Graph Algorithms 2

Gianmaria Silvello

Department of Information Engineering
University of Padua

silvello@dei.unipd.it

http://www.dei.unipd.it/~silvello/

Fundamentals of Information systems

Master Degree in Data Science

Outline

- Single Source Shortest Path
- Minimum Spanning Tree
- All Pairs Shortest Path
- Centrality

Reference: Chapters 22-24
of CLRS

Reference: Chapter 4
of Needham and Hodler, Graph algorithms, O'Reilly, 2019

Single Source Shortest Path

Single Source Shortest Path

- The Shortest Path algorithm calculates the shortest (weighted) path between a pair of nodes. It’s useful for user interactions and dynamic workflows because it works in real time.

Single Source Shortest Path

- The Shortest Path algorithm calculates the shortest (weighted) path between a pair of nodes. It's useful for user interactions and dynamic workflows because it works in real time.
- Use Shortest Path to find optimal routes between a pair of nodes, based on either the number of hops or any weighted relationship value.

Single Source Shortest Path

- The Shortest Path algorithm calculates the shortest (weighted) path between a pair of nodes. It's useful for user interactions and dynamic workflows because it works in real time.
- Use Shortest Path to find optimal routes between a pair of nodes, based on either the number of hops or any weighted relationship value.
- It can provide real-time answers about degrees of separation, the shortest distance between points, or the least expensive route.

Single Source Shortest Path

Single Source Shortest Path

- Let G(V,E) a weighted graph s.t. there exists w:E → **R**

Single Source Shortest Path

- Let G(V,E) a weighted graph s.t. there exists w:E → **R**
- We define a direct path from v_1 to v_k as $v_1 \rightsquigarrow v_k$ as p = $v_1 \rightarrow v_2 \rightarrow \ldots v_{k-1} \rightarrow v_k$

Single Source Shortest Path

- Let G(V,E) a weighted graph s.t. there exists w:E → **R**
- We define a direct path from v_1 to v_k as $v_1 \rightsquigarrow v_k$ as $p = v_1 \rightarrow v_2 \rightarrow \ldots v_{k-1} \rightarrow v_k$
- The weight of p is $w(p) = \sum_i w(v_i, v_{i+1})$
    - The weights can be negative, positive or zero

Single Source Shortest Path

- Let G(V,E) a weighted graph s.t. there exists w:E → **R**
- We define a direct path from v_1 to v_k as $v_1 \rightsquigarrow v_k$ as $p = v_1 \rightarrow v_2 \rightarrow \ldots v_{k-1} \rightarrow v_k$
- The weight of p is $w(p) = \sum_i w(v_i, v_{i+1})$
    - The weights can be negative, positive or zero
- Shortest path from u to v is the $p_{u,v}$ with minimum weight

Single Source Shortest Path

- Let G(V,E) a weighted graph s.t. there exists w:E → **R**
- We define a direct path from v_1 to v_k as $v_1 \rightsquigarrow v_k$ as p = $v_1 \rightarrow v_2 \rightarrow \ldots v_{k-1} \rightarrow v_k$
- The weight of p is $w(p) = \sum_i w(v_i, v_{i+1})$
  - The weights can be negative, positive or zero
- Shortest path from u to v is the $p_{u,v}$ with minimum weight
- A simplification of this problem is to calculate the weight of the shortest path rather than the path itself
  - $\delta(u,v) = \min\{w(p) : p \text{ from u to v}\}$

Dijkstra Algorithm

- Dijkstra's Shortest Path algorithm operates by first finding the lowest-weight relationship from the start node to directly connected nodes.
- It keeps track of those weights and moves to the "closest" node.
- It then performs the same calculation, but now as a cumulative total from the start node.
- The algorithm continues to do this, evaluating a "wave" of cumulative weights and always choosing the lowest weighted cumulative path to advance along, until it reaches the destination node.

