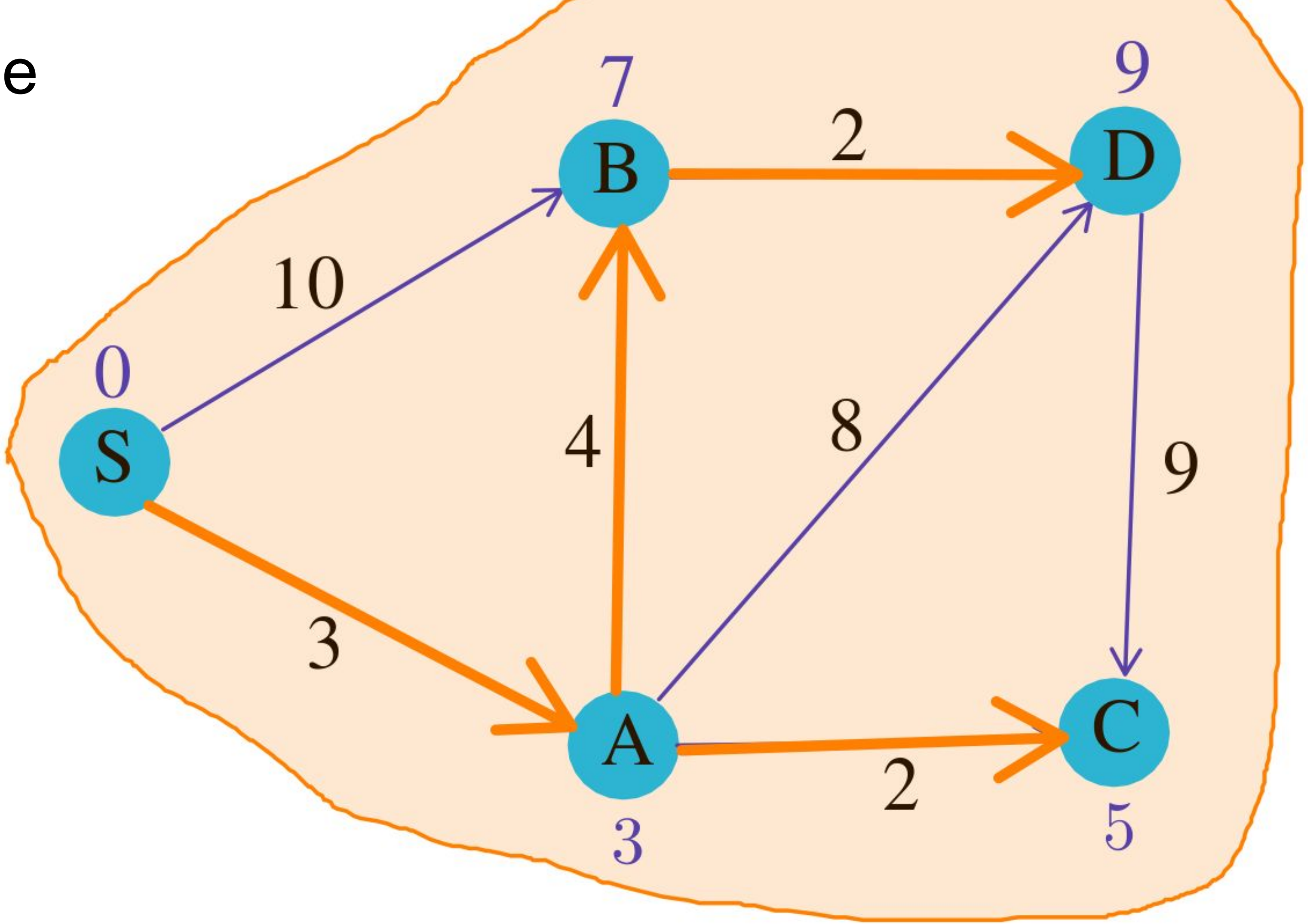
# Example



Example



# Example



# Dijkstra's Algorithm: Implementation

```
d[s] = 0
for each v ∈ V - {s}
    d[v] = ∞
B = ∅
Q = V
while Q ≠ ∅
    u ← EXTRACT-MIN(Q)
    B ← B ∪ {u}
    for each v ∈ Adj[u]
        if (v ∉ B) and ( d[v] > d[u] + w(u, v) )
            d[v] = d[u] + w(u, v)
```

