Lecture #15
Graph Algorithm (2)

Algorithm

JBNU

Jinhong Jung

In This Lecture

❑ Discussions on MST algorithms

- Complexity analysis of Prim's algorithm
- Correctness analysis and other discussions

❑ Single source shortest path

- Dijkstra's algorithm

Outline

- Complexity analysis of Prim's algorithm
- Discussion on MST algorithms
- Correctness analysis of MST algorithms
- Single source shortest path problem
- Dijkstra's algorithm

Time Complexity of Prim's Alg.

- **Space complexity:** $O(n + m)$ space for adj. list and heap
- **Time complexity:** $O(m \log n)$ time
  - m is # of edges and n is # of nodes

```
def prim(G, r):
    Q ← min-heap()
    for each v in V − {r}:
        c[v] ← ∞ & Q.insert(c[v], v)
    c[r] ← 0 & Q.insert(c[r], r)

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

$O(n \log n)$

$O(\log n)$

$O(|N_u| \log n)$

All nodes are checked

$$\sum_{u \in V} |N_u| \log n = \log n \sum_{u \in V} |N_u| = O(m \log n)$$

Outline

- ❑ Complexity analysis of Prim's algorithm
- ❑ **Discussion on MST algorithms**
- ❑ Correctness analysis of MST algorithms
- ❑ Single source shortest path problem
- ❑ Dijkstra's algorithm

Prim v.s. Kruskal

- **If a graph is dense, then Prim's algorithm is better**
  - Prim with binary heap takes $O(m \log n)$ time
    - Can be theoretically optimized with Fibonacci heap to $O(m + n \log n)$
  - Kruskal takes $O(m \log m) = O(m \log n)$ time for sorting edges
- **If a graph is sparse or edges are already sorted, then Kruskal's algorithm is better**
  - For a sparse graph (e.g., $m \simeq n$), both have the same complexity $O(n \log n)$, but disjoint set is more lightweight than binary heap (or Fibonacci heap)
  - For sorted edges, there is no cost for sorting; Kruskal's time complexity becomes $O(n + m)$ in this case.

Outline

- ❑ Complexity analysis of Prim's algorithm
- ❑ Discussion on MST algorithms
- ❑ Correctness analysis of MST algorithms
- ❑ Single source shortest path problem
- ❑ Dijkstra's algorithm

Correctness Analysis of MST

❑ Why are Kruskal's & Prim's algorithms correct?

- Both are greedy algorithms because at each phase,
  - Kruskal **greedily chooses the minimum edge** not producing a cycle,
  - Prim **greedily chooses the minimum edge** crossing two partitions.
- Is each selected edge safe for the optimality of MST?

❑ Flow of proving MST algorithms

- **Greedy choice property** by safe edge theorem
- **Optimal sub-structure property** of MST
- **Generic MST algorithm** based on safe edge
- **Connection** of Kruskal or Prim **to Generic MST**

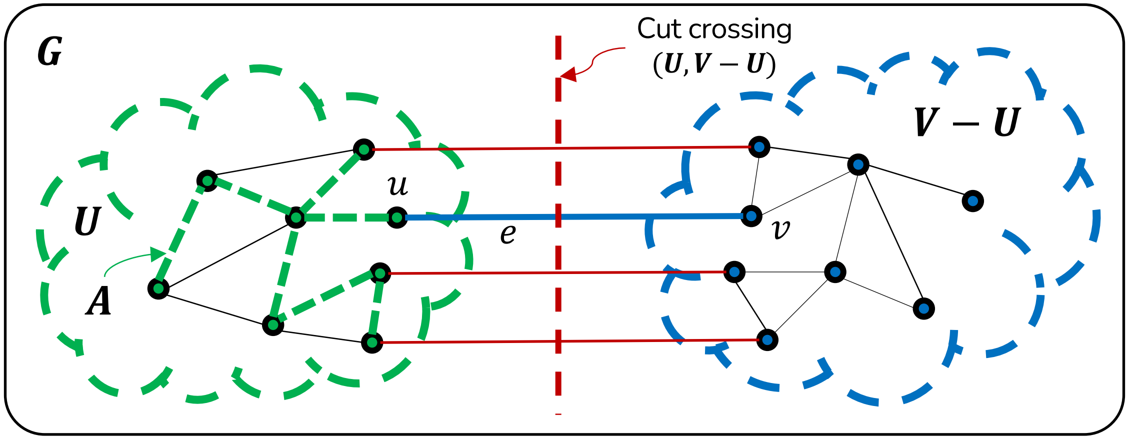
Safe Edge Theorem (1)

❑ Conditions

- $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ is a connected, weighted, and undirected graph.
- Let $\boldsymbol{A}$ be a sub-graph of an MST $\boldsymbol{T}$.
- If $(\boldsymbol{U}, \boldsymbol{V} - \boldsymbol{U})$ is a cut of $\boldsymbol{G}$ w.r.t. $\boldsymbol{A}$, let $e = (u, v)$ be a minimum edge crossing the cut.

❑ Claim: the edge e is safe for $\boldsymbol{A}$

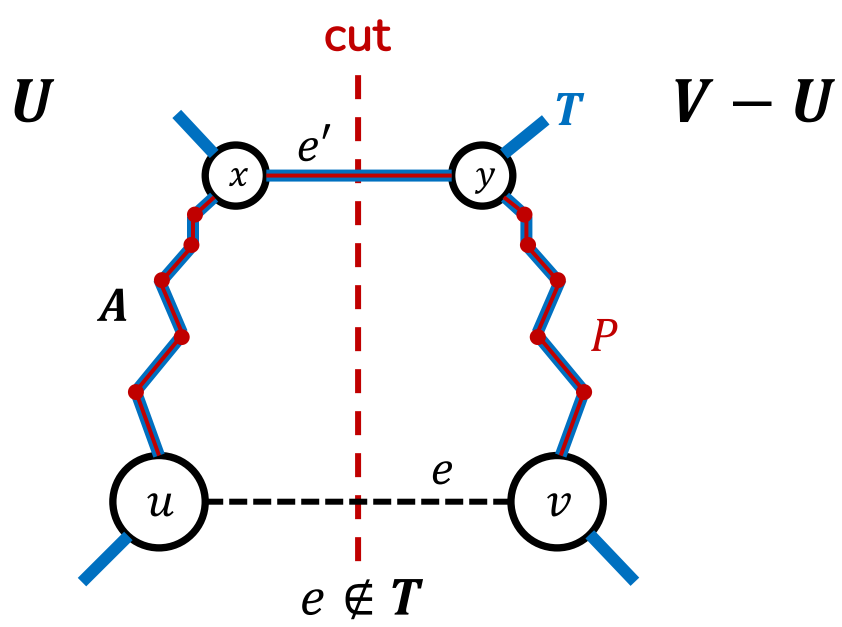
- i.e., e doesn't harm the optimality for constructing MST $\boldsymbol{T}$.



Safe Edge Theorem (2)

❑ Proof by contradiction

- Let's assume the edge e is not in MST $\boldsymbol{T}$.
- Because $\boldsymbol{T}$ is a spanning tree, there should be a path P on $\boldsymbol{T}$ connecting u and v as follows:
  - Note that P is unique because if there are two or more paths between u and v, a cycle forms which contradicts to that $\boldsymbol{T}$ is a tree.
- Let $e' = (x, y)$ be a crossing edge on P.



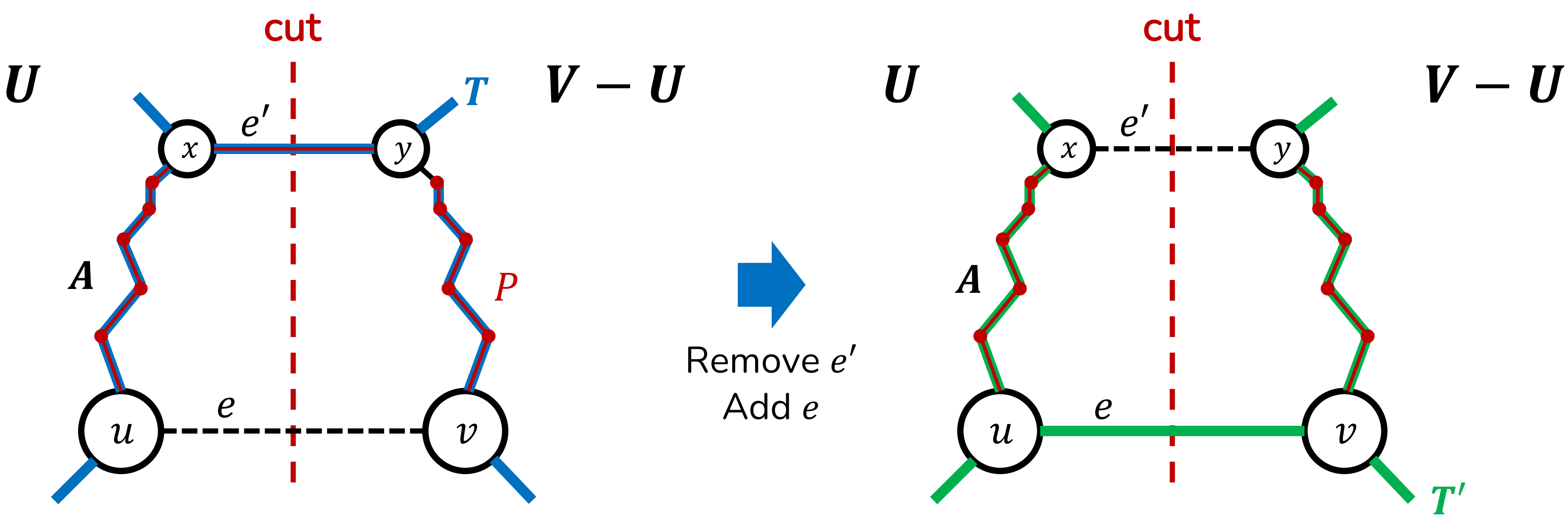
MST $\boldsymbol{T}$: **blue** lines
Path P: **red** lines

Safe Edge Theorem (3)

❑ Proof by contradiction

- Let's remove e' from $\boldsymbol{T}$, and add e; this results in a new tree $\boldsymbol{T'}$ as follows:

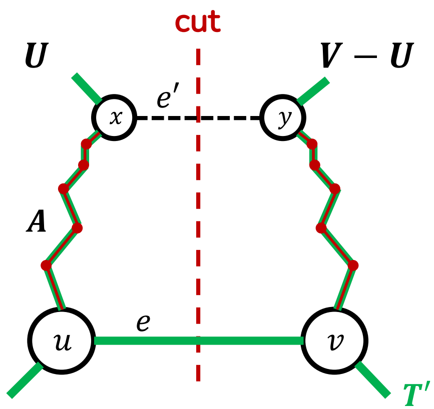
$$\boldsymbol{T'} = \boldsymbol{T} - e' + e$$



Safe Edge Theorem (4)

❑ Proof by contradiction

- Note that $w(e) \leq w(e')$ because e is minimum edge crossing the cut.
- Then, $w(\boldsymbol{T'}) \leq w(\boldsymbol{T})$ due to the following:
    - $w(e)$ = edge's weight, & $w(T)$ = the cost of a tree T



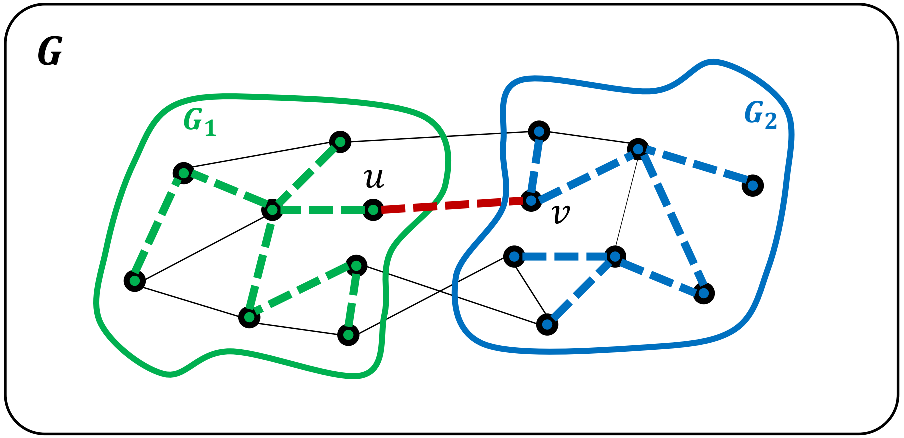
$$w(\boldsymbol{T'}) = w(\boldsymbol{T}) - w(e') + w(e) \leq w(\boldsymbol{T})$$

- This leads to
    - **Case 1)** If $w(e) = w(e')$, then $w(T') = w(T)$. Because T is minimal, T' is also minimal. $\Rightarrow T$ is replaceable with $T' \Rightarrow$ the edge e is in MST.
    - **Case 2)** If $w(e) < w(e')$, then $w(T') < w(T)$ which contradicts to that T is minimal $\Rightarrow$ the edge e is in MST by contradiction.

- Thus, adding e to $\boldsymbol{A}$ is not damaged for constructing MST, implying that e is safe for $\boldsymbol{A}$.

Optimal Sub-structure of MST

- **Claim:** the optimal (minimum spanning) tree is composed of optimal (MS) sub-trees.
  - Consider an edge $e = (u, v)$ in an MST $\boldsymbol{T}$.
  - Removing e partitions $\boldsymbol{T}$ into $\boldsymbol{T}_1$ and $\boldsymbol{T}_2$, i.e., $w(\boldsymbol{T}) = w(\boldsymbol{T}_1) + w(e) + w(\boldsymbol{T}_2)$
  - Then, $\boldsymbol{T}_1$ and $\boldsymbol{T}_2$ are MSTs of $\boldsymbol{G}_1$ and $\boldsymbol{G}_2$, respectively,
    - Where $\boldsymbol{G}_i$ be an induced sub-graph of nodes in $\boldsymbol{T}_i$
    - Because there cannot be better sub-trees than $\boldsymbol{T}_1$ and $\boldsymbol{T}_2$, otherwise $\boldsymbol{T}$ would be sub-optimal.



Prob. in G $= (u, v) +$ Sub-prob. in G_1 $+$ Sub-prob. in G_2

Optimal sub-structure of MST

Generic MST Algorithm

Pseudocode