Dijkstra Algorithm

```
DIJKSTRA(G, w, s)
  INIT-SINGLE-SOURCE(G, s)
  S = ∅
  for each vertex u ∈ G.V
      INSERT(Q, u)
  while Q ≠ ∅
      u = EXTRACT-MIN(Q)
      S = S ∪ {u}
      for each vertex v ∈ G.Adj[u]
          RELAX(u, v, w)
          if v.d changed
              DECREASE-KEY(Q, v, v.d)
```

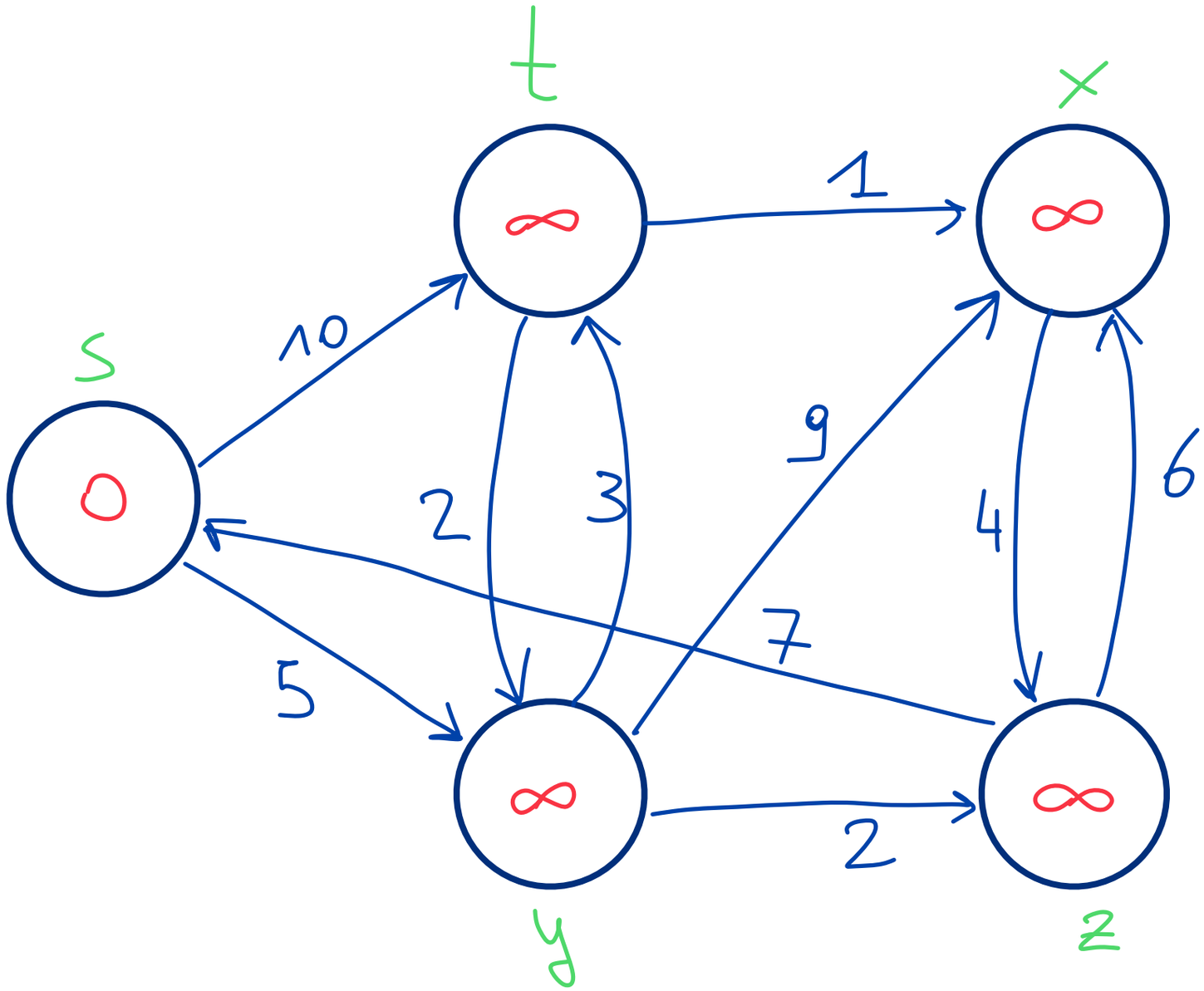
```
INIT-SINGLE-SOURCE(G, s)
  for each v ∈ G.V
      v.d = ∞
      v.π = NIL
  s.d = 0
```

```
RELAX(u, v, w)
  if v.d > u.d + w(u, v)
      v.d = u.d + w(u, v)
      v.π = u
```

Q is a Min Priority Queue

Example at the blackboard based on CLRS book.

