Lecture #16
Graph Algorithm (3)

Algorithm

JBNU

Jinhong Jung

In This Lecture

- Discussion on Dijkstra's algorithm
  - Implementation details
  - Discussions

- Single source shortest path with negative edges
  - Bellman-Ford's algorithm

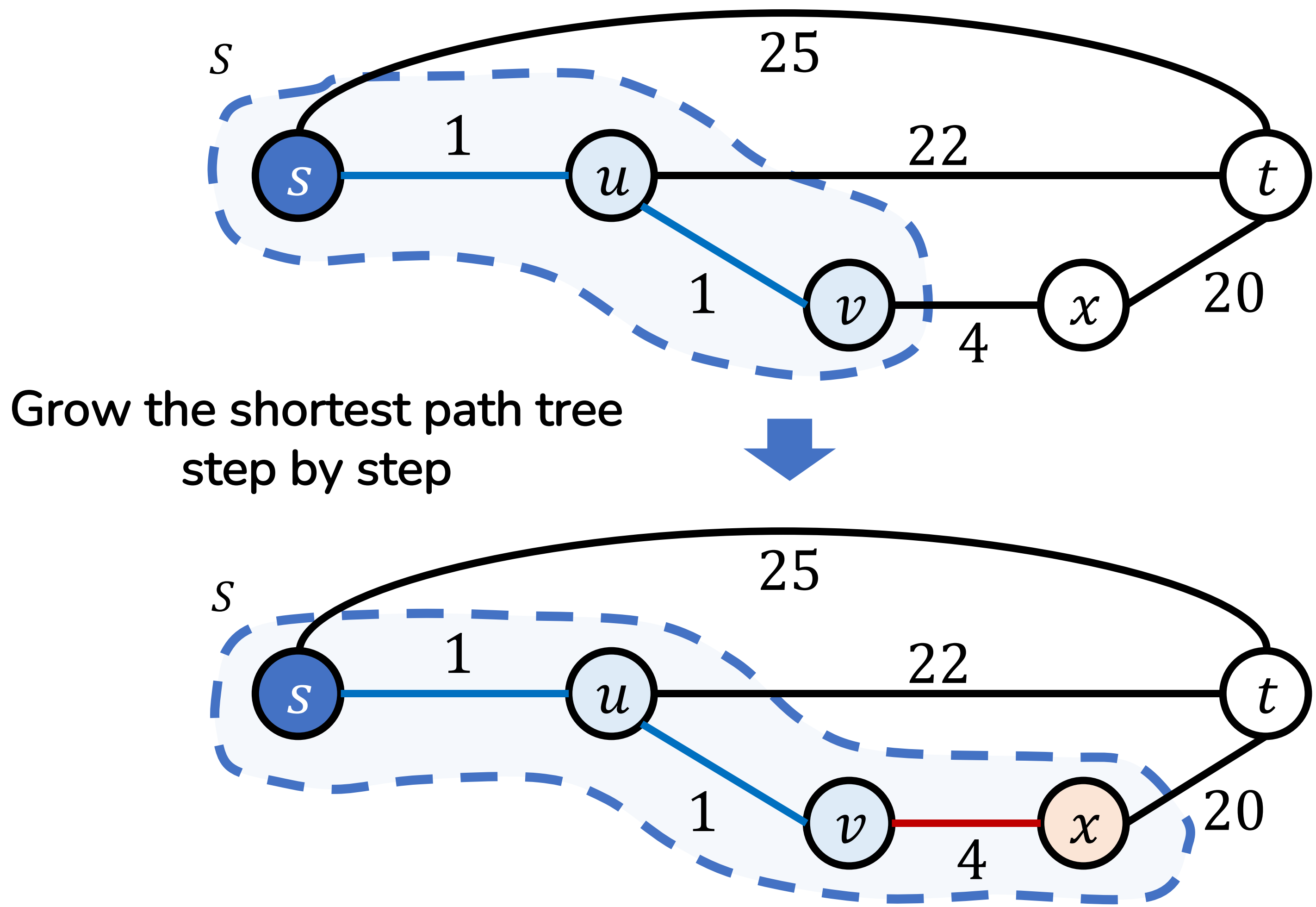
Outline

- ❑ **Dijkstra's algorithm**

- ❑ Bellman-Ford algorithm

Dijkstra's Intuition (Remind)

- **Incrementally grow shortest paths starting from source node s** (i.e., grow the shortest path tree S)



Dijkstra's Algorithm

❑ Pseudocode

```
def dijkstra(G, s):
    S ← ∅                          # set of sure nodes
    for each v in V:
        D[v] ← ∞
    D[s] ← 0 & parent[s] ← s

    while S is not V:
        u ← extract-min(V − S, D)
        for each v in N_u:
            if v ∈ V − S and D[u] + w(u,v) < D[v]:
                D[v] ← D[u] + w(u,v)   # relaxation
                parent[v] ← u          # trace

        S ← S ∪ {u}

    return D and parent
```

Step 1. Initialization
Maintain two node sets
S (sure) and $V - S$ (not-sure)

Step 2-1. Pick a "not-sure" node (smallest estimate)

Step 2-2. Update all "not-sure" neighbors of the selected node u

Step 2-3. Mark the selected node u as "sure"
(can be move up after Step 2-1)

The complexities and implmentation of Dijkstra's algorithm are the same as those of Prim's one

Implmentation Details

❑ Implementation of Dijkstra's algorithm

- Using min-heap to extract node u with smallest $D[u]$

```
def dijkstra(G, s):
    Q ← min-heap()
    for each v in V − {v}:
        D[v] ← ∞ & Q.insert(D[v], v)
    D[s] ← 0 & parent[s] ← s & Q.insert(D[s], s)

    while Q is not empty:
        u ← Q.remove()
        for each v in N_u:
            if v ∈ Q and D[u] + w(u,v) < D[v]:
                D[v] ← D[u] + w(u,v)
                Q.decrease-key(v, D[v])
                parent[v] ← u
```

See the appendix for the version without decrease-key

Correctness Analysis

- **Claim.** When a node u is marked as "sure" by DA, its estimate $D[u]$ is equal to the true shortest distance
  - Let $\delta(s, u)$ be the true shortest distance from s to u
  - **[Sketch] Proof by contradiction**
    - Let's say the claim is false $\Rightarrow D[u] > \delta(s, u)$
    - This derives some errors that contradict the above (details in appendix) $\Rightarrow$ "The claim is false" is false
  - Implies that
    - The shortest path tree in "sure" set is maintained whenever a node is added into the set.
    - **The tree expanded by DA is always a shortest path tree!**

Discussion (1)

❑ Can we find SSSP in Kruskal's approach?

- No, since we cannot track an SST from a source s in a forest.

❑ Dijkstra's algorithm is greedy algorithm

- At each time, Dijkstra selects the best crossing edge making the distance from s to its end-point smallest (like Prim's).

❑ Dijkstra's algorithm is dynamic programming

- Distances are updated using previously calculated values through relaxation (but, Prim's algorithm is not DP).

Prim's algorithm

$$\textbf{if } v \in Q \textbf{ and } w(u,v) < c[v]\textbf{:}$$
$$c[v] \leftarrow w(u,v)$$

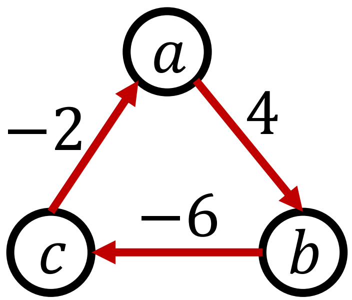
Dijkstra's algorithm

$$\textbf{if } v \in Q \textbf{ and } D[u] + w(u,v) < D[v]\textbf{:}$$
$$D[v] \leftarrow \underbrace{D[u]}_{\text{Use previous solution}} + w(u,v)$$

Discussion (2)

❑ No negative edge weights are allowed

- e.g., profit (+) and loss (−) in trading networks
- **C1.** What if there is a negative cycle?
  - A cycle is negative when its cost (sum of edge weights) is negative



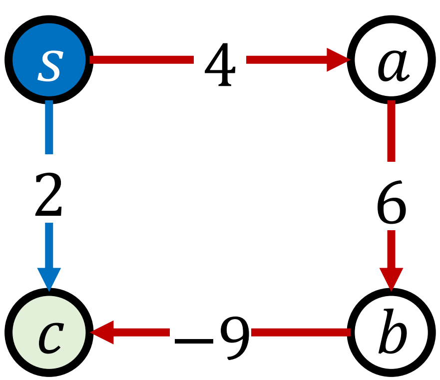
Shortest path cannot be defined

A cycle C: $a \rightarrow b \rightarrow c \rightarrow a \Rightarrow -4$

$C \rightarrow C \Rightarrow -8$

$C \rightarrow \cdots \rightarrow C \Rightarrow -\infty$

- **C2.** What if there is an edge with a negative weight?