```
def Generic-MST(G):
    Graph A ← ∅
    while A is not a spanning tree:
        find a safe edge e for A
        add e to A
    return A
```

Claim: A has $n-1$ safe edges forming an MST of G.

- (Spanning tree) See the appendix.
- (Minimality) Greedily selecting a safe edge is safe for the optimality by the Safe edge theorem. By inductively selecting a safe edge of each sub-problem, the selected safe edges don't harm the optimality.

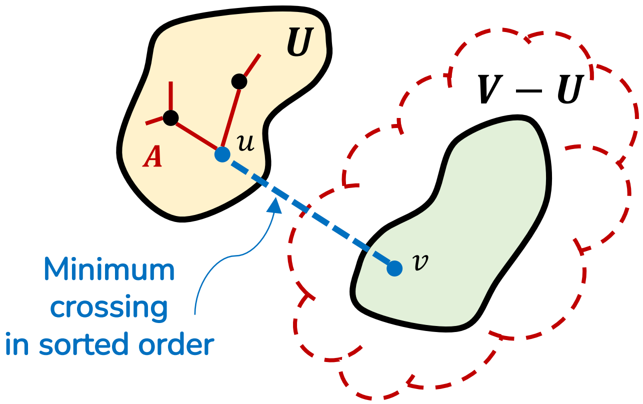
$$\boxed{\text{Prob. in } G} = (u,v) + \boxed{\text{Sub-prob. in } G_1} + \boxed{\text{Sub-prob. in } G_2}$$

Kruskal's Alg. To Generic MST

❑ Connection of Kruskal's alg. to generic MST

```
def kruskal(G):
    T ← list()  ⇒ A

    for each edge (u,v) ∈ sorted(E):
        if (u,v) doesn't form a cycle in T:
            add (u,v) to T  ⇒ safe edge for A
```

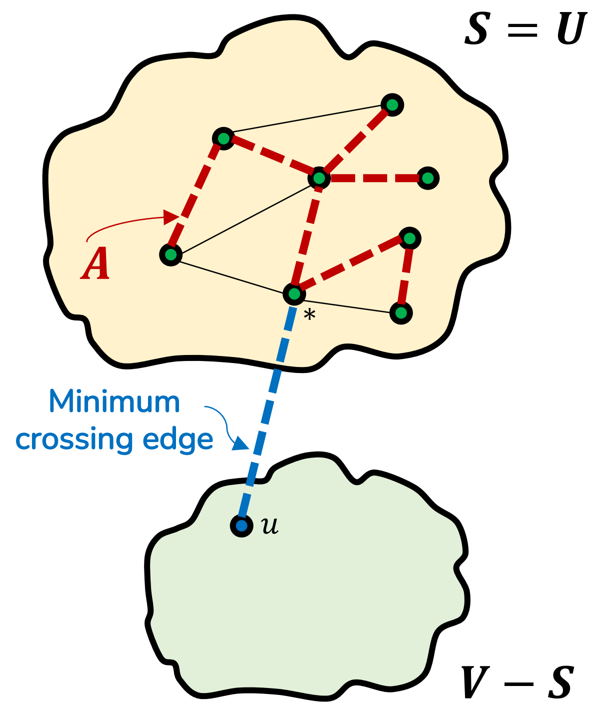


- For (u, v), let $u \in \boldsymbol{U}$ (a comp.) and $v \in \boldsymbol{V} - \boldsymbol{U}$ (another comp.)
- Let $\boldsymbol{A}$ be a tree in $\boldsymbol{U}$; then, $e = (u, v)$ is the minimum edge crossing the cut $(\boldsymbol{U}, \boldsymbol{V} - \boldsymbol{U})$, implying it's safe for $\boldsymbol{A}$.
- $\boldsymbol{T}$ has $n - 1$ safe edges with the same reason as Generic MST.
- In conclusion, Kruskal's algorithm is correct by Generic MST.

Prim's Alg. To Generic MST

❑ Connection of Prim's alg. to generic MST