Relaxation step

# Dijkstra's Algorithm: Implementation

```
d[s] = 0
for each v ∈ V - {s}
    d[v] = ∞
B = ∅
Q = V
while Q ≠ ∅
    u ← EXTRACT-MIN(Q)
    B ← B ∪ {u}
    for each v ∈ Adj[u]
        if (v ∉ B) and ( d[v] > d[u] + w(u, v) )
            d[v] = d[u] + w(u, v)
            parent[v] = u
```

Relaxation step

# Dijkstra's Algorithm: Runtime Analysis

```
           d[s] = 0
O(V) →     for each v ∈ V - {s}
               d[v] = ∞
           B = ∅
           Q = V
O(V) →     while Q ≠ ∅
O(V) →         u ← EXTRACT-MIN(Q)
               B ← B ∪ {u}
O(deg(u)) →    for each v ∈ Adj[u]
                   if (v ∉ B) and ( d[v] > d[u] + w(u, v) )
                       d[v] = d[u] + w(u, v)
                       parent[v] = u
```

Relaxation step

**Runtime**: If the distances are stored in an array: **$O(V^2 + E) = O(V^2)$**

# Dijkstra's Algorithm: Runtime Analysis

**Q** is a min-heap maintaining V–B. The key of each node v is d[v]

```
          d[s] = 0
O(V) →    for each v ∈ V - {s}
              d[v] = ∞
          B = ∅
          Q = V
O(V) →    while Q ≠ ∅
O(log V) →    u ← EXTRACT-MIN(Q)
              B ← B ∪ {u}
O(deg(u)) →   for each v ∈ Adj[u]
                  if (v ∉ B) and ( d[v] > d[u] + w(u, v) )
O(log V)              d[v] = d[u] + w(u, v)  #Decrease key of v to d[v]
                      parent[v] = u
```