DA chooses $s \rightarrow c$ (2) immediately as the shortest path while the true shortest path is $s \rightarrow a \rightarrow b \rightarrow c$ (1)

Outline

- Dijkstra’s algorithm

- **Bellman-Ford algorithm**

Bellman-Ford Algorithm

❑ Find shortest paths with negative edges

- **Input**: a weighted graph G & a source node s
  - G can have negative edges, but **cannot have negative cycles.**
  - **A negative edge cannot be undirected since it forms a negative cycle.**
- **Output**: shortest paths from s to all other nodes
  - Result in the shortest path tree rooted at s

❑ Bellman-Ford algorithm

- Performs in $O(mn)$ time
  - Slower than Dijkstra's algorithm
  - But, can handle negative edges
- Named by Richard Bellman & Lester Ford in 1950s

Richard Bellman

Lester Ford

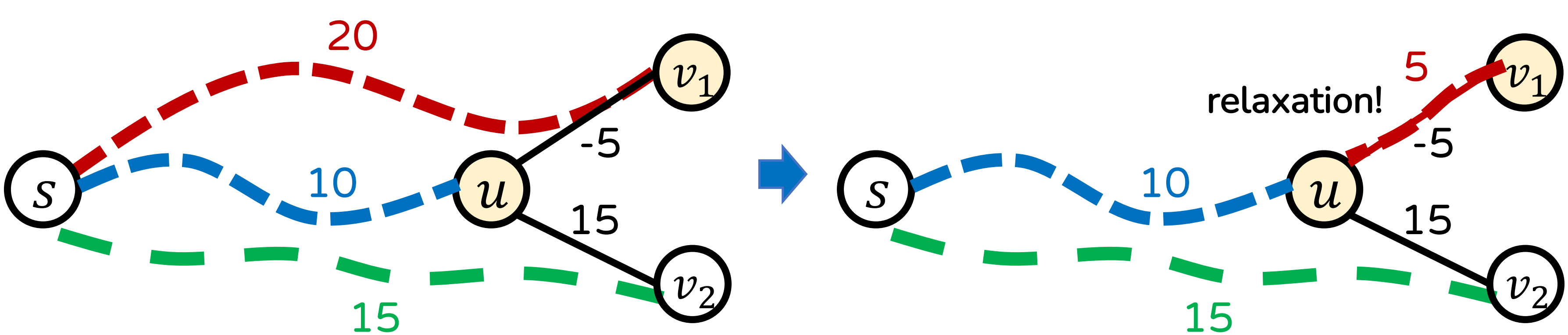
Bellman-Ford's Intuition

❑ Let's consider shortest paths $s \rightsquigarrow v$ for any node v

- Assume each path is composed of at most i edges.
- For each edge (u, v), let's perform relaxation on (u, v).
  - If (u, v) makes a shorter path (new) $s \rightsquigarrow u \to v$ than (old) $s \rightsquigarrow v$, we can find another shortest path $s \rightsquigarrow u \to v$ consisting of at most $i + 1$ edges

Shortest paths consisting of at most i edges

Shortest paths consisting of at most $i + 1$ edges



Bellman-Ford's Intuition

n is # of nodes

❑ Let's repeat relaxations on all edges $n-1$ times!

- Each shortest path consists of at most $n-1$ edges, and **true shortest paths from s are checked during this process**!
- Why?
  - Assume there is a shortest path P having more $n-1$ edges
  - ⇒ there is at least one cycle because there are n nodes
  - ⇒ these cycles are positive (by the definition of G)
  - ⇒ if we exclude these cycles from P, the distance of the modified path is equal to or shorter than P, which contradicts to the assumption
  - Thus, true shortest paths must be in shortest paths consisting of at most $n-1$ edges

Bellman-Ford's Intuition

❑ Process of Bellman-Ford algorithm

- **Step 1.** Initialize the estimate $D_i[v]$ for each node v for i-th iteration
  - Initialize $D_0[s] = 0$ & $D_0[v] = \infty$ for all other nodes v

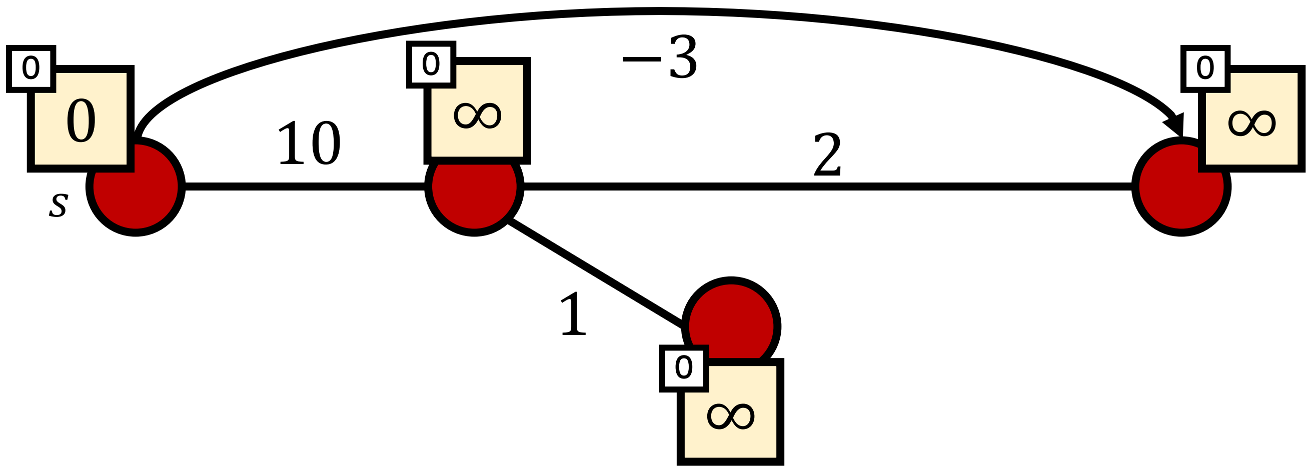
- **Step 2.** Perform relaxations of all of edges for i-th iteration
  - For each node u, relax neighbor v of node u whose $D_{i-1}[u]$ is not ∞, i.e., $D_i[v] = \min\big(D_{i-1}[v], D_{i-1}[u] + w(u, v)\big)$ if $D_{i-1}[u] \neq \infty$

- Repeat Step 2, $n - 1$ times
  - No more update occurs after $(n - 1)$-th iteration

Example (1)

❑ Step 1. Initialize the estimate $D_i[v]$ for each node v

- Initialize $D_0[s] = 0$ & $D_0[v] = \infty$ for all other nodes v



Directed edge

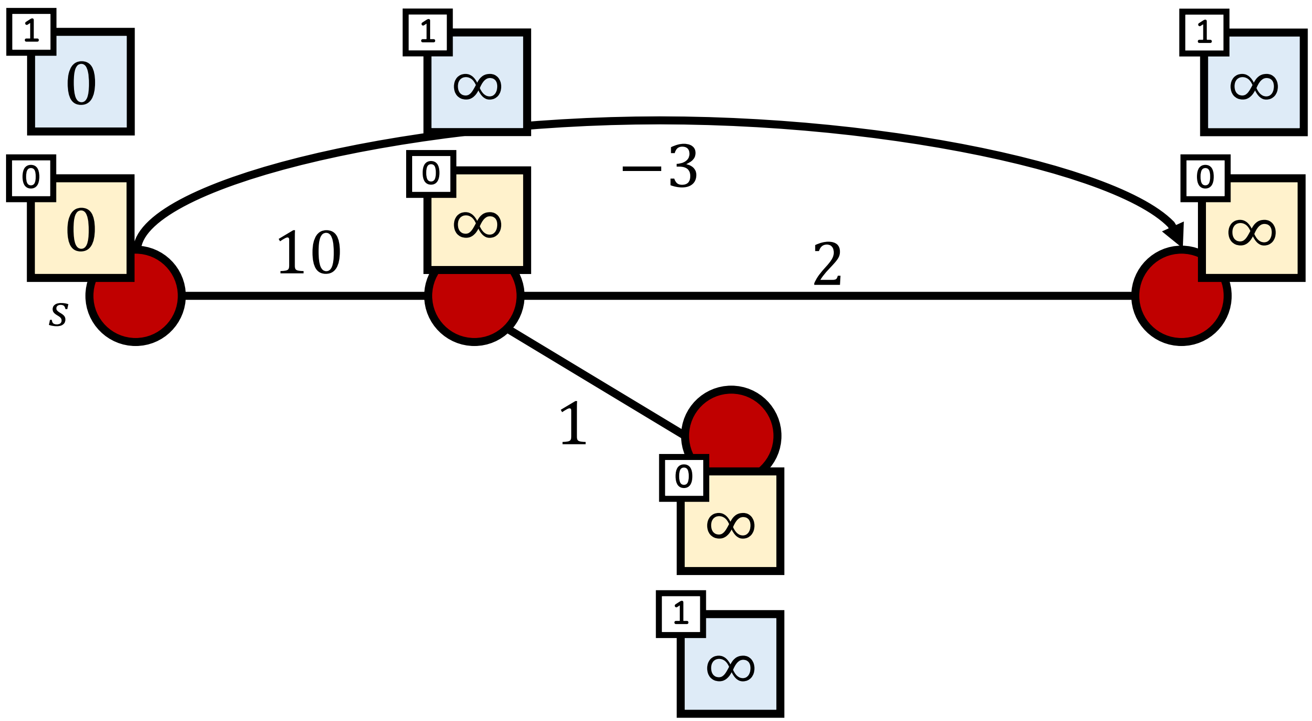
Undirected edge

i $D_i[v]$ Shortest distance of $P_{s \rightsquigarrow v}$ consisting of at most i edges