```
def prim(G, r):
    S ← ∅   ⇒ A
    ……
    while S is not V:
        u ← extract-min(V − S, c)
        S ← S ∪ {u}  ⇒ safe edge (∗, u) for A where ∗ ← parent[u]
        for each v in N_u:
            if v ∈ V − S and w(u, v) < c[v]:
                c[v] ← w(u, v)
                parent[v] ← u
```



- Let S be U, and A be a tree in U; then, the selected edge is the minimum edge crossing the cut $(S, V - S)$, implying it's safe for A.
- S has $n - 1$ safe edges with the same reason as Generic MST.
- In conclusion, Prim's algorithm is correct by Generic MST.

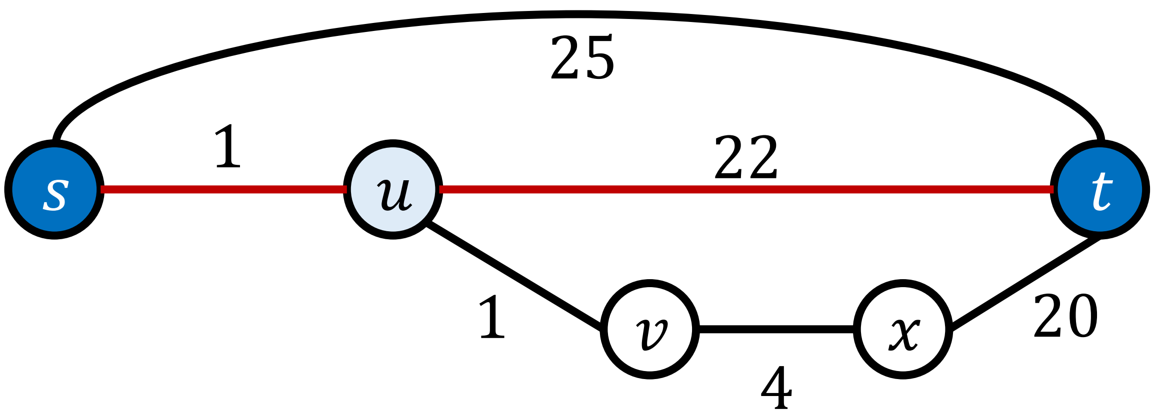
Outline

- ❑ Complexity analysis of Prim's algorithm
- ❑ Discussion on MST algorithms
- ❑ Correctness analysis of MST algorithms
- ❑ **Single source shortest path problem**
- ❑ Dijkstra's algorithm

Shortest Path

❑ Shortest path between two nodes

- Cost of the path = the sum of weights of its edges
- Shortest path has the minimum cost

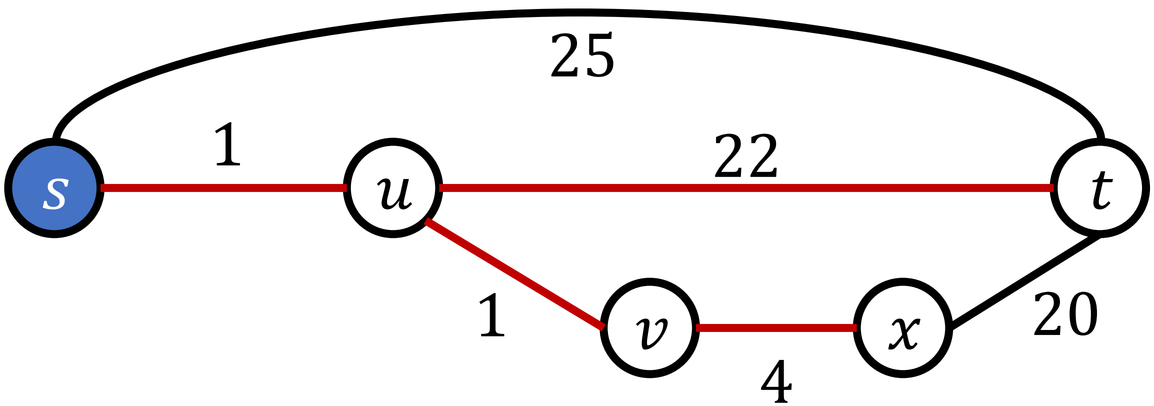


The shortest path from s to t is $s \to u \to t$, and its distance is 23

Single Source Shortest Path

❑ Shortest paths from a single source

- Let's find the shortest paths from a single source node s to all other nodes (input graph G is connected and weighted).
- These paths result in **shortest-path tree** rooted at s

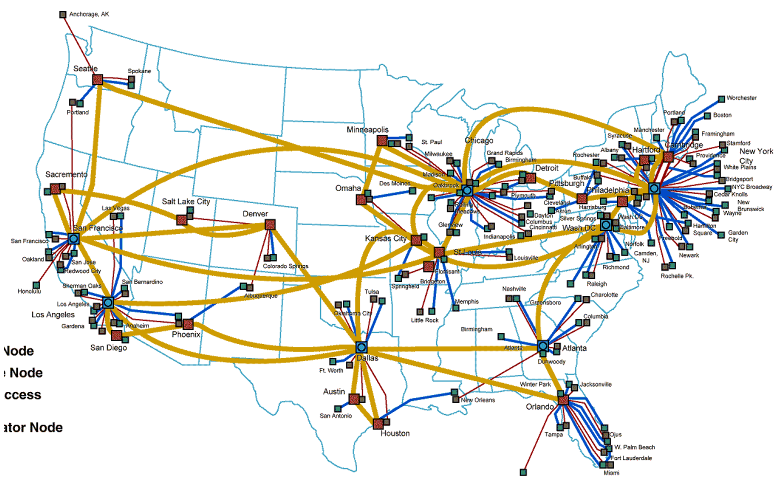


Shortest paths from source node s to all other nodes

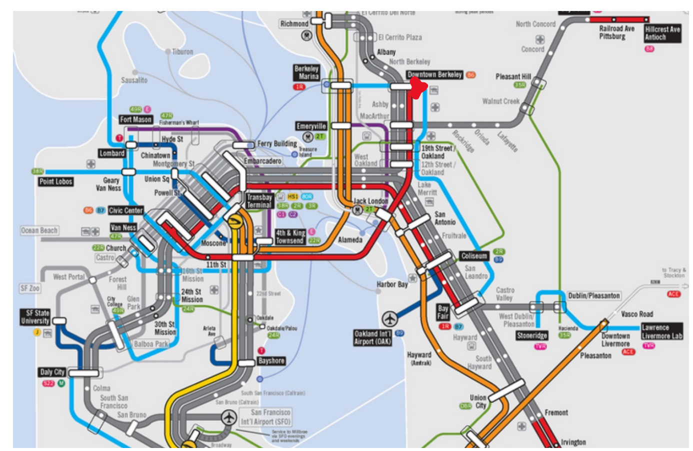
Why Important?

❑ Used in various algorithms & applications on graphs

- **Network routing protocols**
  - Send information over the internet, from my computer to all over the world; how should we send information quickly?
- **Geographic information systems**
  - Find the shortest path between two locations



Internet network



Road network in a map

How To Find Shortest Paths?

❑ Naïve solution

- Enumerate all possible paths (w/o cycle) b.t.w. two nodes.
- If the graph is complete, # of those paths is $O(n!)$.
  - # of paths between two nodes with extra k nodes is $\binom{n-2}{k}$
  - # of orders of k nodes in a path is $k!$