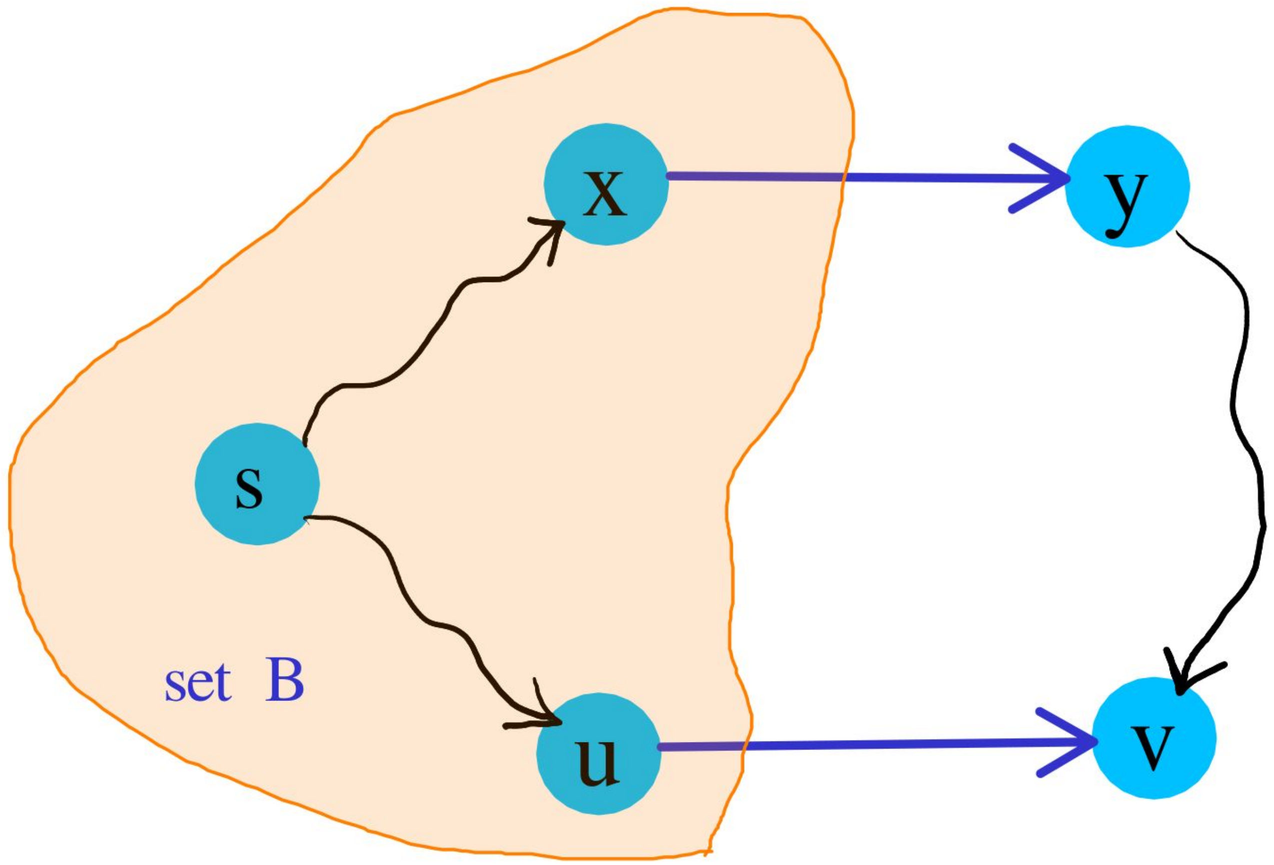
Relaxation step

**Runtime:** if the distances are stored in a priority queue(heap)

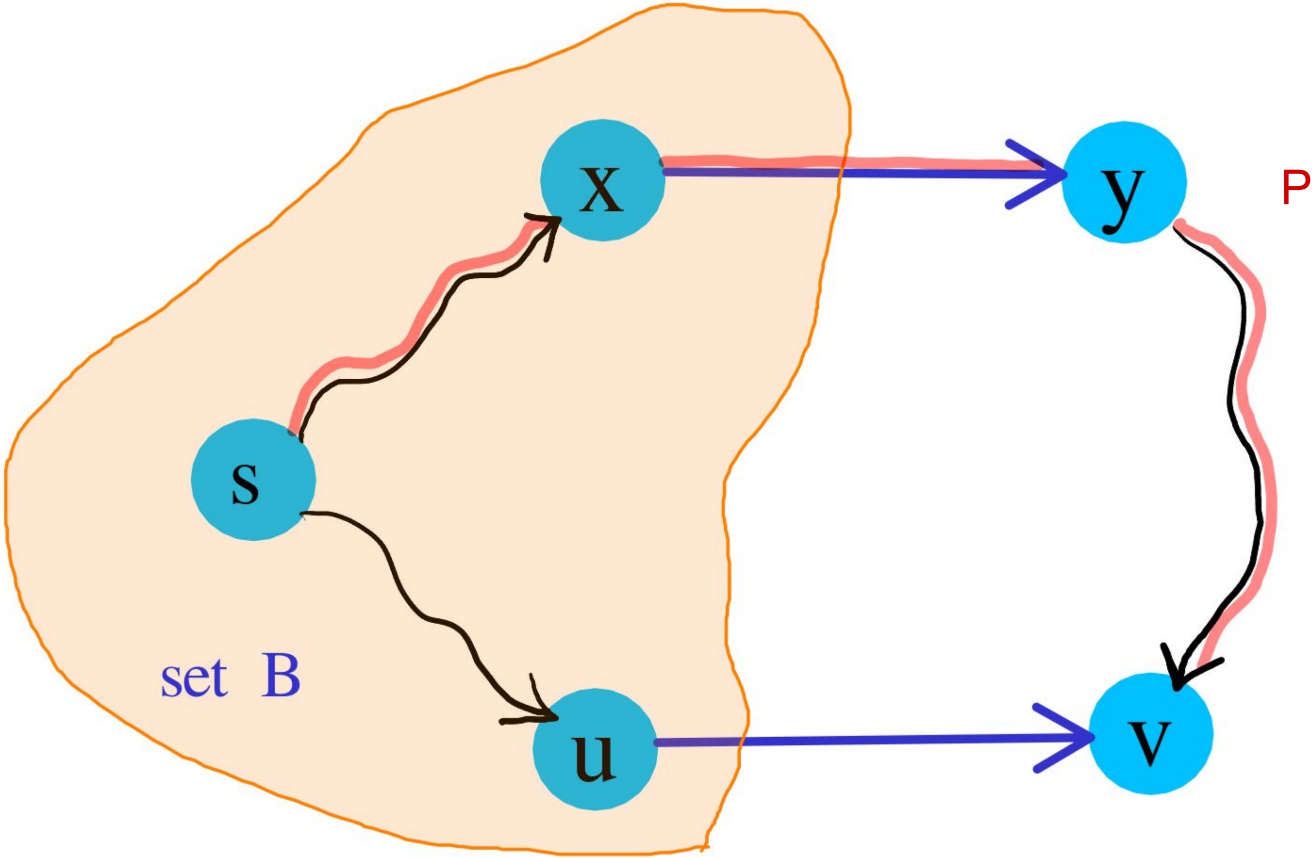
**O(V log V + E log V) = O(E log V)**

# Correctness Proof

- We prove by induction on size of of B.
- **T(k)**: |B| = k, for all u $\in$ B, **d[u]** is the length of the shortest path to all vertices u in B
  - **Base case**: T(1) is always true. In this case B={s}, |B| = 1, and d(s) = 0
  - **Induction Hypothesis**: Suppose T(k) is true
  - **Induction Step**: Prove T(k+1) is true
    - Suppose **v** is the vertex k+1 that is added by an edge (u,v)
    - d[v] = d[u] + w(u, v) (is done by algorithm)
    - $\mathbf{P_v}$: shortest path from s to v ( (u,v) is the final edge on s-v path $P_v$)
    - For contradiction, suppose $P_v$ is not the shortest path to v, say another path P is shorter
    - This path must leave the set B somewhere. Let y be the first node on P that is not in B, and let x in B be the node just before y
- w(P) ≥ w (path from s to y) ≥ w(path from s to x) + w(x, y)
- ≥ w(shortest path from s to x) + w(x, y)= d[x] + w(x,y)