Example (2)

❑ Step 2. Perform the relaxations of all of edges

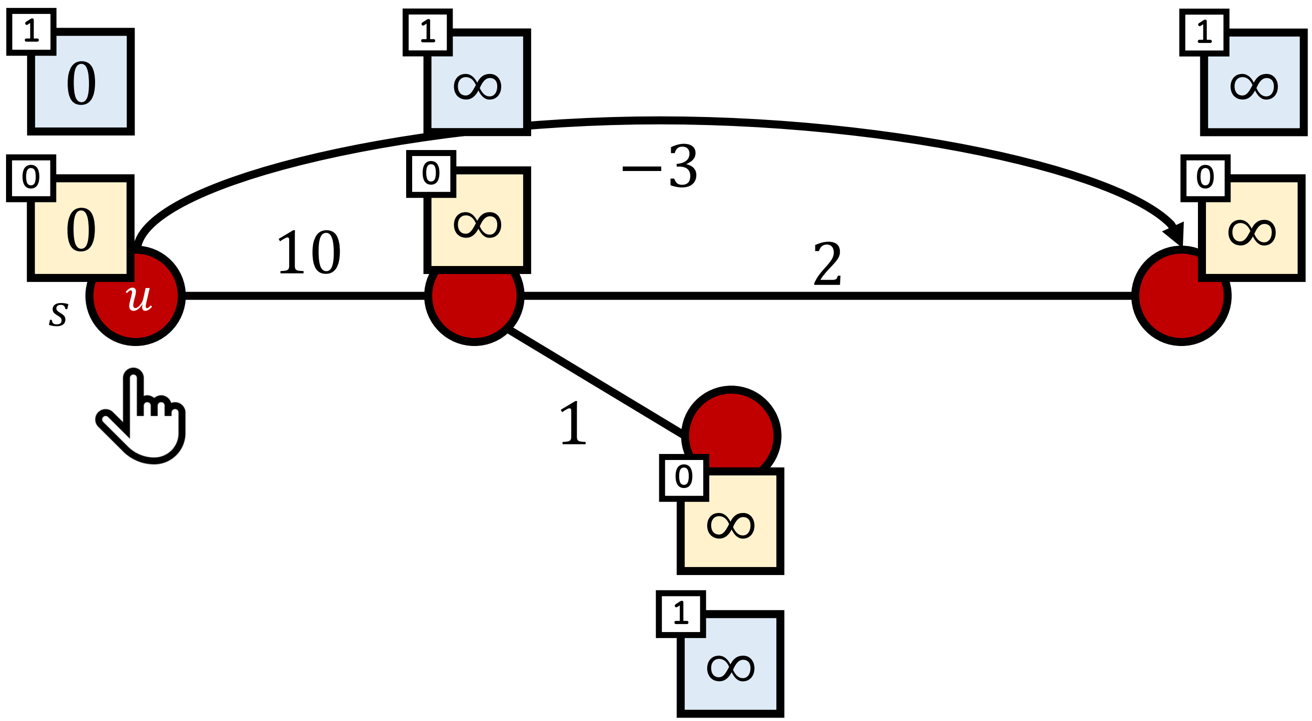
- At the first iteration ($i = 1$ while $n = 4$), prepare $D_i[u]$ for all nodes by copying $D_{i-1}[u]$ for each node u



Example (3)

❑ Step 2. Perform the relaxations of all of edges

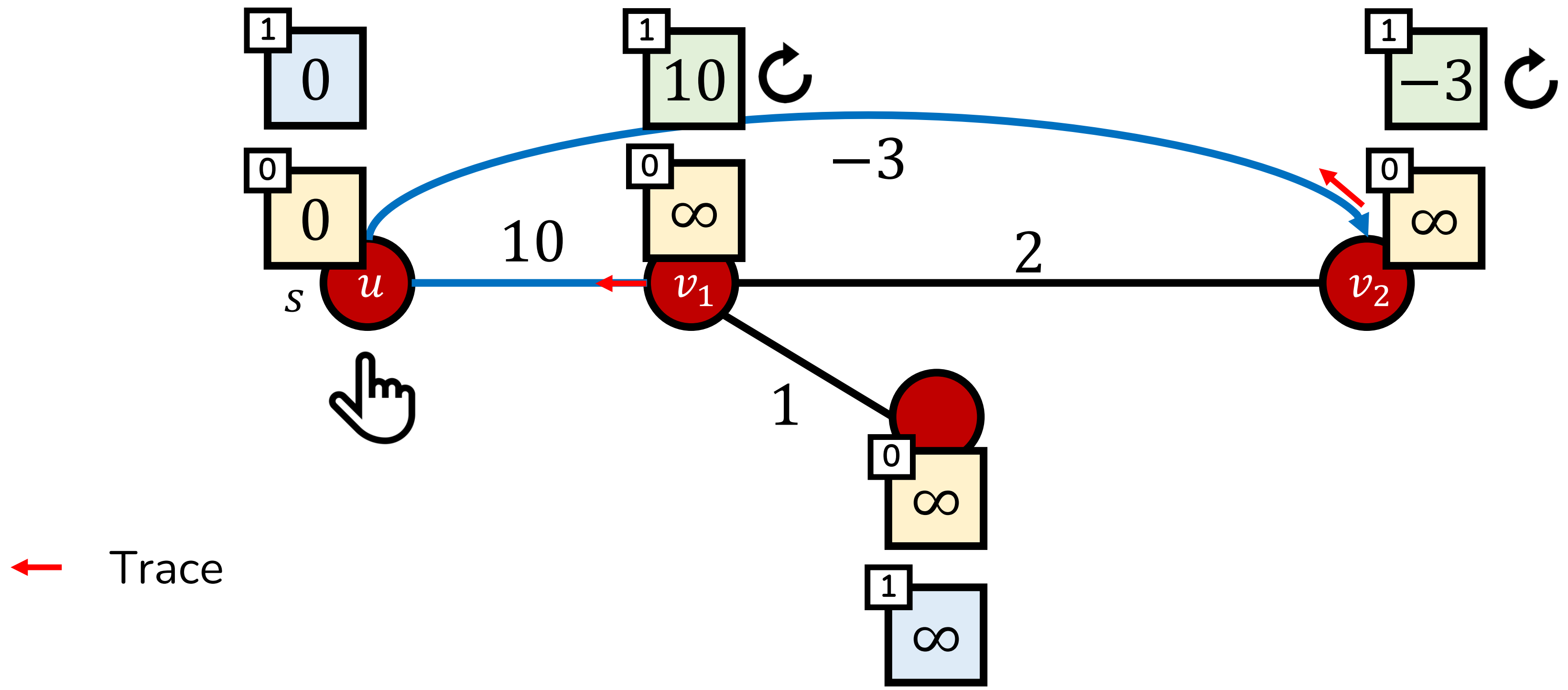
- At the first iteration ($i = 1$ while $n = 4$), select node u whose $D_{i-1}[u]$ is not ∞.



Example (4)

❑ Step 2. Perform the relaxations of all of edges

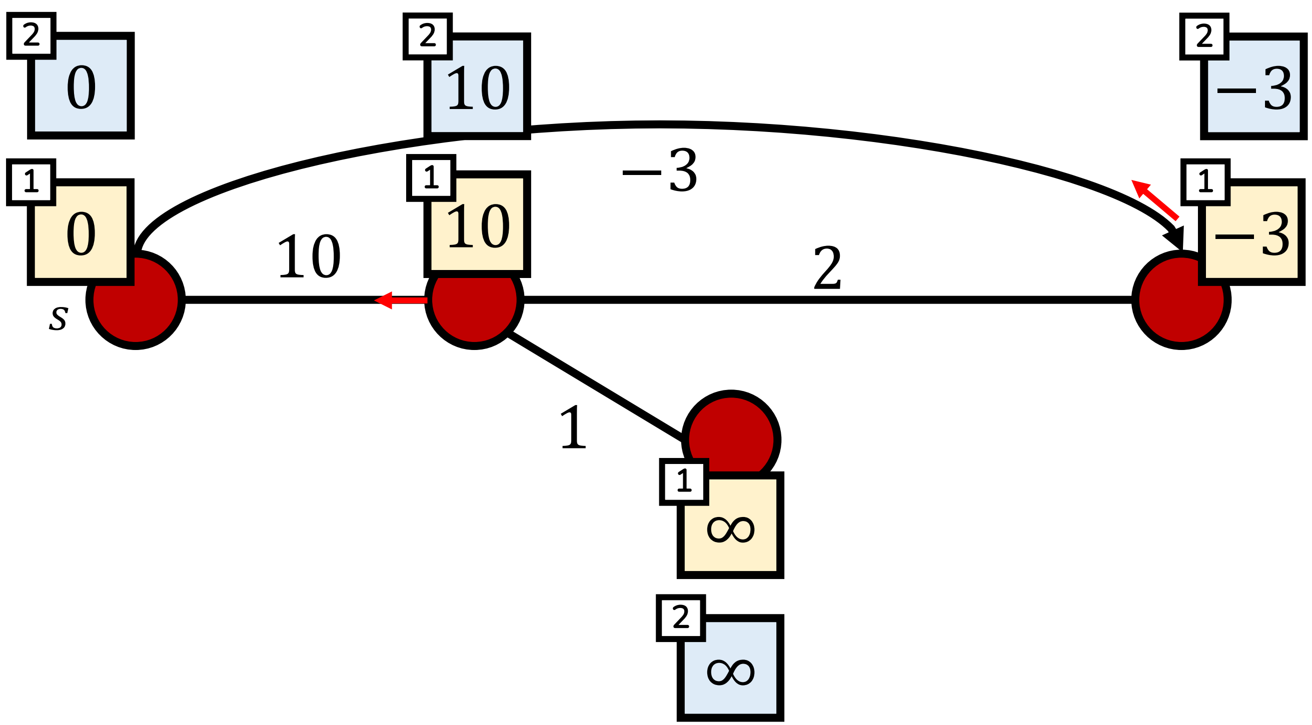
- At the first iteration ($i = 1$), relax neighbor v of node u
  - $D_i[v] = \min\left(D_{i-1}[v], D_{i-1}[u] + w(u,v)\right)$