  - # of all possible paths is

Euler's number

$$e = \sum_{j=0}^{\infty} \frac{1}{j!} = 2.718\cdots$$

$$\sum_{k=0}^{n-2} \binom{n-2}{k} k! = \sum_{k=0}^{n-2} \frac{(n-2)!}{(n-2-k)!} = (n-2)! \sum_{j=0}^{n-2} \frac{1}{j!} < (n-2)!\, e \in O(n!)$$

❑ Thus, it's impractical! How to find them quickly?

Outline

- ❑ Complexity analysis of Prim's algorithm
- ❑ Discussion on MST algorithms
- ❑ Correctness analysis of MST algorithms
- ❑ Single source shortest path problem
- ❑ **Dijkstra's algorithm**

Dijkstra's Algorithm

❑ Efficiently find single source shortest paths

- **Input:** a weighted graph G & a source node s
- **Output:** shortest paths from s to all other nodes
  - Result in the **shortest path tree** rooted at s
- **Condition:** G should have **non-negative weights** on edges
  - It could fail if the graph has negative edge weights

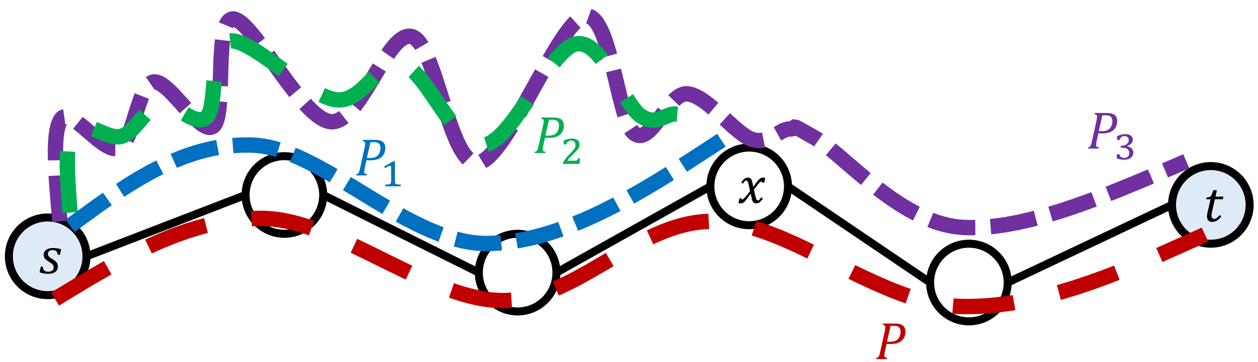
❑ Dijkstra's algorithm

- Performs in $O(m \log n)$ time
  - if it uses min heap
- Proposed by Edsger Dijkstra in 1956
- Similar to Prim's algorithm

Edsger W. Dijkstra

Optimal Sub-structure of SSPS

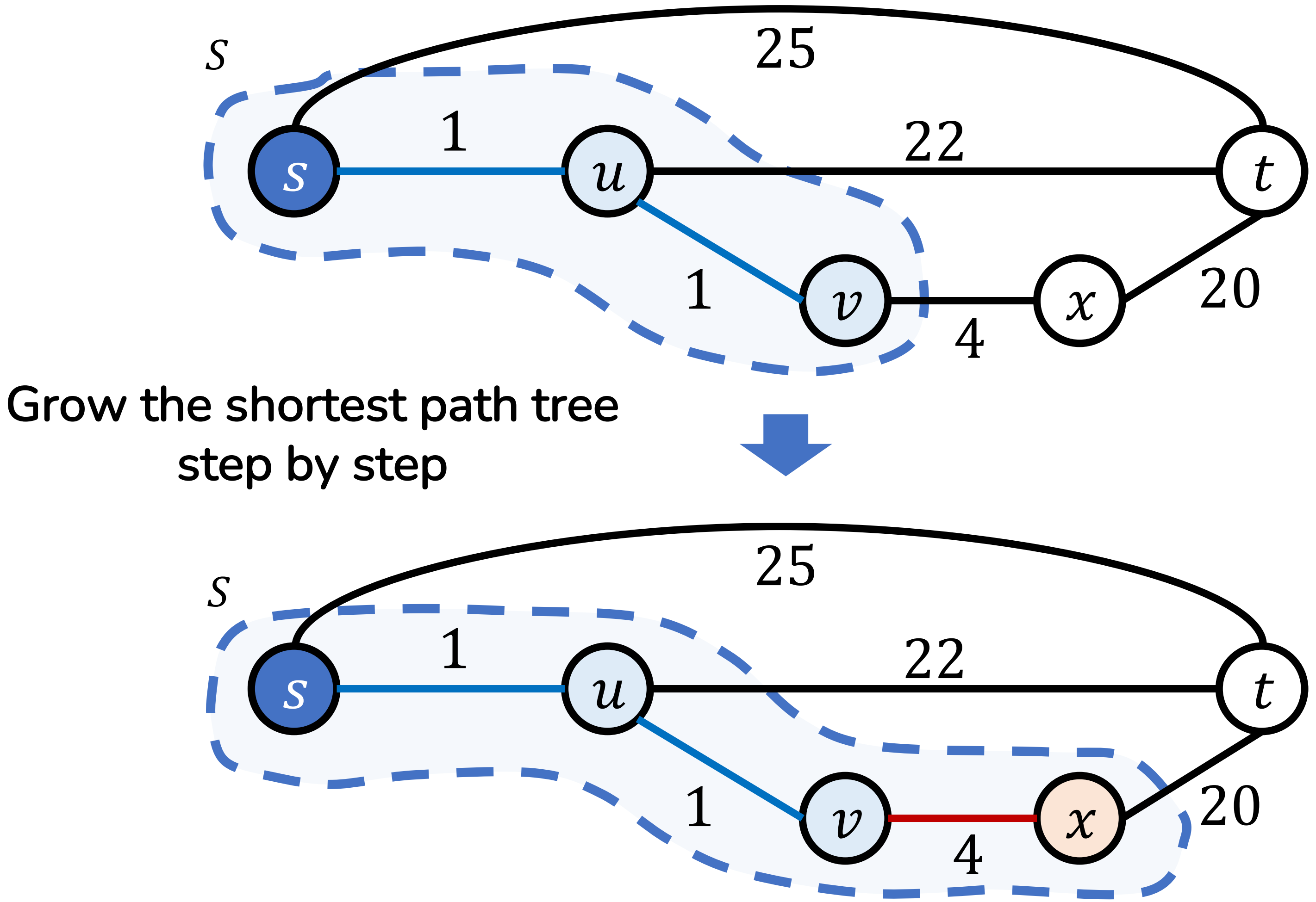
- **A sub-path of a shortest path is also shortest!**
  - Let $P: s \rightsquigarrow t$ be the shortest path from s to t
  - Then, the sub-path $P_1: s \rightsquigarrow x$ in P is shortest
    - Assume P_1 is not shortest $\Rightarrow$ There is another shortest path $P_2: s \rightsquigarrow x$
    - $\Rightarrow P_3: P_2 \cup x \rightsquigarrow t$ is shorter than P, which contradicts to P is shortest
    - Thus, "assume P_1 is not shortest" is false $\Rightarrow P_1$ is shortest



  - Implies that if we know a shortest sub-path, we can find another shortest path built upon the sub-path

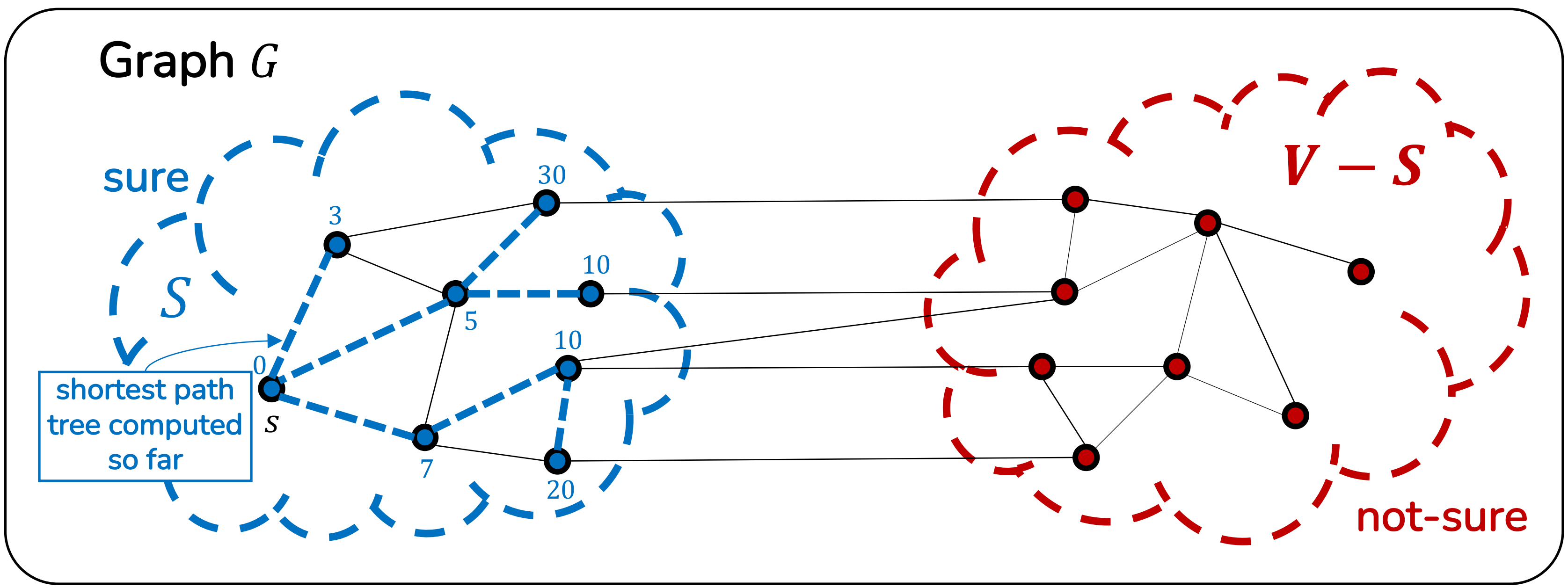
Dijkstra's Intuition

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



Dijkstra's Algorithm (1)

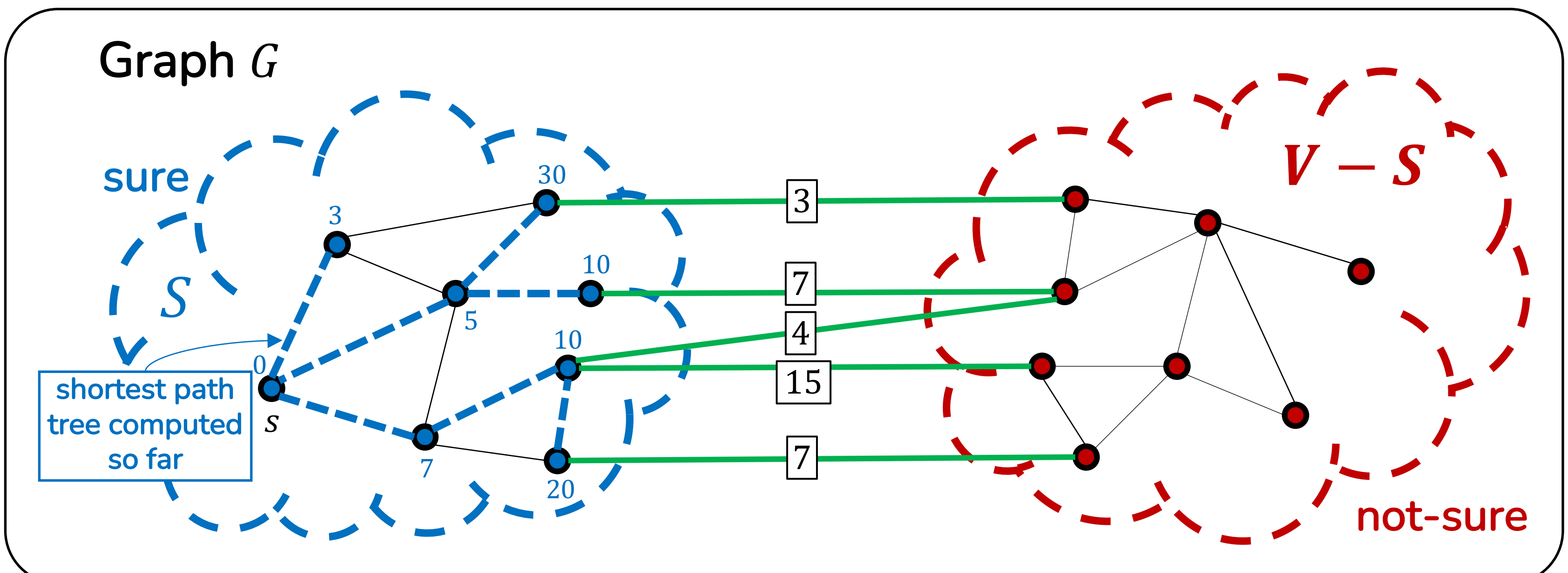
- **Incrementally grow shortest paths starting from source node s** (i.e., grow the shortest path tree S)
  - **Step 1.** maintain two sets $\boldsymbol{S}$ and $\boldsymbol{V-S}$