- **w(P) ≥ d[x] + w(x, y) ≥ d[u] + w(u, v) = d[v]**
- w(P) ≥ d[v] for any other path P from s to v



x
y
s
set B
u
v




$$w(P) \geq w(\text{path from } s \text{ to } y) \geq d[x] + w(x, y) \geq d[u] + w(u, v) = w(P_v) = d[v]$$

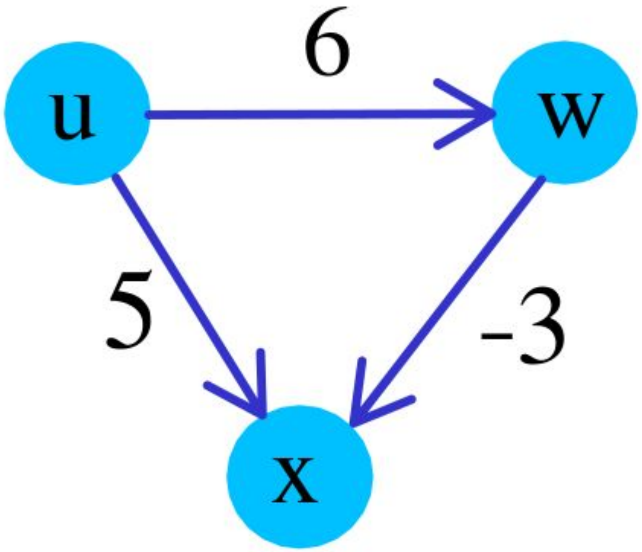
# Dijkstra

Dijkstra was known for many contributions to computer science, e.g., structured programming, concurrent programming. He designed the above algorithm to demonstrate the capabilities of a new computer (to find railway journeys in the Netherlands). At that time (the 50’s) the result was not considered important. He wrote:

- At the time, algorithms were hardly considered a scientific topic. I wouldn’t have known where to publish it... The mathematical culture of the day was very much identified with the continuum and infinity. Could a finite discrete problem be of any interest? The number of paths from here to there on a finite graph is finite; each path is a finite length; you must search for the minimum of a finite set. Any finite set has a minimum — next problem, please. It was not considered mathematically respectable.