Example (5)

❑ Step 2. Perform the relaxations of all of edges

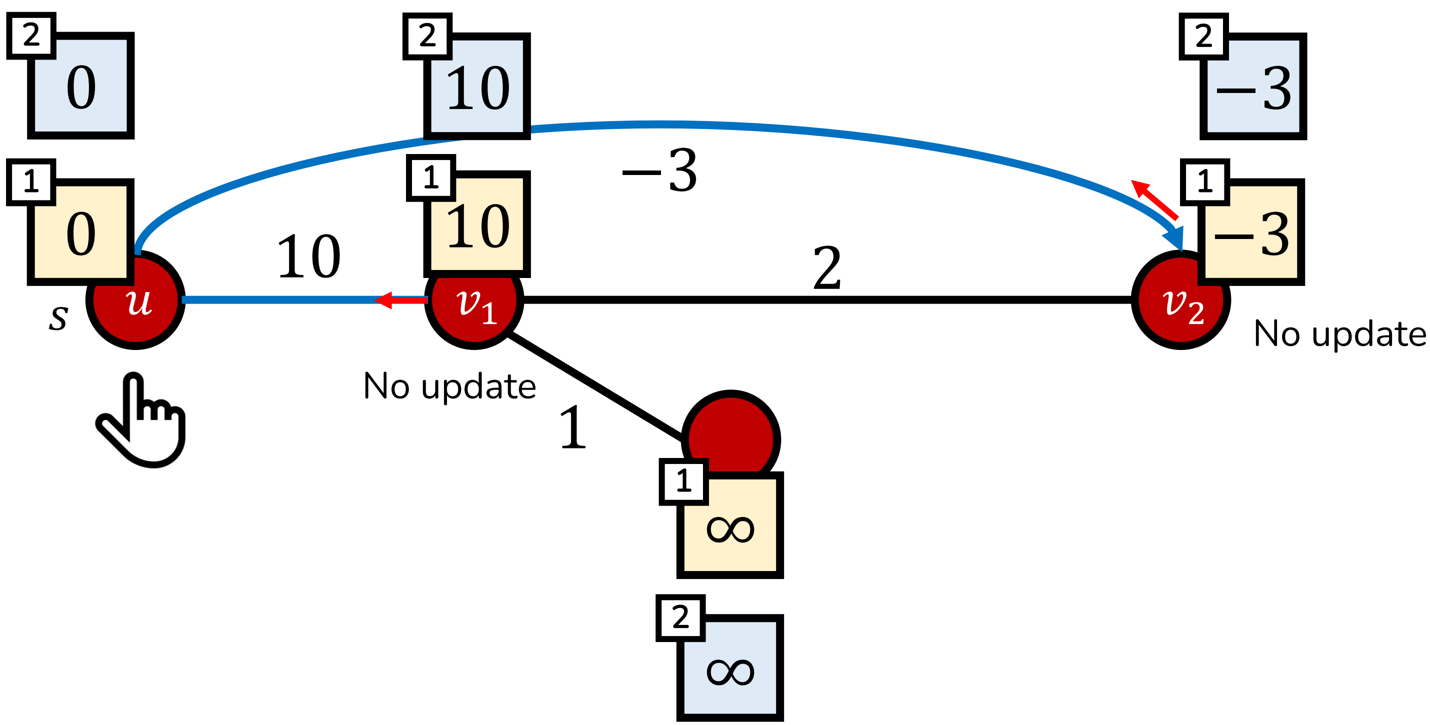
- At the second iteration ($i = 2$ while $n = 4$), prepare $D_i[u]$ for all nodes by copying $D_{i-1}[u]$ for each node u



Example (6)

❑ Step 2. Perform the relaxations of all of edges

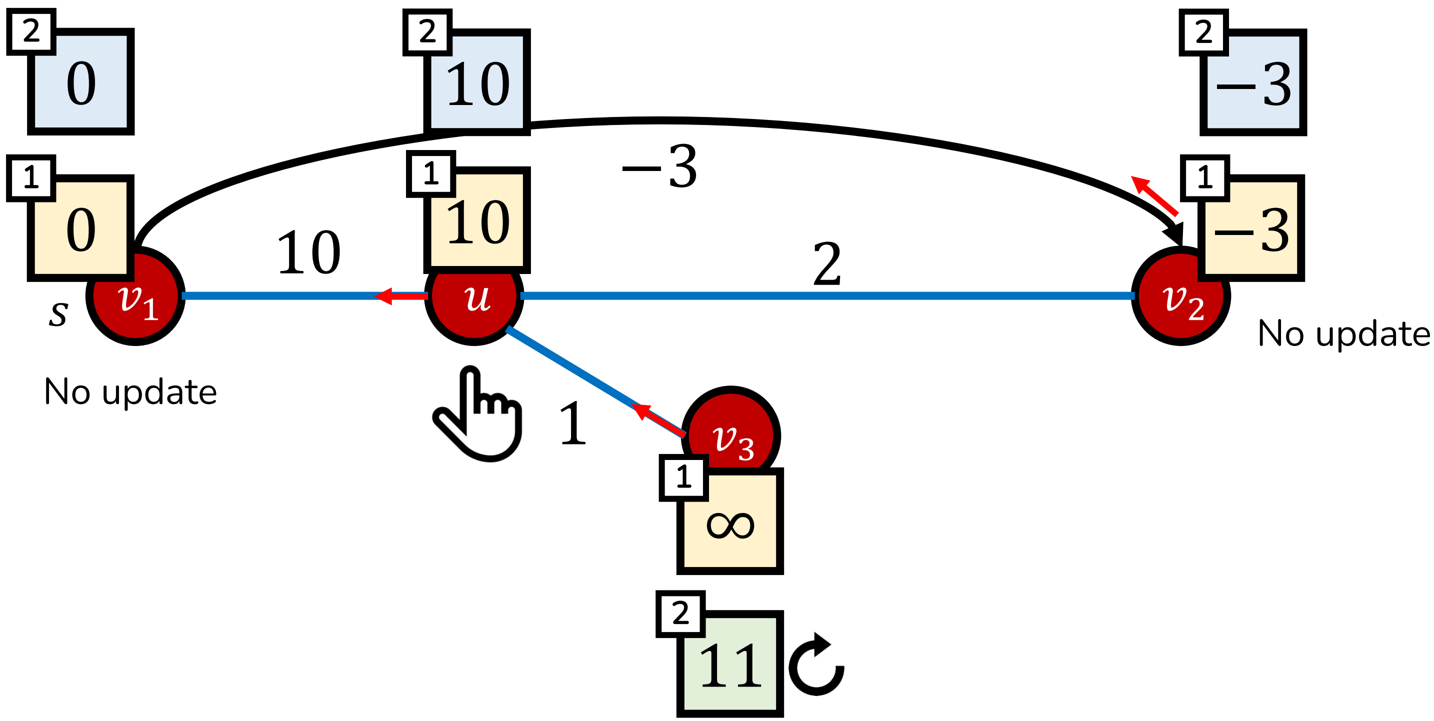
- At the second iteration ($i = 2$), relax neighbor v of node u whose $D_{i-1}[u]$ is not ∞, i.e., $D_i[v] = \min\left(D_{i-1}[v], D_{i-1}[u] + w(u,v)\right)$



Example (7)