    - $\boldsymbol{S}$ is a set of nodes consisting of the current shortest path tree
    - $\boldsymbol{V-S}$ is a set of remaining nodes where $\boldsymbol{V}$ is set of nodes in G



Number on a node = the path's cost from s to the node

Dijkstra's Algorithm (2)

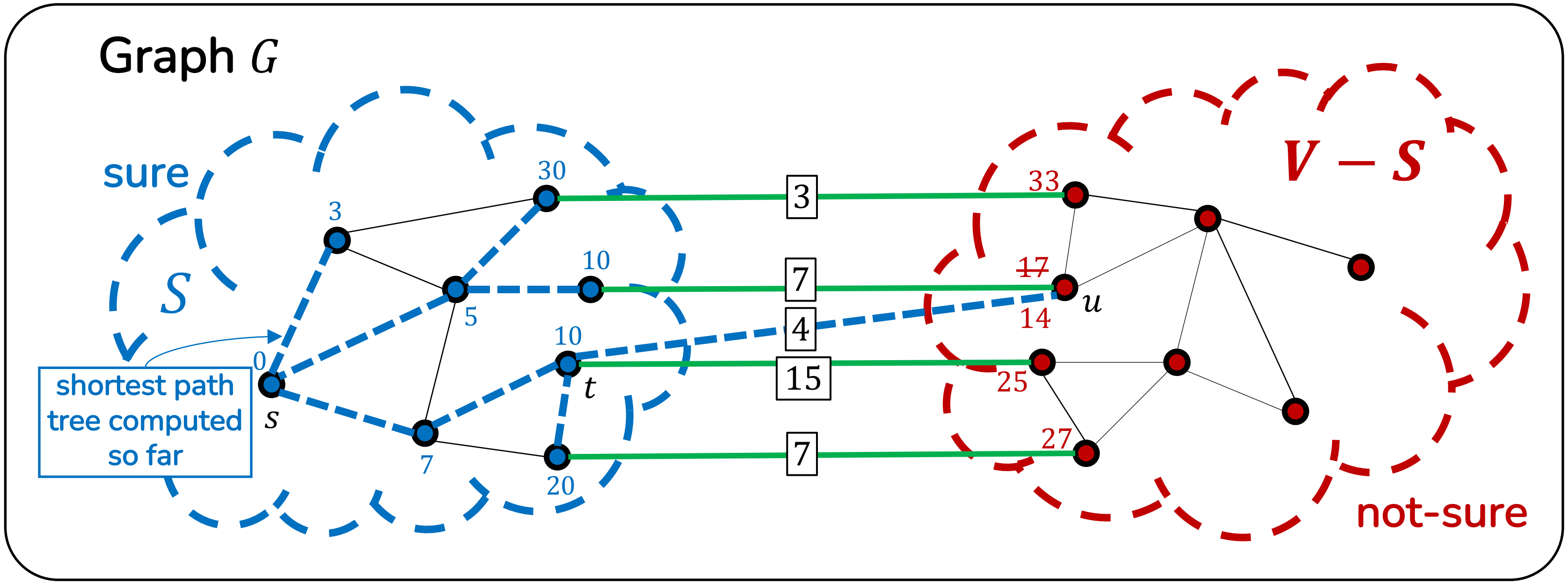
- Incrementally grow shortest paths starting from source node s (i.e., grow the shortest path tree S)
  - **Step 2.** Pick a crossing edge (t, u) from $\boldsymbol{S}$ to $\boldsymbol{V} - \boldsymbol{S}$ such that the distance of path $s \rightsquigarrow t \rightarrow u$ is minimized
    - Green edges are crossing edges between $\boldsymbol{S}$ and $\boldsymbol{V} - \boldsymbol{S}$



Number on a node = the path's cost from s to the node

Dijkstra's Algorithm (3)

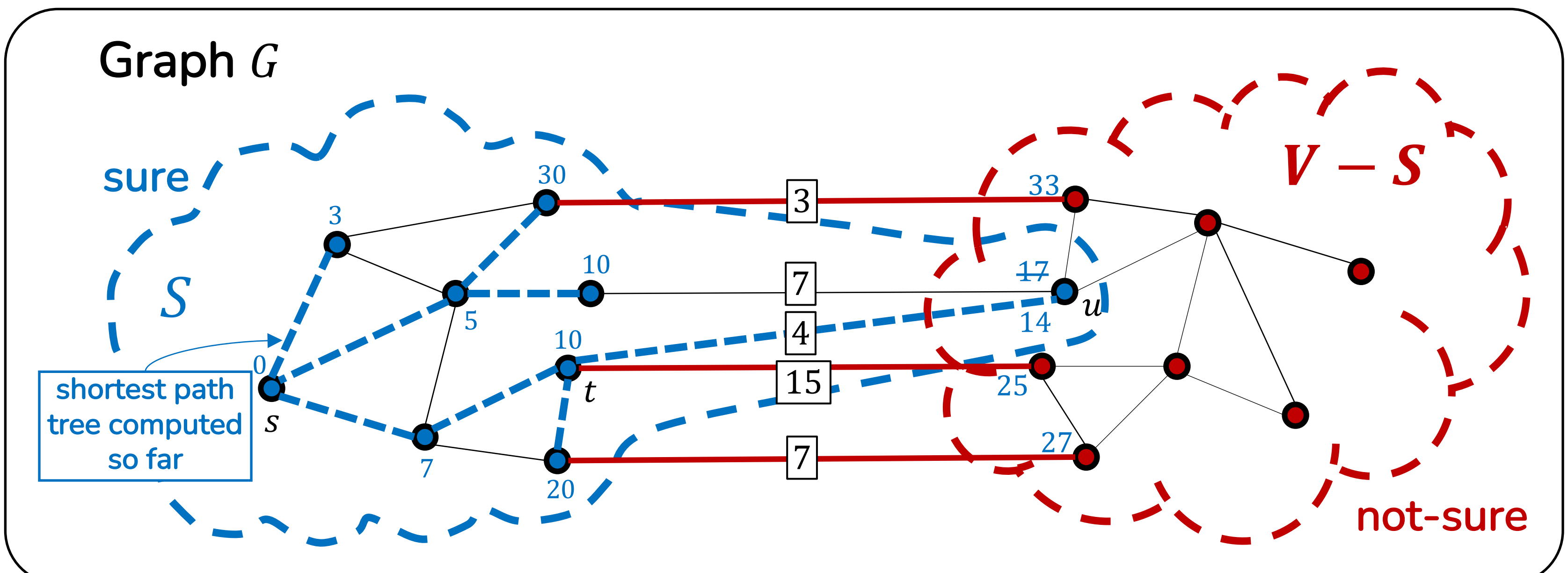
- **Incrementally grow shortest paths starting from source node s** (i.e., grow the shortest path tree S)
  - **Step 2.** Pick a crossing edge (t, u) from $\boldsymbol{S}$ to $\boldsymbol{V} - \boldsymbol{S}$ such that the distance of path $s \rightsquigarrow t \to u$ is minimized
    - Among the crossing edges, **the blue edge** minimizes the distance.



Number on a node = the path's cost from s to the node

Dijkstra's Algorithm (4)

- **Incrementally grow shortest paths starting from source node s** (i.e., grow the shortest path tree S)
  - Step 2 leads to moving the other endpoint of the edge into $\boldsymbol{S}$
    - The shortest path tree is expanded!
  - Repeat Step 2 until S becomes V

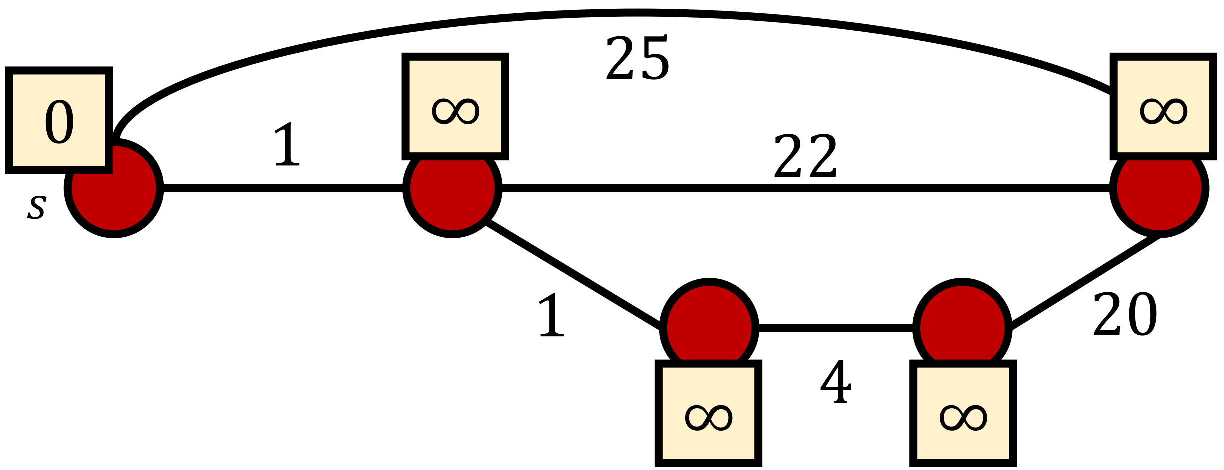


Number on a node = the path's cost from s to the node

Example (1)

❑ Step 1. Maintain two node sets

- Initialize $D[s] = 0$ & $D[v] = \infty$ for all other nodes v
- Mark all nodes as "**not-sure**" (in $V - S$)



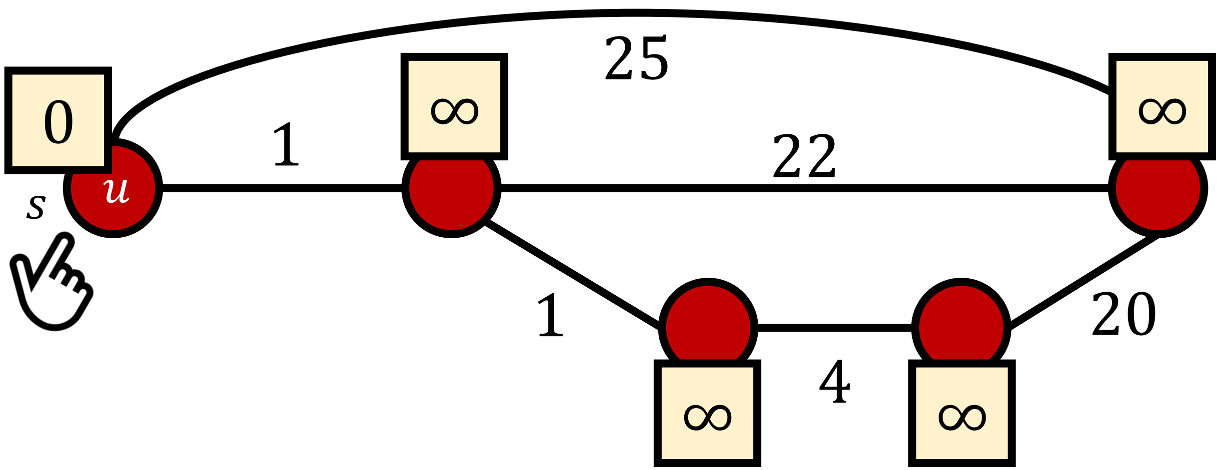
$D[v]$ Estimate of a node

Red: "not-sure"
Blue: "sure"

- If $v \in \boldsymbol{S}$, $D[v]$ is the minimum (true) distance from s to v.
- If $v \in \boldsymbol{V} - \boldsymbol{S}$, $D[v]$ is the smallest (estimate) distance from s to v, **checked so far.**

Example (2)