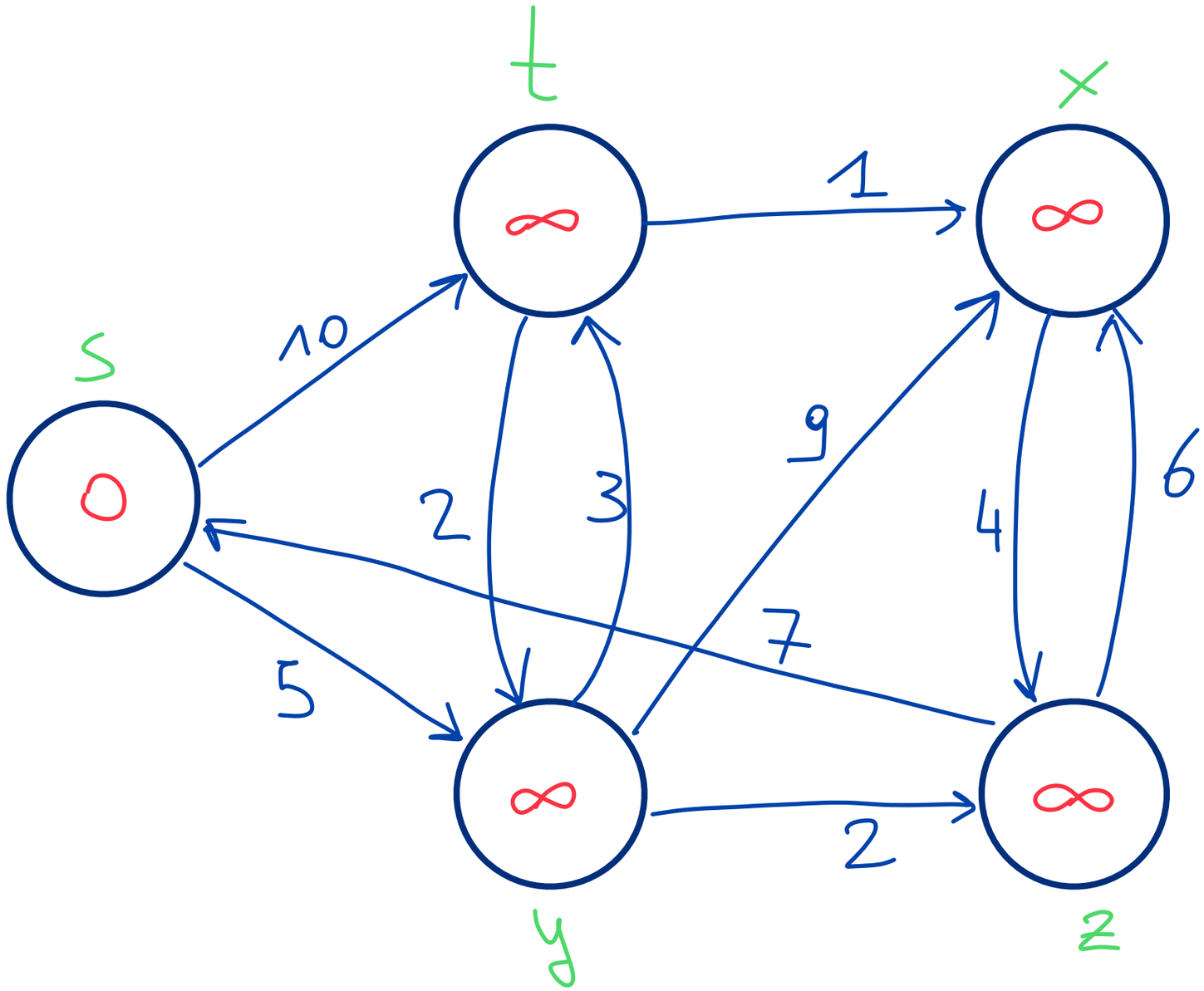
Dijkstra exercise



$Q = [s, t, x, y, z]$

$0 \quad \infty \quad \infty \quad \infty \quad \infty$

Dijkstra exercise

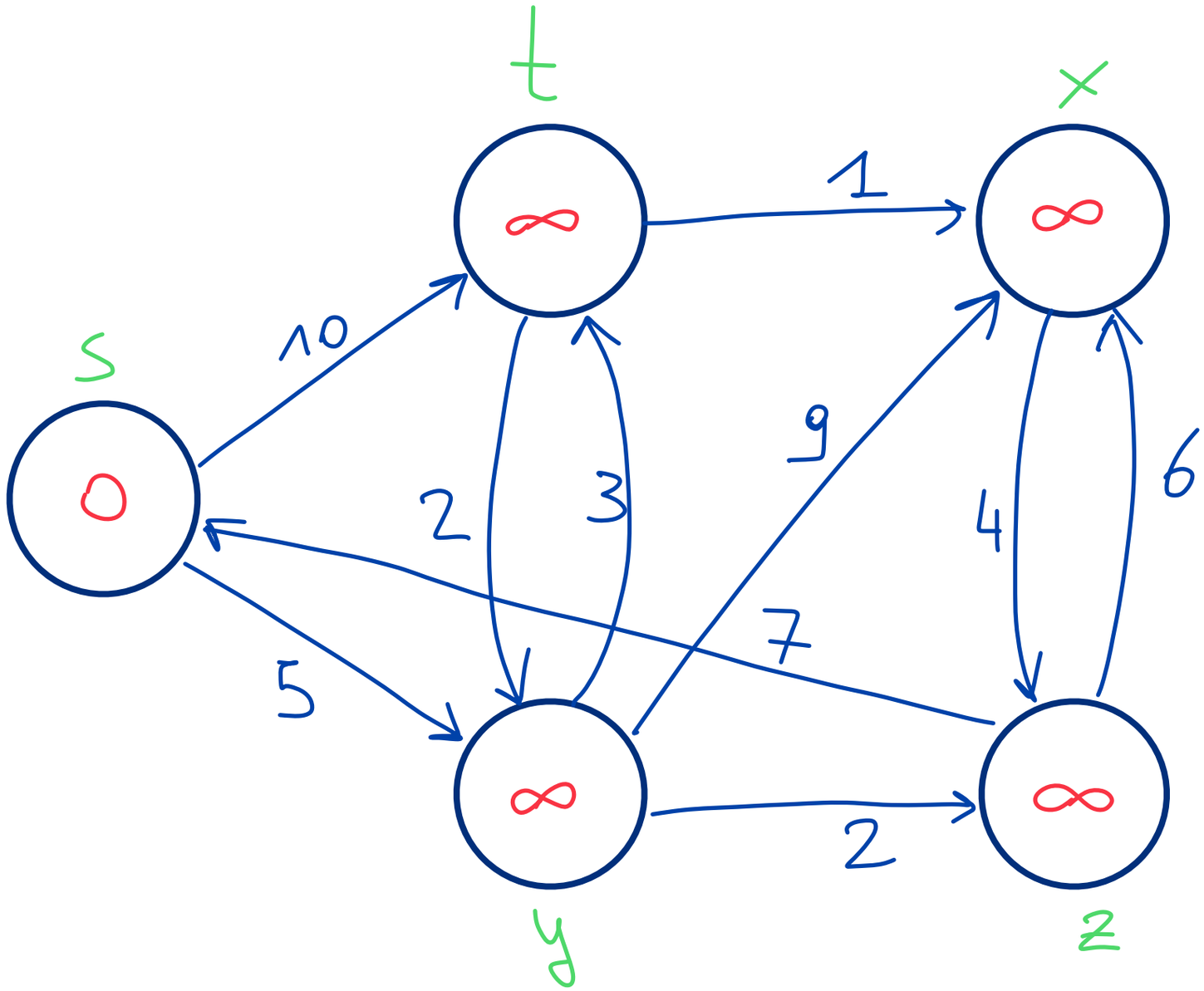


Initialization