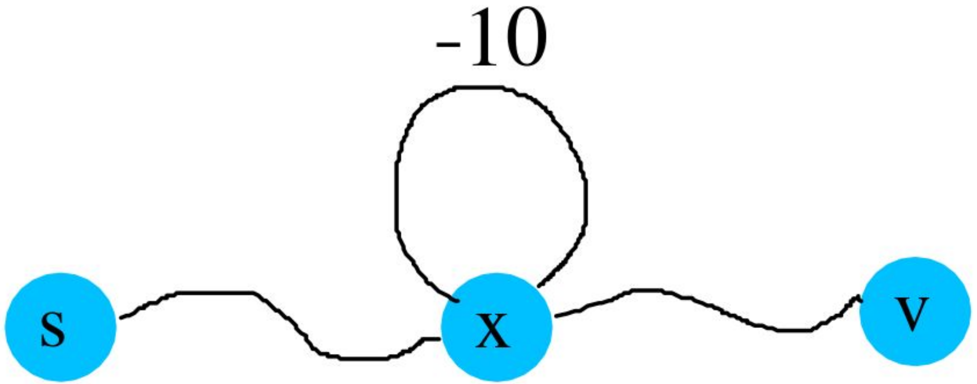
# What if a graph has negative-weights edges?

Dijkstra algorithm fails

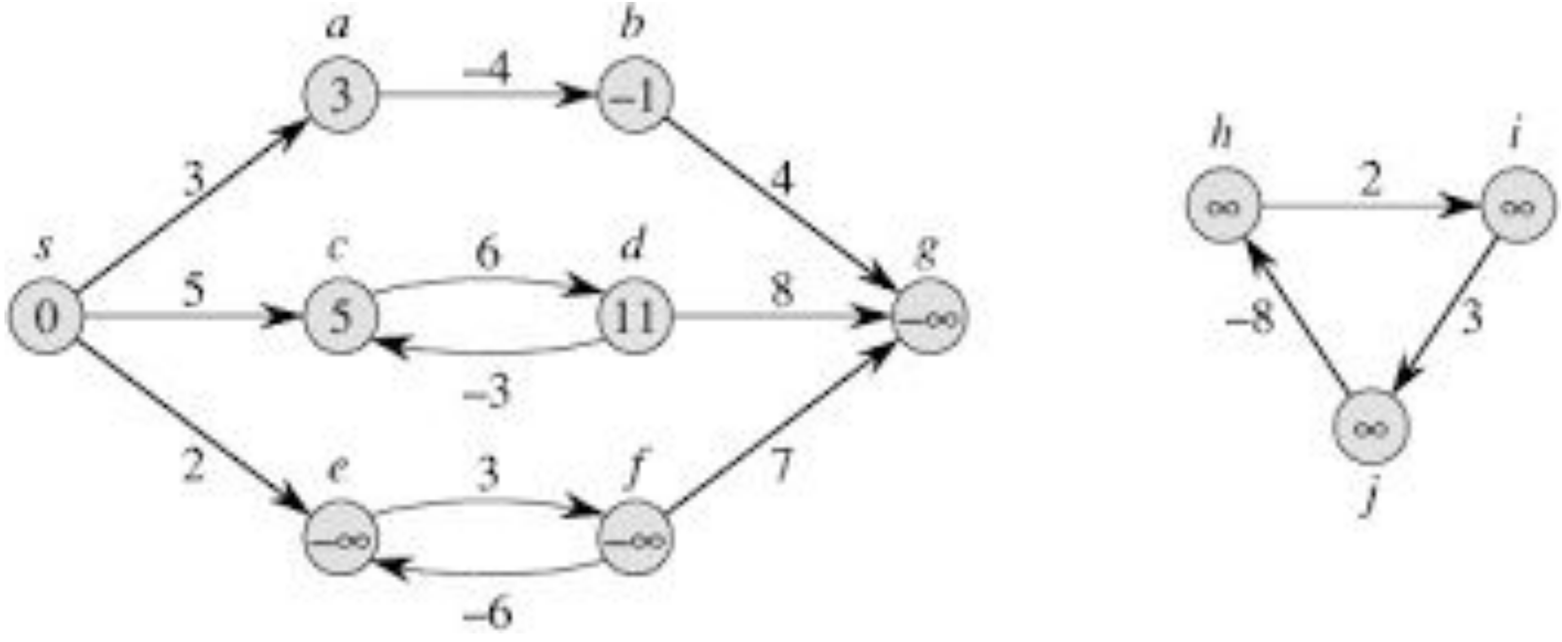


# What if a graph has negative-weights cycles?

- If the graph G contains a negative-weight cycle reachable from s
  - We can go around the negative cycle as many times as we want
  - shortest-path weights are not well defined
- If G contains no negative-weight cycle reachable from the source s
  - for all $v \in V$, the shortest-path weight is well defined (it could have a negative value)
- What is the meaning of the weights?