- **Step 2.** Pick a crossing edge from S to $V - S$
  - **Step 2-1.** Pick a "**not-sure**" node u with the smallest estimate $D[u]$



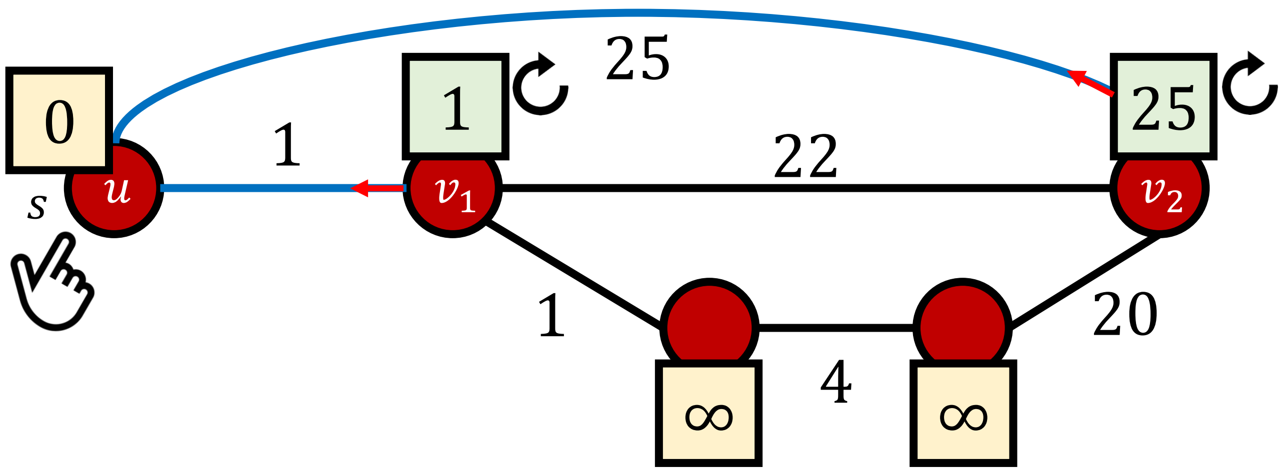
$D[v]$ Estimate of a node

Red: "not-sure"
Blue: "sure"

Example (3)

❑ Step 2. Pick a crossing edge from S to $V - S$

- **Step 2-2.** Update the estimate $D[v]$ to all **not-sure** neighbors v of node u if (u, v) makes a shorter path to v
  - $D[v] = \min\big(D[v], D[u] + w(u, v)\big)$ # called relaxation

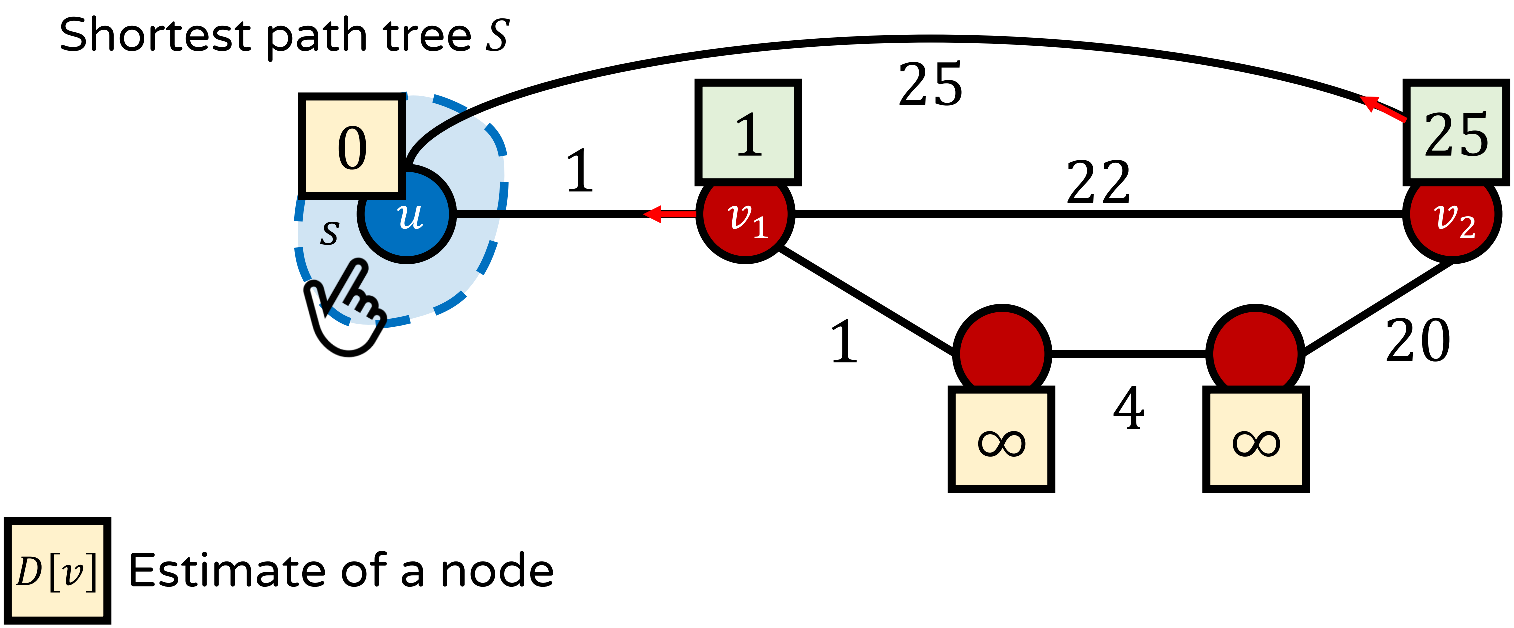


$D[v]$ Estimate of a node

Red: "not-sure"
Blue: "sure"

Example (4)

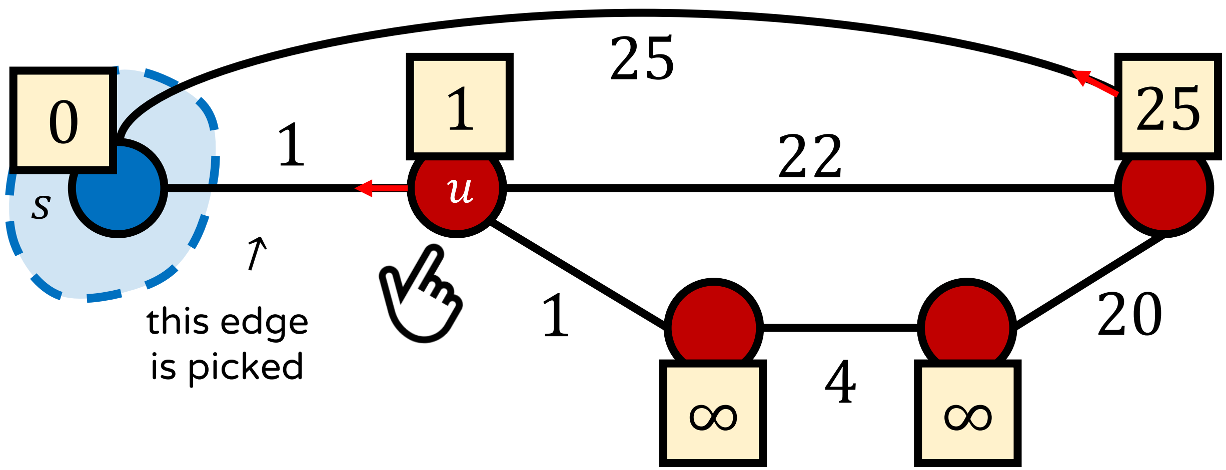
- **Step 2.** Pick a crossing edge from S to $V - S$
  - **Step 2-3.** Move the other node u of the edge into S by marking the selected node u as "**sure**"



Red: "not-sure"
Blue: "sure"

Example (5)

- **Step 2.** Pick a crossing edge from S to $V - S$
  - **Step 2-1.** Pick a "**not-sure**" node u with the smallest estimate $D[u]$

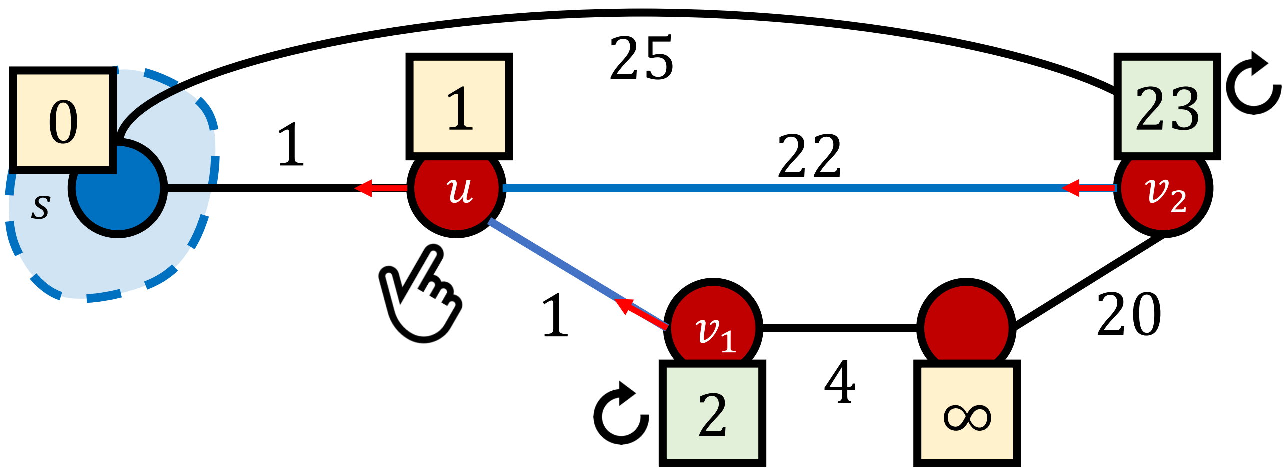


$D[v]$ Estimate of a node

Red: "not-sure"
Blue: "sure"

Example (6)

- **Step 2.** Pick a crossing edge from S to $V - S$
  - **Step 2-2.** Update the estimate $D[v]$ to all **not-sure** neighbors v of node u if (u, v) makes a shorter path to v
    - $D[v] = \min\big(D[v], D[u] + w(u, v)\big)$ # called relaxation

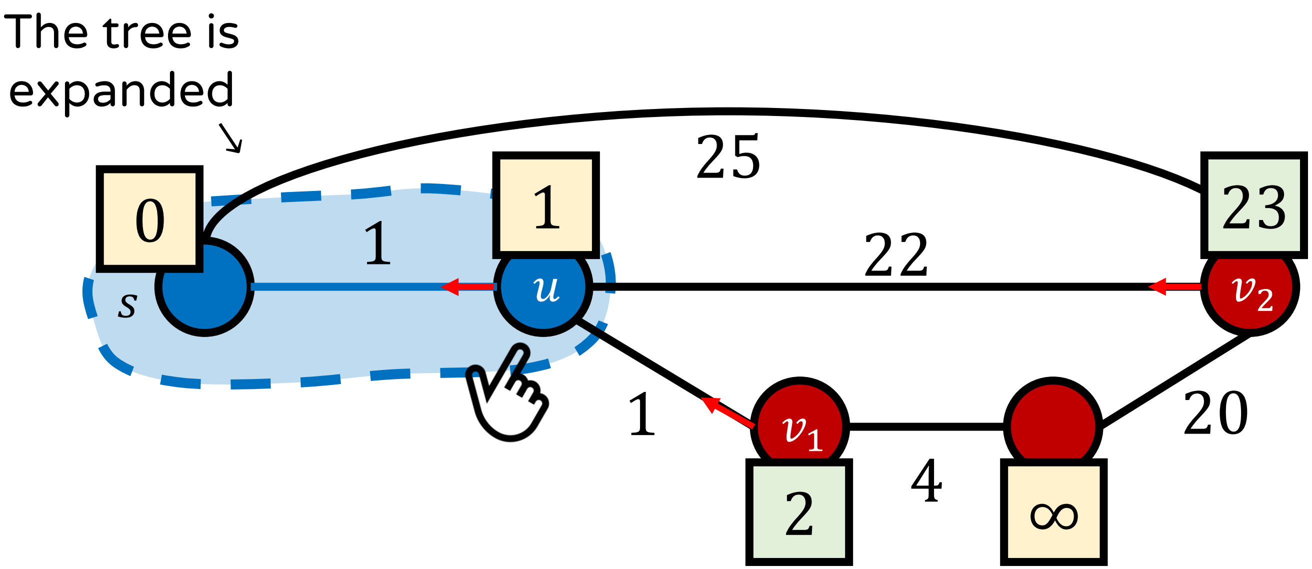


$D[v]$ Estimate of a node

Red: "not-sure"
Blue: "sure"

Example (7)

- **Step 2.** Pick a crossing edge from S to $V - S$
  - **Step 2-3.** Move the other node u of the edge into S by marking the selected node u as "**sure**"

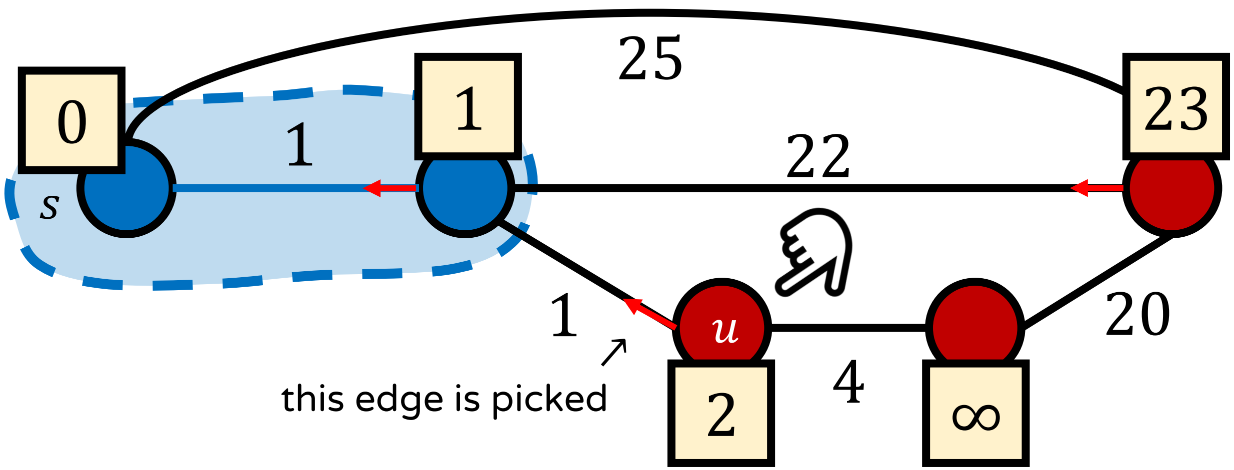


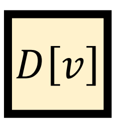
$D[v]$ Estimate of a node

Red: "not-sure"
Blue: "sure"

Example (8)

- **Step 2.** Pick a crossing edge from S to $V - S$
  - **Step 2-1.** Pick a "**not-sure**" node u with the smallest estimate $D[u]$



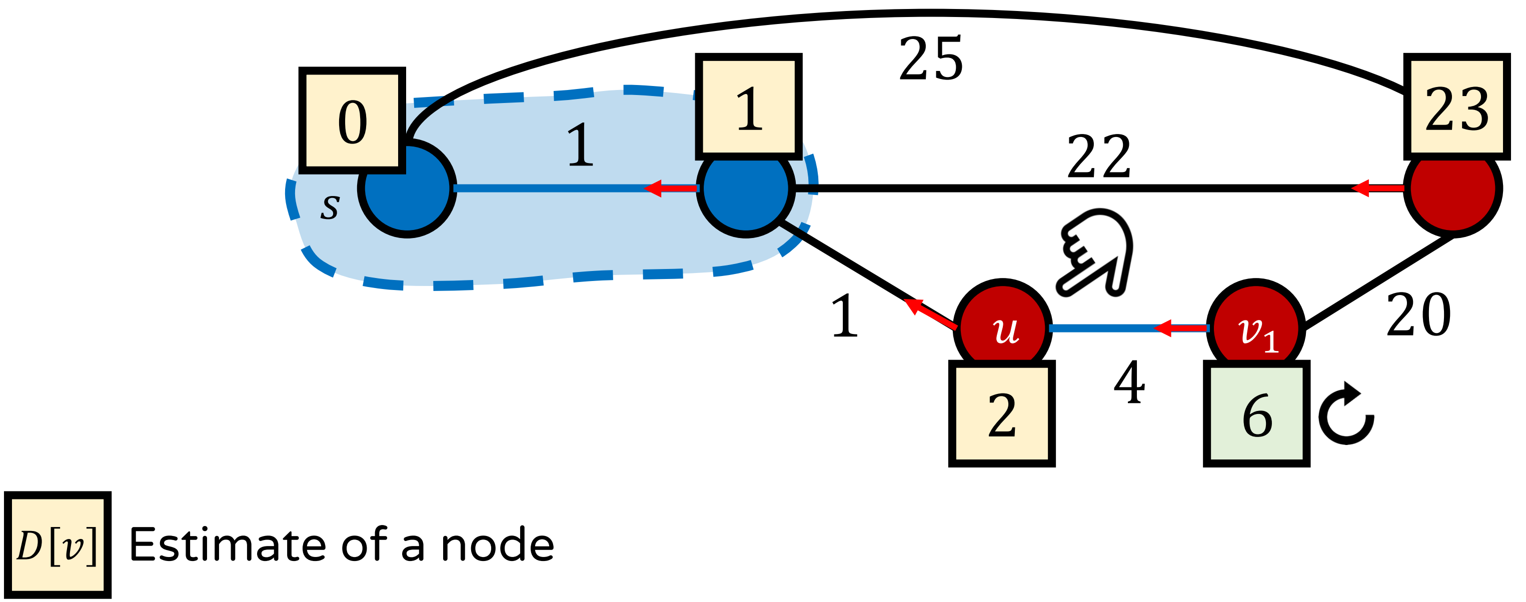


Estimate of a node

Red: "not-sure"
Blue: "sure"

Example (9)

- **Step 2.** Pick a crossing edge from S to $V - S$
  - **Step 2-2.** Update the estimate $D[v]$ to all **not-sure** neighbors v of node u if (u, v) makes a shorter path to v