❑ Step 2. Perform the relaxations of all of edges

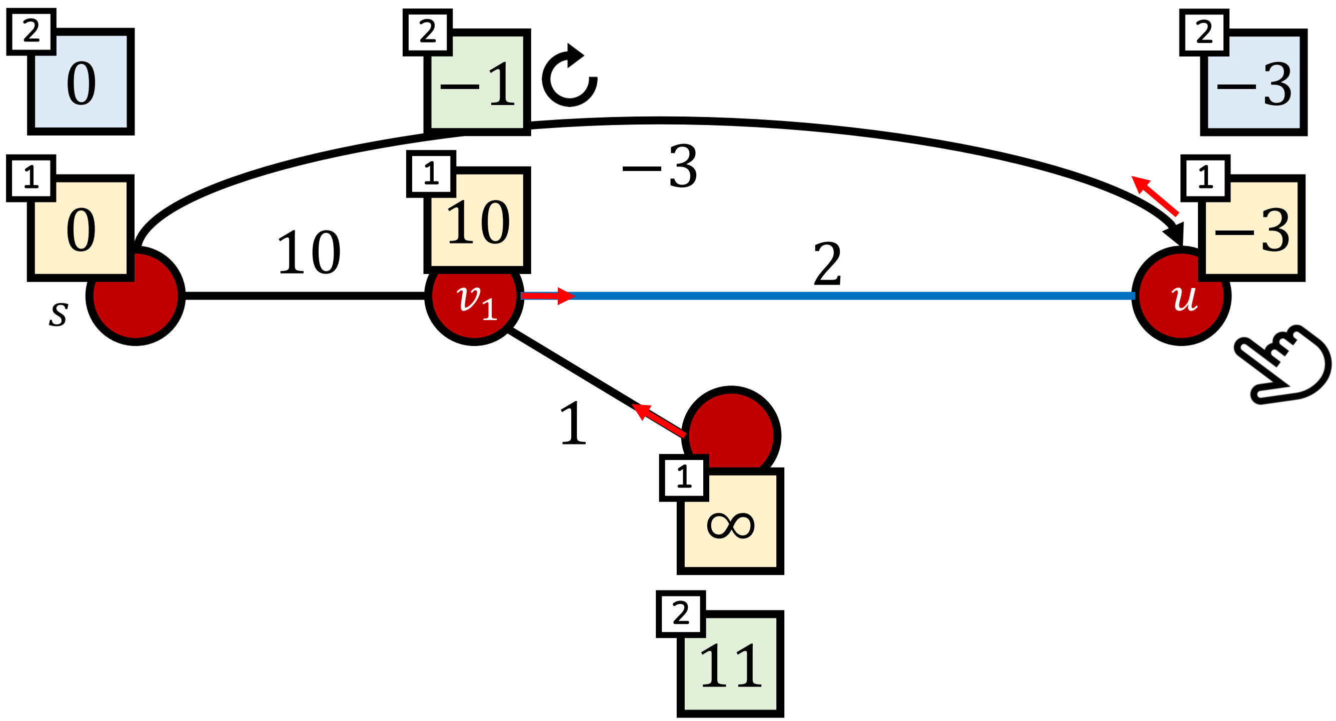
- At the second iteration ($i = 2$), relax neighbor v of node u whose $D_{i-1}[u]$ is not ∞, i.e., $D_i[v] = \min\left(D_{i-1}[v], D_{i-1}[u] + w(u, v)\right)$



Example (8)

❑ Step 2. Perform the relaxations of all of edges

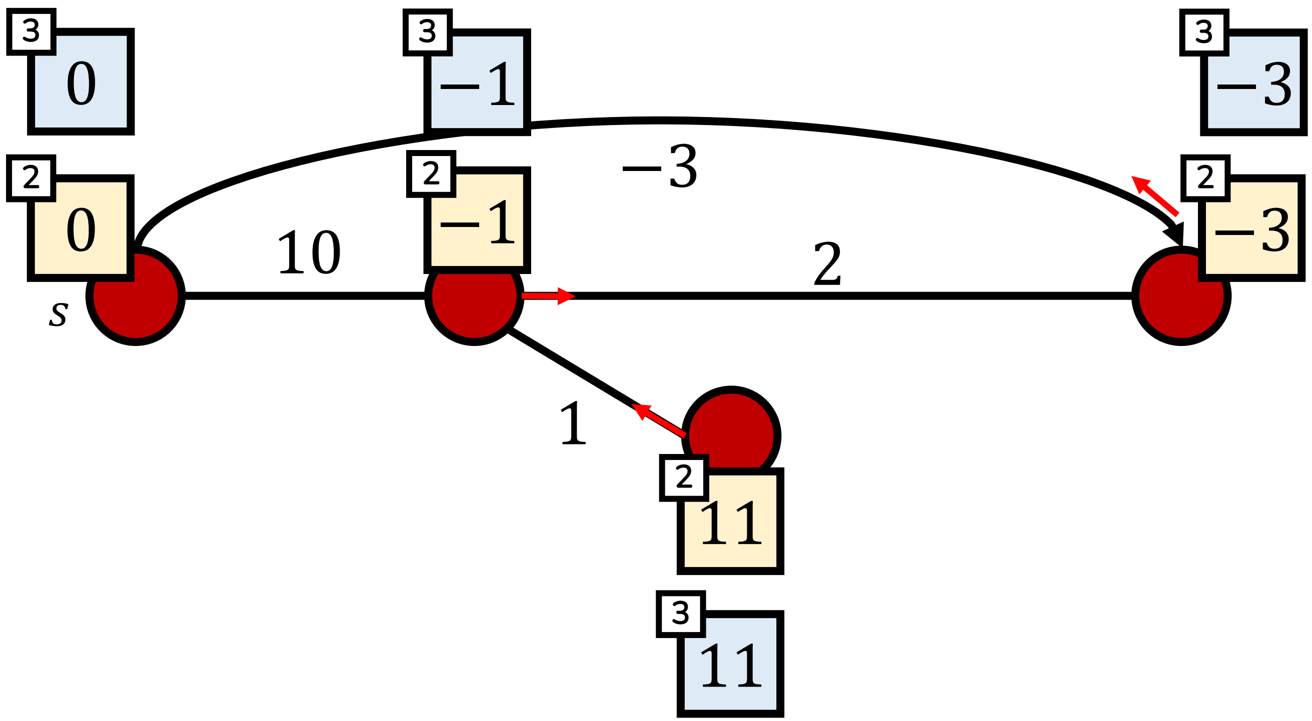
- At the second iteration ($i = 2$), relax neighbor v of node u whose $D_{i-1}[u]$ is not ∞, i.e., $D_i[v] = \min\left(D_{i-1}[v], D_{i-1}[u] + w(u,v)\right)$



Example (9)

❑ Step 2. Perform the relaxations of all of edges

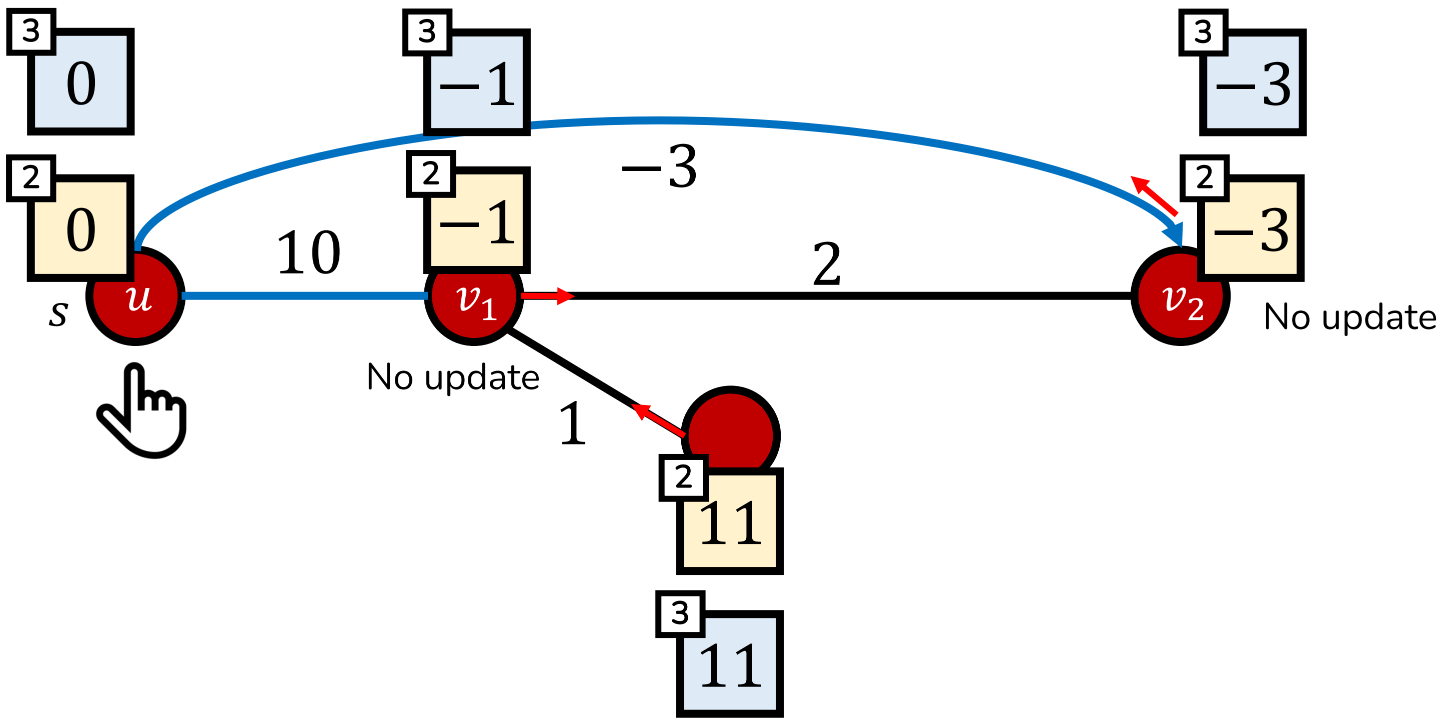
- At the third iteration ($i = 3$ while $n = 4$), prepare $D_i[u]$ for all nodes by copying $D_{i-1}[u]$ for each node u