# Negative edge weights in a directed graph




Each vertex contains the shortest-path weight from source s.

# Cycles in a shortest path

- Can a shortest path contain a cycle?
  - negative-weight cycle
  - positive-weight cycle
  - 0-weight cycle
    - →Shortest paths are simple: can contains at most |v| distinct vertices and at most |V|-1 edges

# Shortest Path Algorithms

- Given u and v, find shortest uv path
  - Involves solving the more general problem
- Given u, find shortest uv path for evey v in V
- Single-source-shortest path problem
  - Unweighted graphs
    - BFS
  - Weighted graphs (non-negative weights)
    - Greedy algorithm: Dijkstra
  - Directed Acyclic graphs
  - General weights (negative and non-negative weights) but no negative cycle
    - Dynamic programming: Bellman-ford algorithm
- All pairs shortest path

# Single-source shortest path in DAG

Shortest paths are always well defined in a DAG, Since there are no negative-weight cycle in a graph

- If the DAG contains a path from u to v, u precedes v in the topological sort
- If u comes before v in the topological order, there is no path from v to u

# Single-source shortest path in DAG

DAG-Shortest-Paths(G, s)

Topologically sort the vertices of G

$d[s] \leftarrow 0$

**for** each $v \in V - \{s\}$

**do** $d[v] \leftarrow \infty$

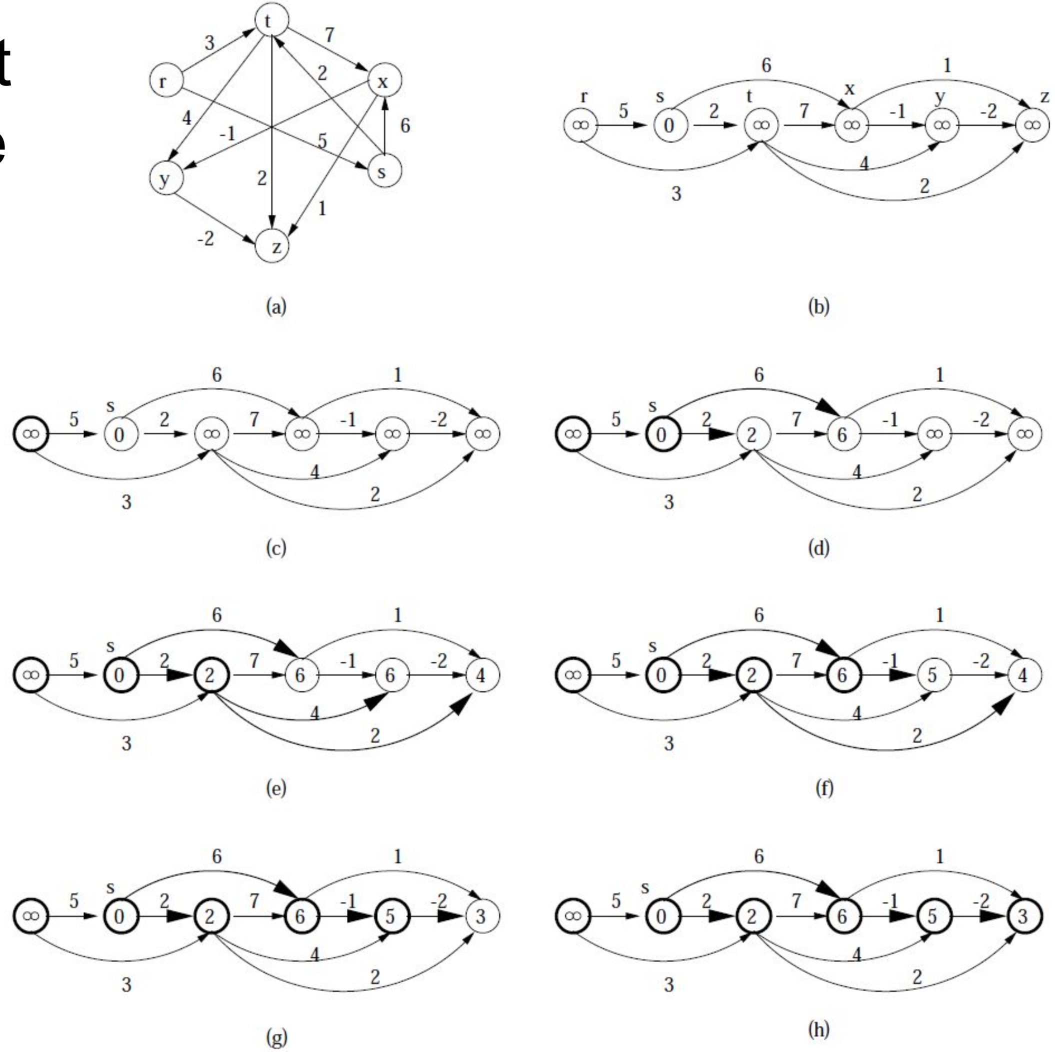
**for** each vertex $u,$ taken in topologically sorted order