    - $D[v] = \min\big(D[v], D[u] + w(u, v)\big)$ # called relaxation

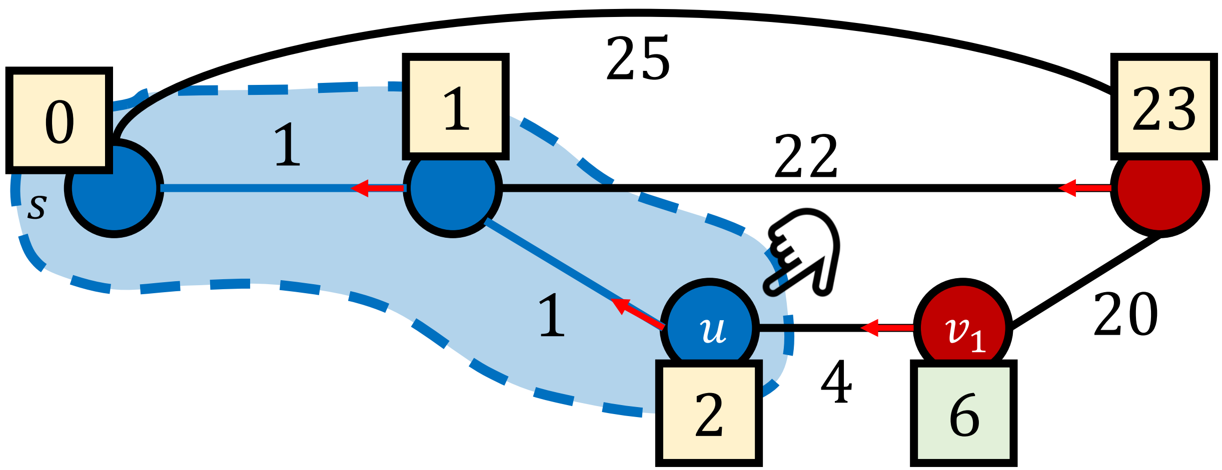


Estimate of a node

Red: “not-sure”
Blue: “sure”

Example (10)

- **Step 2.** Pick a crossing edge from S to $V - S$
  - **Step 2-3.** Move the other node u of the edge into S by marking the selected node u as "**sure**"

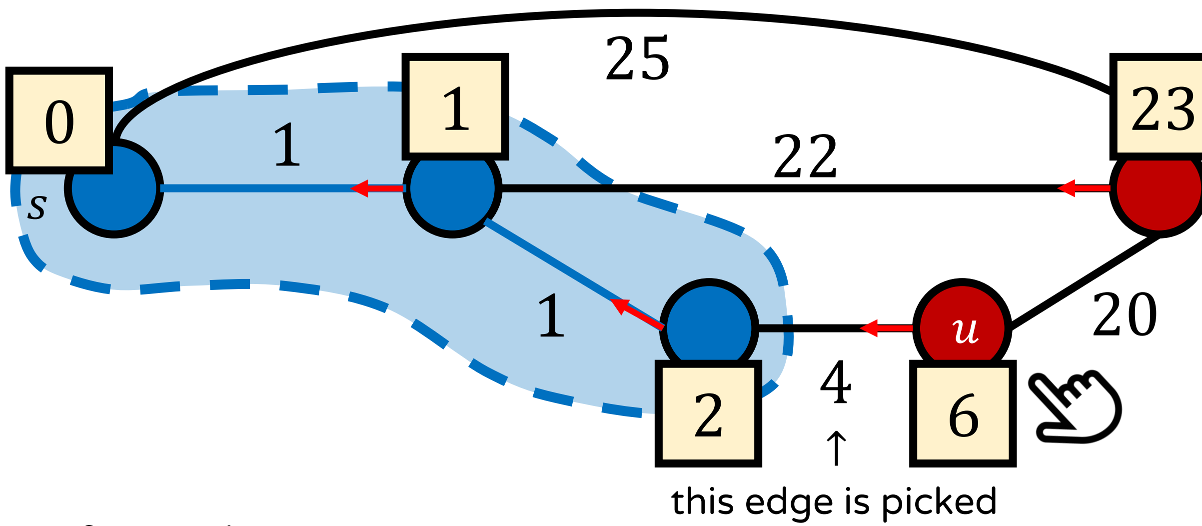


$D[v]$ Estimate of a node

Red: "not-sure"
Blue: "sure"

Example (10)

- **Step 2.** Pick a crossing edge from S to $V - S$
  - **Step 2-1.** Pick a "**not-sure**" node u with the smallest estimate $D[u]$

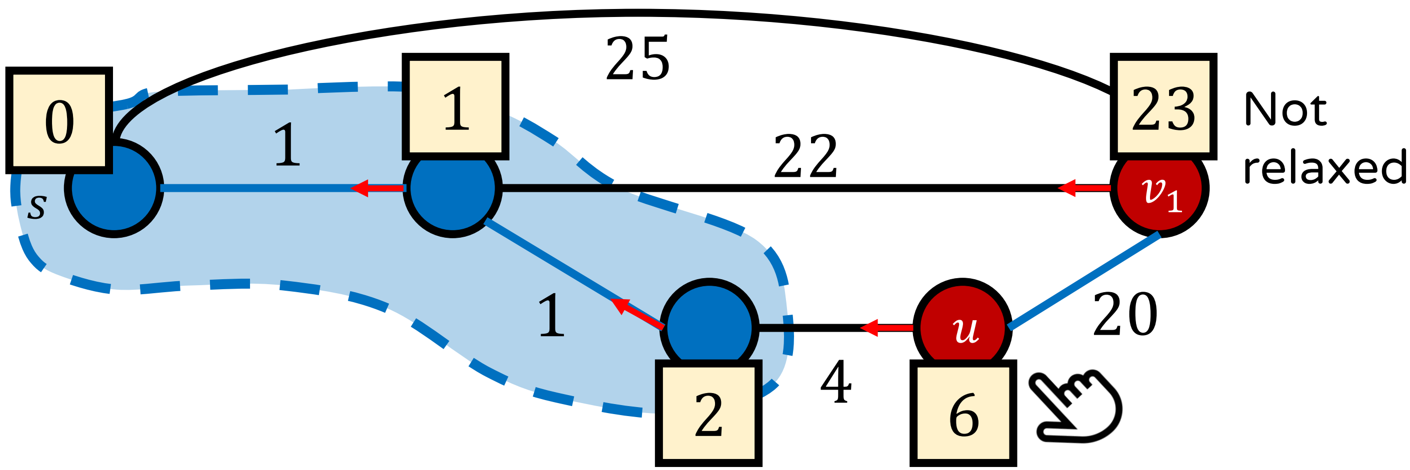


$D[v]$ Estimate of a node

Red: "not-sure"
Blue: "sure"

Example (10)

- **Step 2.** Pick a crossing edge from S to $V - S$
  - **Step 2-2.** Update the estimate $D[v]$ to all **not-sure** neighbors v of node u if (u, v) makes a shorter path to v
    - $D[v] = \min\big(D[v], D[u] + w(u, v)\big)$ # called relaxation

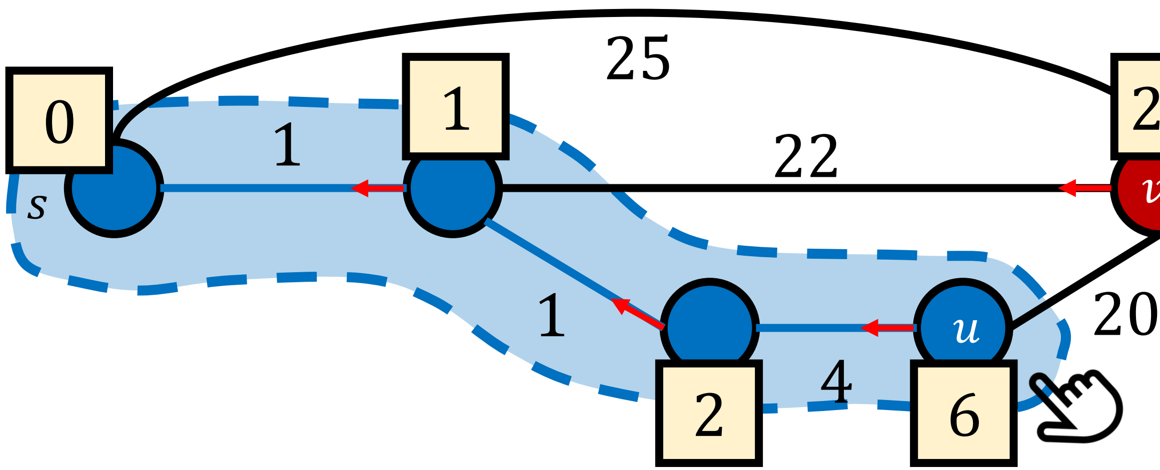


$D[v]$ Estimate of a node

Red: “not-sure”
Blue: “sure”

Example (11)

- **Step 2.** Pick a crossing edge from S to $V - S$
  - **Step 2-3.** Move the other node u of the edge into S by marking the selected node u as "**sure**"

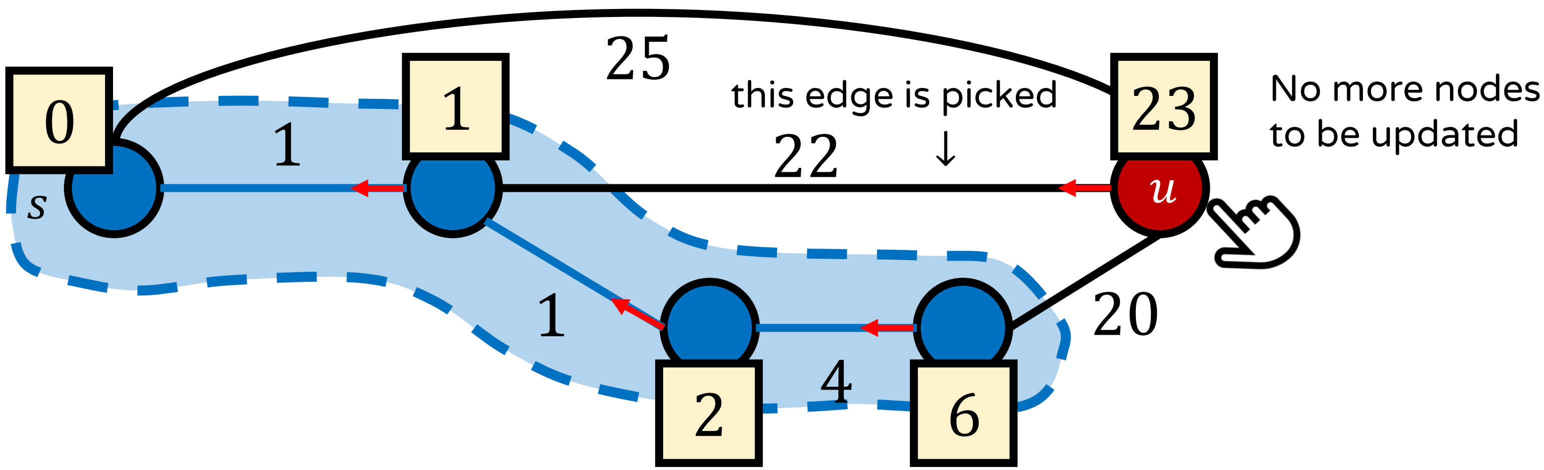


$D[v]$ Estimate of a node

Red: "not-sure"
Blue: "sure"

Example (12)

- **Step 2.** Pick a crossing edge from S to $V - S$
  - **Step 2-1.** Pick a "**not-sure**" node u with the smallest estimate $D[u]$

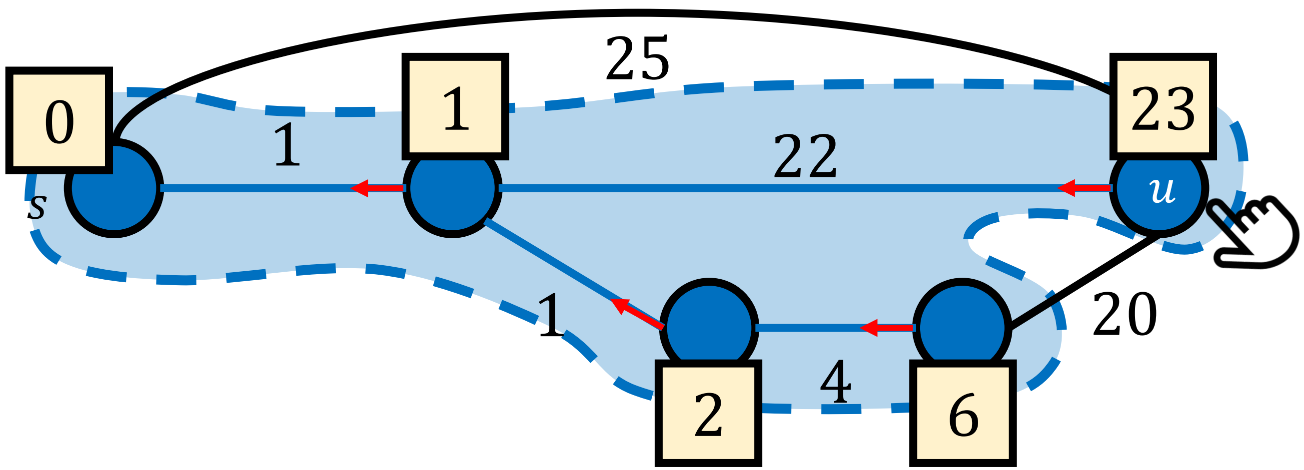


$D[v]$ Estimate of a node

Red: "not-sure"
Blue: "sure"

Example (12)

- **Step 2.** Pick a crossing edge from S to $V - S$
  - **Step 2-3.** Move the other node u of the edge into S by marking the selected node u as "**sure**"



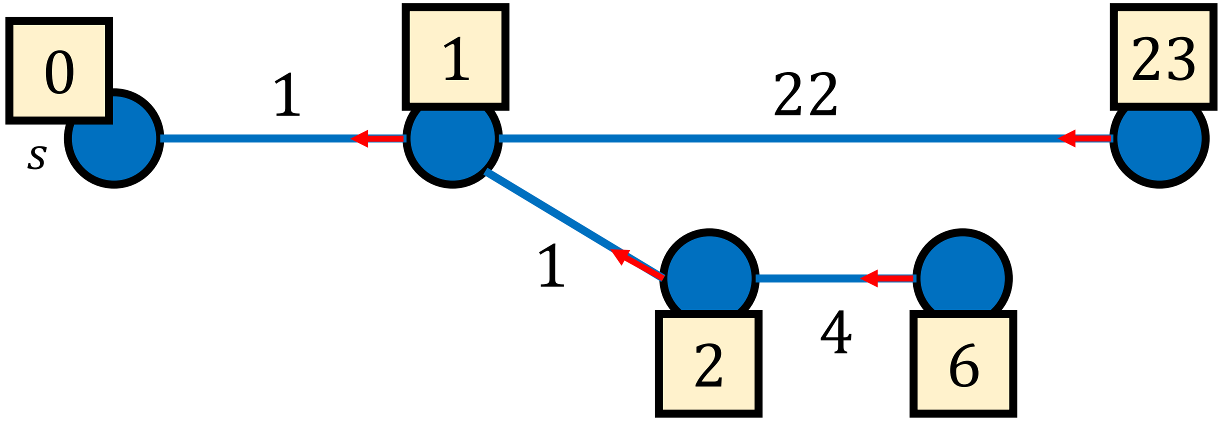
$D[v]$ Estimate of a node

Red: "not-sure"
Blue: "sure"

Example (12)

❑ Final shortest path tree with distances

- By Dijkstra's algorithm



$D[v]$ Estimate of a node

Red: "not-sure"
Blue: "sure"

Dijkstra's Algorithm

❑ Pseudocode (See the Appendix for details)