Example (10)

❑ Step 2. Perform the relaxations of all of edges

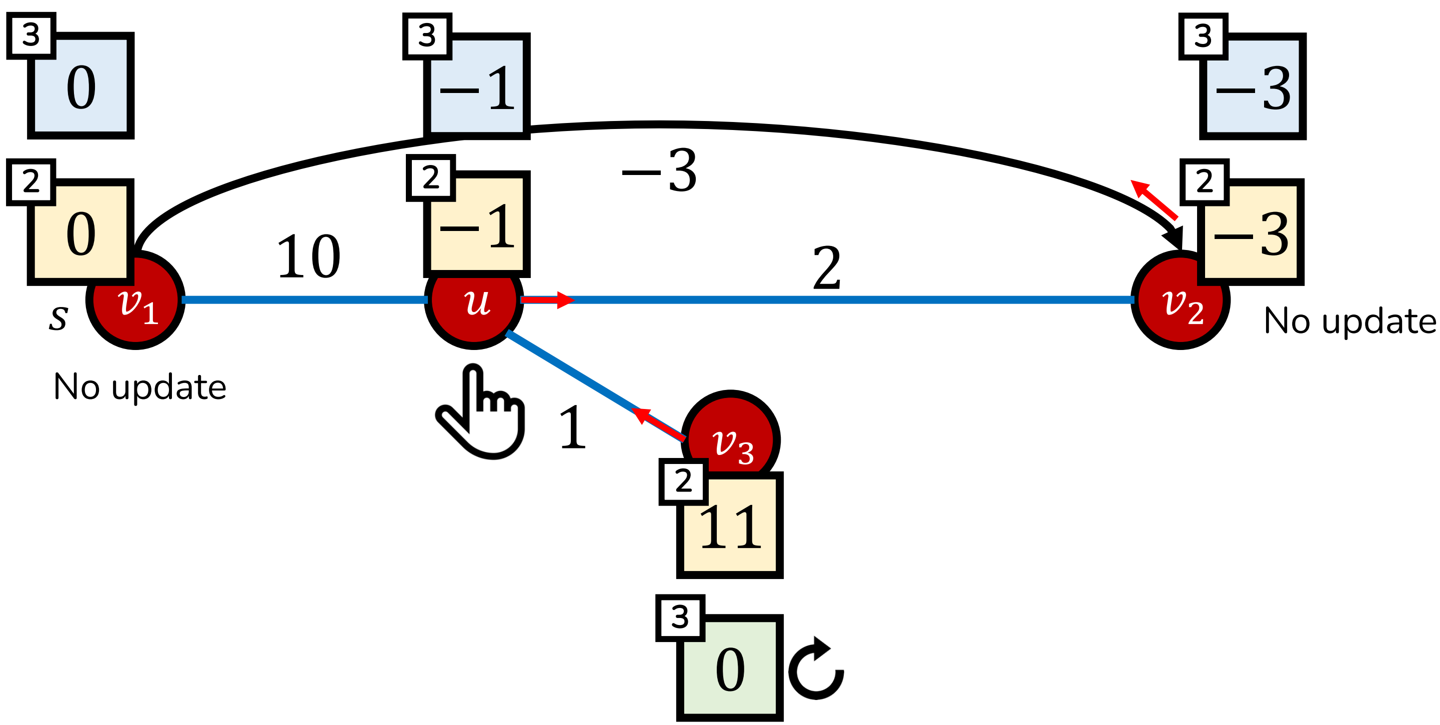
- At the third iteration ($i = 3$), relax neighbor v of node u whose $D_{i-1}[u]$ is not ∞, i.e., $D_i[v] = \min\left(D_{i-1}[v], D_{i-1}[u] + w(u, v)\right)$



Example (11)

❑ Step 2. Perform the relaxations of all of edges

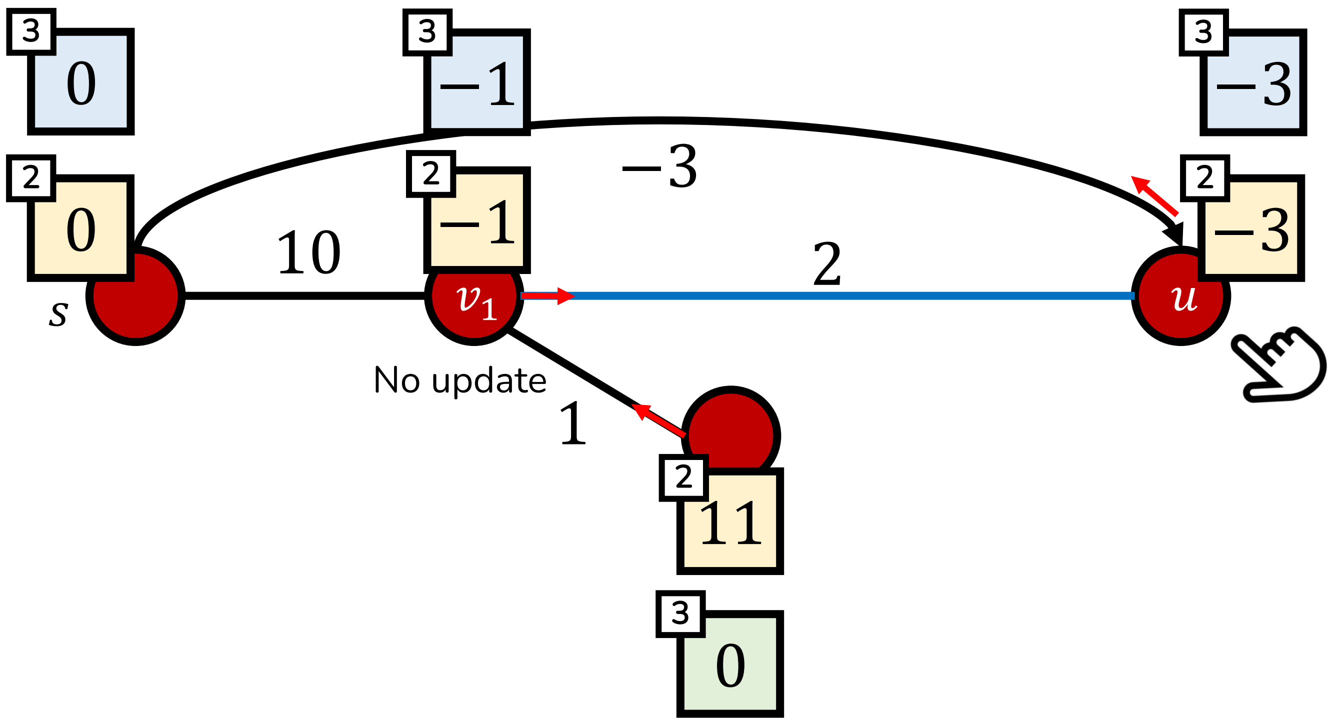
- At the third iteration ($i = 3$), relax neighbor v of node u whose $D_{i-1}[u]$ is not ∞, i.e., $D_i[v] = \min(D_{i-1}[v], D_{i-1}[u] + w(u, v))$



Example (12)

❑ Step 2. **Perform the relaxations of all of edges**

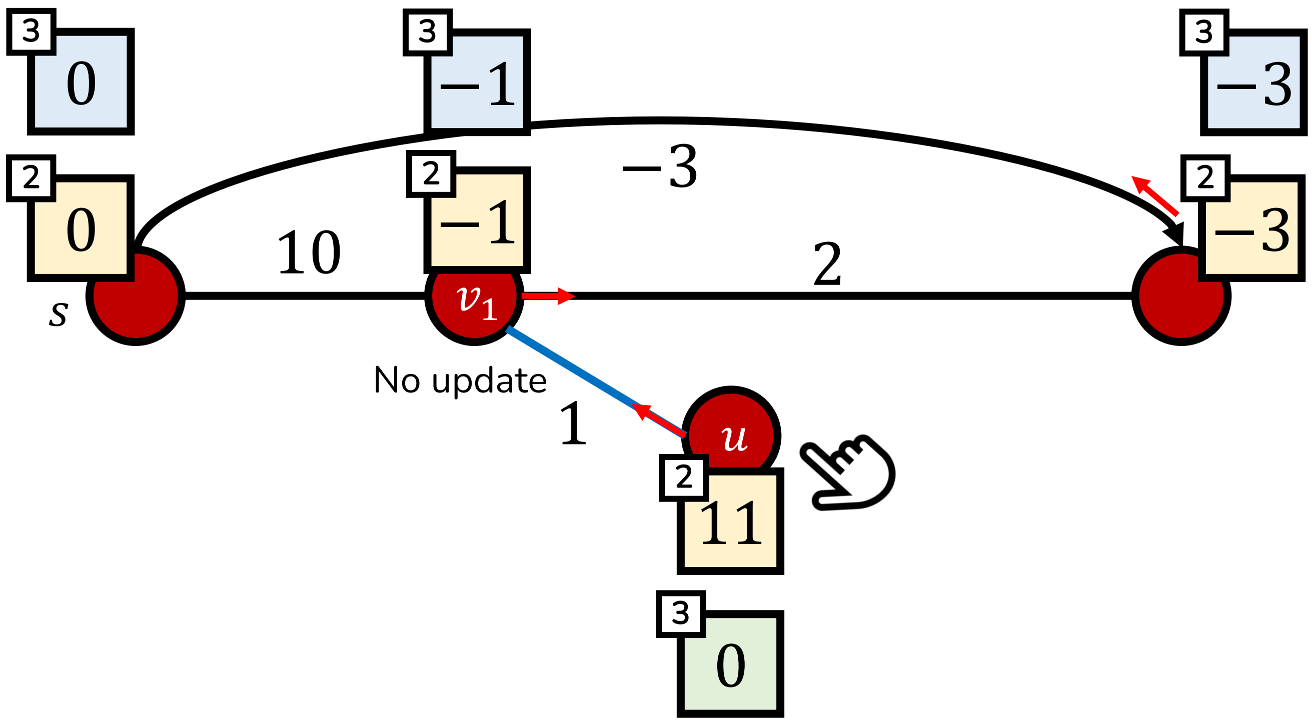
- At the third iteration ($i = 3$), relax neighbor v of node u whose $D_{i-1}[u]$ is not ∞, i.e., $D_i[v] = \min(D_{i-1}[v], D_{i-1}[u] + w(u, v))$