**for** each $v \in Adj[u]$

**if** $d[v] > d[u] + w(u, v)$

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

# Single-source shortest path in DAG: Example



# Single-source shortest path in DAG: Runtime: Θ(V+E)

DAG-Shortest-Paths(G, s)

Topologically sort the vertices of G

$d[s] \leftarrow 0$

**for** each $v \in V - \{s\}$

**do** $d[v] \leftarrow \infty$

Θ(V)

**for** each vertex $u$, taken in topologically sorted order

**for** each $v \in Adj[u]$

**if** $d[v] > d[u] + w(u, v)$

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

Θ(V+E)

# Single-source shortest path in DAG: Correctness

**Theorem**. When the algorithm terminates, $d[v] = \delta(s, v)$ for all vertices $v \in V$

**Proof.**

- If v is not reachable from s, then $d[v] = \delta(s, v) = \infty$
- If v is reachable from s, there is a shortest path $p = \langle v_0, v_1, \dots, v_k \rangle$ where $v_0 = s$ and $v_k = v$.
- The algorithm process the vertices in topologically sorted order
- Therefore, the edges on p are relaxed in the order $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$
- We can prove by induction on the number of relaxation steps that $d[v] = \delta(s, v)$

# Single-source shortest path in DAG: Correctness

- **Theorem.** After the k-th edge of path p is relaxed, we have $d[v_k] = \delta(s, v_k)$
- **Proof by induction:** induction on the number of relaxation steps.
- **Induction hypothesis:** After the i-th edge of path p is relaxed, $d[v_i] = \delta(s, v_i)$
- **Base Case: i=0**
  - before any edge of p have been relaxed, we have $d[v_0] = d[s] = 0 = \delta(s, s)$
- **Induction step.** Assuming $d[v_{i-1}] = \delta(s, v_{i-1})$ after the (i-1)-th edge was relaxed → we want to show that $d[v_i] = \delta(s, v_i)$ after the i-th edge is relaxed
  - $\mathbf{d[v_i] \leq \delta(s, v_i)}$
    - After relaxing edge $(v_{i-1},v_i)$, we have $d[v_i] \leq d[v_{i-1}] + w(v_{i-1},v_i)$
      - before relaxing the edge, there are two cases
        - $d[v_i] > d[v_{i-1}] + w(v_{i-1},v_i)$ if this is the case the algorithm does the following
          - $d[v_i] = d[v_{i-1}] + w(v_{i-1},v_i)$
        - $d[v_i] \leq d[v_{i-1}] + w(v_{i-1},v_i)$ if this is the case, no change happen and the property hols
      - $d[v_i] <= d[v_{i-1}] + w(v_{i-1},v_i) = \boldsymbol{\delta(s, v_{i-1}) + w(v_{i-1},v_i) = \delta(s, v_i)}$ (subpaths of shortest path are also shortest path)
  - $\mathbf{d[v_i] \geq \delta(s, v_i)}$
- Therefore $\mathbf{d[v_i] = \delta(s, v_i)}$