```
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

What You Need To Know

- **Discussion on minimum spanning tree**
  - Complexity analysis of Prim's algorithm
  - Discussion on MST algorithms
- **Single source shortest path**
  - Given a weighted graph G and a source node s, it is to find the shortest paths from s to all other nodes
- **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

In Next Lecture

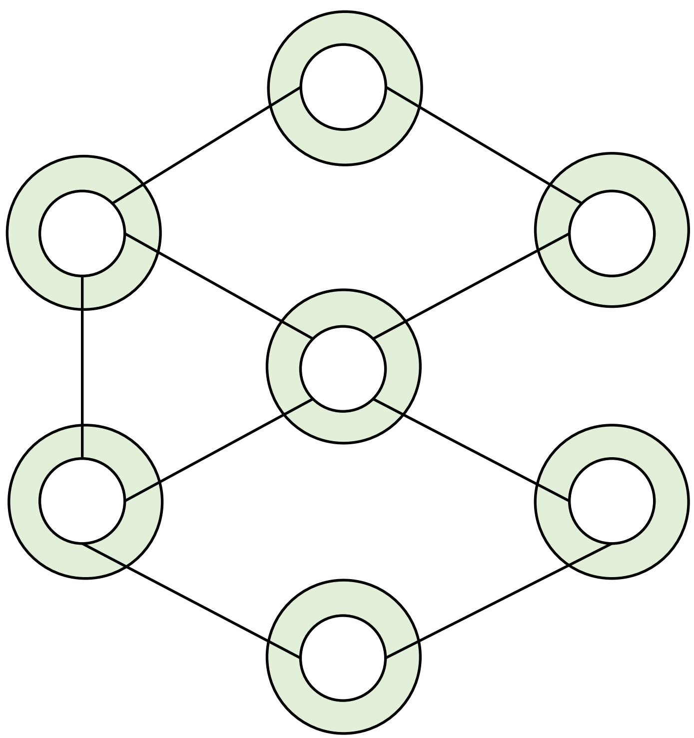
- **Discussion on Dijkstra's algorithm**
  - Implementation details
  - Discussions

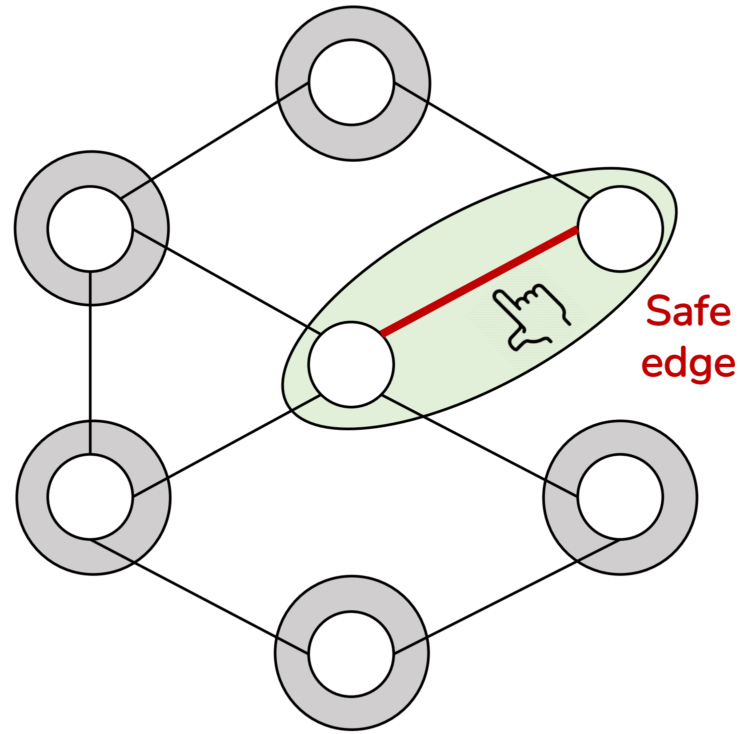
- **Single source shortest path with negative edges**
  - Bellman-Ford's algorithm

Thank You

Appendix: Generic MST (1)

- **Claim:** Generic MST produces a spanning tree of $n-1$ edges
  - Initially, Generic MST starts with n groups for each node as the below example.
  - One safe edge merges two groups into one group





Appendix: Generic MST (2)

- **Lemma 1:** $n - h$ groups remains in G with h safe edges.
  - **Base case:** If $h = 0$ (no safe edges), there are n groups in G.
  - **Inductive step:** Assume the lemma holds for $h = k$ safe edges.
    - Given $k + 1$ safe edges, k safe edges make $n - k$ groups remain.
    - One remaining safe edge will merge two groups into one, meaning # of groups decreases by 1; thus, there will be $n - k - 1$ groups.
      - A safe edge is crossing two groups by its definition; thus, adding it will merge the groups.
    - This means that the lemma also holds for $k + 1$ safe edges.
  - Thus, the lemma holds for any h safe edges.

Appendix: Generic MST (3)

❑ **Claim:** Generic MST produces a spanning tree of $n-1$ edges

- Thus, Generic MST terminates when one group remains in G, meaning it selects $h = n - 1$ safe edges by Lemma 1.

- All n nodes are inlucded in the final group, and connected with $n-1$ edges, which satisfies the definition of spanning tree.