Example (13)

❑ Step 2. Perform the relaxations of all of edges

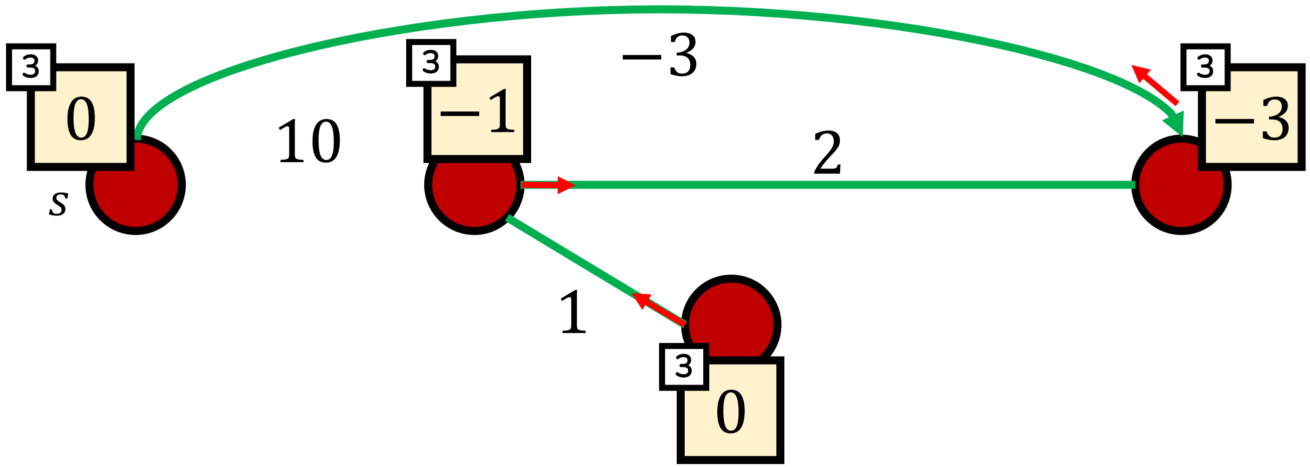
- At the third iteration ($i = 3$), relax neighbor v of node u whose $D_{i-1}[u]$ is not ∞, i.e., $D_i[v] = \min(D_{i-1}[v], D_{i-1}[u] + w(u, v))$

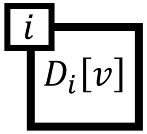


Example (14)

❑ Final shortest path tree with distances

- By Bellman-Ford algorithm





Shortest distance of $P_{s \rightsquigarrow v}$ consisting of at most i edges

Bellman-Ford Algorithm

❑ Pseudocode

```
def bellman-ford(G, s):
    for each v in V:
        D_0[v] ← ∞
    D_0[s] ← 0 & parent[s] ← s

    for i ← 1 to n−1:
        for each u ∈ V:
            if D_{i−1}[u] is not ∞:
                for each v ∈ N_u:
                    if D_{i−1}[u] + w(u,v) < D_{i−1}[v]:
                        D_i[v] ← D_{i−1}[u] + w(u,v)
                        parent[v] ← u

    return D_{n−1} and parent
```

Step 1. Initialization

Step 2. Performs the relaxations on all of edges $n-1$ times

Correctness Analysis

- **Claim.** After i-th iteration of BF, shortest paths consisting of at most i edges are computed.
  - **Base case)** For source node s, $D_0[s] = 0$; the claim holds.
  - **Inductive case)** Let's assume the claim holds for $i = k$.
    - When $i = k + 1$, BF checks $D_k[u] + w(u, v) < D_k[v]$ for each edge (u, v) where $D_k[u]$ is the cost of the shortest path $P_{s \rightsquigarrow u}^{(k)}$ consisting of at most k edges.
    - If (u, v) is relaxable, $D_{k+1}[v]$ is relaxed, leading to $P_{s \rightsquigarrow v}^{(k+1)} = P_{s \rightsquigarrow u}^{(k)} \cup (u, v)$ is the shortest path consisting of at most $k + 1$ edges.
    - Otherwise, $P_{s \rightsquigarrow v}^{(k)}$ is the shortest path of at most $k < k + 1$ edges.
    - Therefore, the claim also holds for $i = k + 1$.
- **Thus, bellman-ford algorithm is correct because**
  - As checked before, each shortest path consists of at most $n - 1$ edges and BF repeats $n - 1$ iterations.

Implementation Details (1)

❑ The previous pseudocode uses $O(n^2)$ space

- Because $D_i[v]$ is represented by 2D-array, i.e., `D[i][v]`
- What if we remove the sub-script i at D?
  - BF also works! Some future relaxations are pre-performed at i-th iteration, but the answer is guaranteed by $n-1$ iterations

```
def bellman-ford(G, s):
    for each v in V:
        D_0[v] ← ∞
    D_0[s] ← 0 & parent[s] ← s

    for i ← 1 to n−1:
        for each u ∈ V:
            if D_{i−1}[u] is not ∞:
                for each v ∈ N_u:
                    if D_{i−1}[u] + w(u,v) < D_{i−1}[v]:
                        D_i[v] ← D_{i−1}[u] + w(u,v)
                        parent[v] ← u
```

➡

```
def bellman-ford(G, s):
    for each v in V:
        D[v] ← ∞
    D[s] ← 0 & parent[s] ← s

    for i ← 1 to n−1:
        for each u ∈ V:
            ~~if D[u] is not ∞:~~
                for each v ∈ N_u:
                    if D[u] + w(u,v) < D[v]:
                        D[v] ← D[u] + w(u,v)
                        parent[v] ← u
```

This is absorbed by the relaxation condition

Implementation Details (2)

❑ The nested-loop is compactly represented as follows:

- Because checking neighbors of all nodes = checking all edges
- Time complexity is $O(nm)$ and space complexity is $O(n+m)$
  - n is # of nodes and m is # of edges

```
def bellman-ford(G, s):
    for each v in V:
        D[v] ← ∞
    D[s] ← 0 & parent[s] ← s

    for i ← 1 to n−1:
        for each u ∈ V:
            for each v ∈ N_u:
                if D[u] + w(u,v) < D[v]:
                    D[v] ← D[u] + w(u,v)
                    parent[v] ← u

    return D and parent
```

➡

```
def bellman-ford(G, s):
    for each v in V:
        D[v] ← ∞
    D[s] ← 0 & parent[s] ← s

    for i ← 1 to n−1:
        for each (u,v) ∈ E:
            if D[u] + w(u,v) < D[v]:
                D[v] ← D[u] + w(u,v)
                parent[v] ← u

    return D and parent
```

Negative Cycle Detection (1)

❑ What if the graph having negative cycles is given to Bellman-Ford algorithm?

- Note that Bellman-Ford does not perform relaxations anymore after $(n-1)$-th iteration.
- However, if there is a negative cycle, then this can make other shortest paths having more $n-1$ edges.
- Thus, if there is any relaxation after the last iteration, then it indicates "there is a negative cycle!"

Negative Cycle Detection (2)

❑ Pseudocode with NC detection

```
def bellman-ford(G, s):
    for each v in V:
        D[v] ← ∞
    D[s] ← 0 & parent[s] ← s

    for i ← 1 to n−1:
        for each (u,v) ∈ E:
            if D[u] + w(u,v) < D[v]:
                D[v] ← D[u] + w(u,v)
                parent[v] ← u

    for each edge (u,v) ∈ E:
        if D[u] + w(u,v) < D[v]:
            throw "a negative cycle is detected!"

    return D and parent
```

Step 0. Initialization

Step 1. Performs the relaxations on all of edges $n-1$ times

Negative cycle detection

What You Need To Know

❑ Dijkstra's algorithm (negative weights aren't allowed)

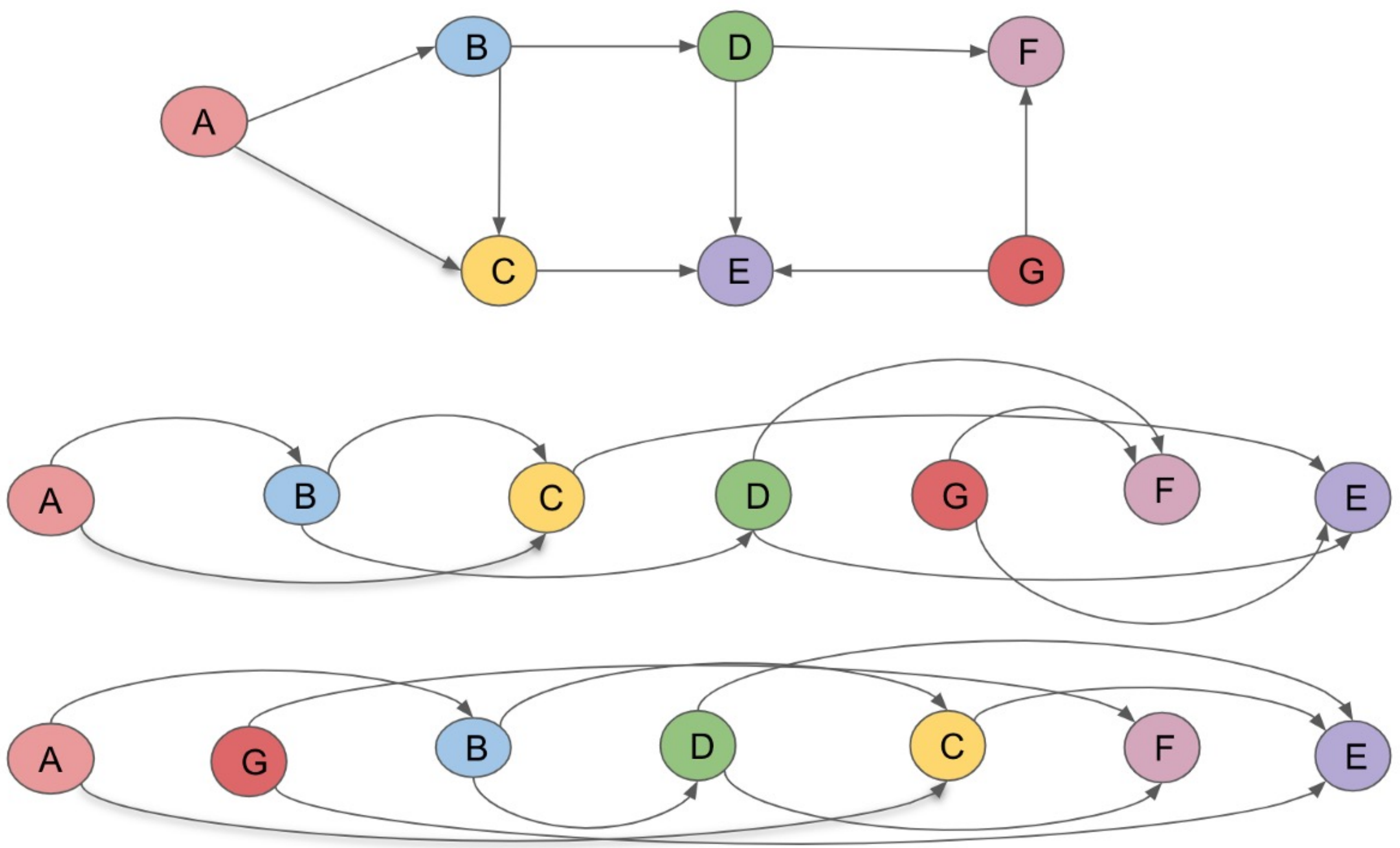
- Incrementally grow shortest paths starting from source node s (i.e., grow the shortest path tree S)
  - Similar to Prim's algorithm
- Time complexity is $O(m \log n)$ using min heap

❑ Bellman-Ford algorithm (negative weights are allowed)

- Repeats relaxations on all of edges $n - 1$ times
  - If there is any relaxation after the last iteration, then it indicates "there is a negative cycle!"
- Time complexity is $O(mn)$

In Next Lecture

- Topological sort on a graph



Topological Sort

Thank You

Appendix: Implmentation Details

❑ If you don't know how to implement decrease-key

- Just add a new item when the relaxation part.

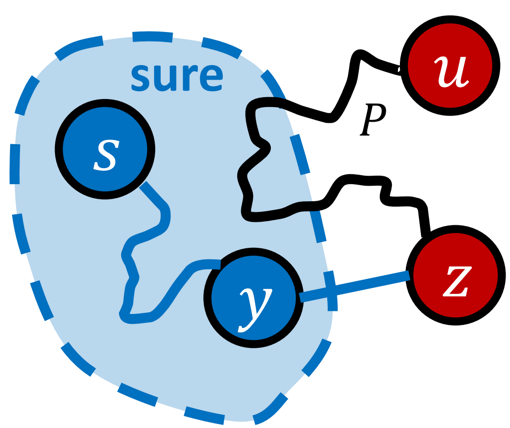
```
def dijkstra(G, s):
    Q ← min-heap()
    for each v in V:
        D[v] ← ∞ & Q.insert(D[v], v) & inSST[v] ← false
    D[s] ← 0 & parent[s] ← s & Q.insert(D[s], s)

    while Q is not empty:
        u ← Q.remove()
        if inSST[u] is true : continue
        inSST[u] ← true
        for each v in N_u:
            if inSST[v] is false and D[u] + w(u,v) < D[v]:
                D[v] ← D[u] + w(u,v)
                Q.insert(D[v], v)
                parent[v] ← u
```

Appendix: Dijkstra's Algorithm

- **Claim.** When a node u is added to "sure" set by DA, its estimate is equal to the true shortest distance, i.e., $D[u] = \delta(s, u)$ **[Proof by contradiction]**
  - Suppose the statement is false
  - $\Rightarrow$ A: $D[u] > \delta(s, u)$ when u is added into "sure" set
    - P: the (real) shortest path from s to u
    - z: first node which is not in "sure" set, and is on the path P
    - y: predecessor of z on the path P

General situation when DA is about to add node u into "**sure**" set

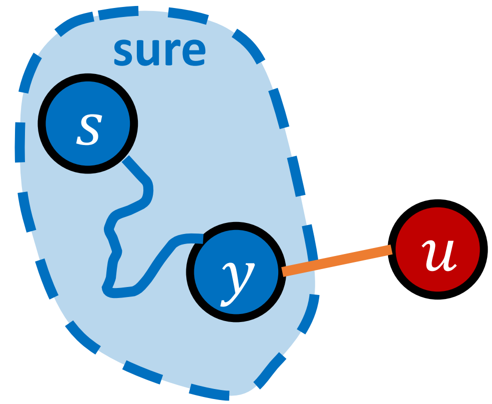


Two cases are possible

- C1) $z = u$
- C2) $z \neq u$

Appendix: Dijkstra's Algorithm

- Case 1) $z = u$
  - $D[y] = \delta(s, y)$
    - y is already in the set before u is added
  - $D[u] \leq D[y] + w(y, u) = \delta(s, y) + w(y, u)$
    - u has been *relaxed* by y
  - Note that **a sub-path in a shortest path is a shortest path**
    - $\mathrm{P}: s \rightsquigarrow u = \mathrm{P}: s \rightsquigarrow y \cup y \rightarrow u$
  - Hence, $\delta(s, y) + w(y, u) = \delta(s, u)$
  - i.e., $D[u] \leq \delta(s, u)$, contradicts to A: $D[u] > \delta(s, u)$



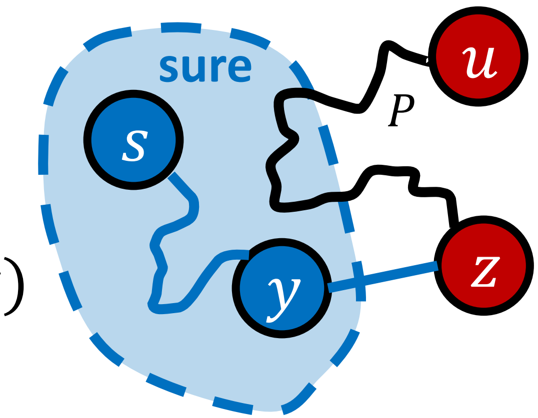
Appendix: Dijkstra's Algorithm

- Case 2) $z \neq u$
  - E1. $D[y] = \delta(s, y)$
    - y is already in the set before u is added
  - E2. $D[z] \leq D[y] + w(y, z) = \delta(s, y) + w(y, z)$
    - z has been *relaxed* by y on the shortest path P
  - E3. $D[u] \leq D[z]$
    - u is selected by DA, i.e., u has the smallest estimate
  - $\Rightarrow D[u] \leq D[z]$ From E3

$$\leq \delta(s, y) + w(y, z)$$ From E2

$$\leq \delta(s, y) + w(y, z) + \delta(z, u)$$ Manually adding

$$= \delta(s, u)$$ Shortest by definition



- $\delta(z, u) \geq 0$ since edge weights are non-negative