$Q = [s, t, x, y, z]$

$0 \quad \infty \quad \infty \quad \infty \quad \infty$

Dijkstra exercise



Initialization

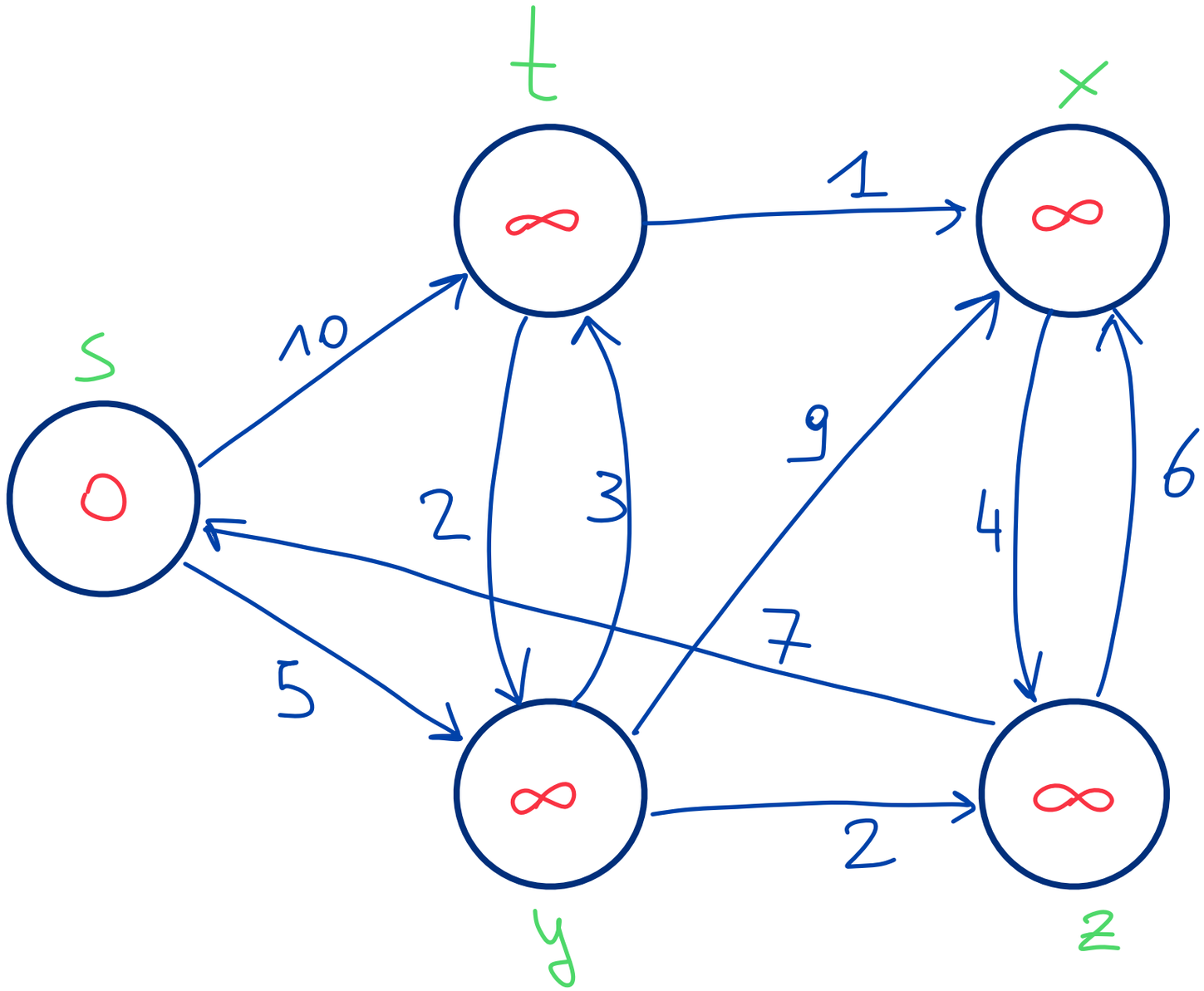
$Q = [s, t, x, y, z]$

$0 \quad \infty \quad \infty \quad \infty \quad \infty$

extract-min

$S = \{s\}$

Dijkstra exercise



Initialization

$Q = [s, t, x, y, z]$

$0 \quad \infty \quad \infty \quad \infty \quad \infty$

extract-min

$S = \{s\}$

→ RELAX $\forall$ vertex adj to s

→ $adj(s) = \{t, y\}$

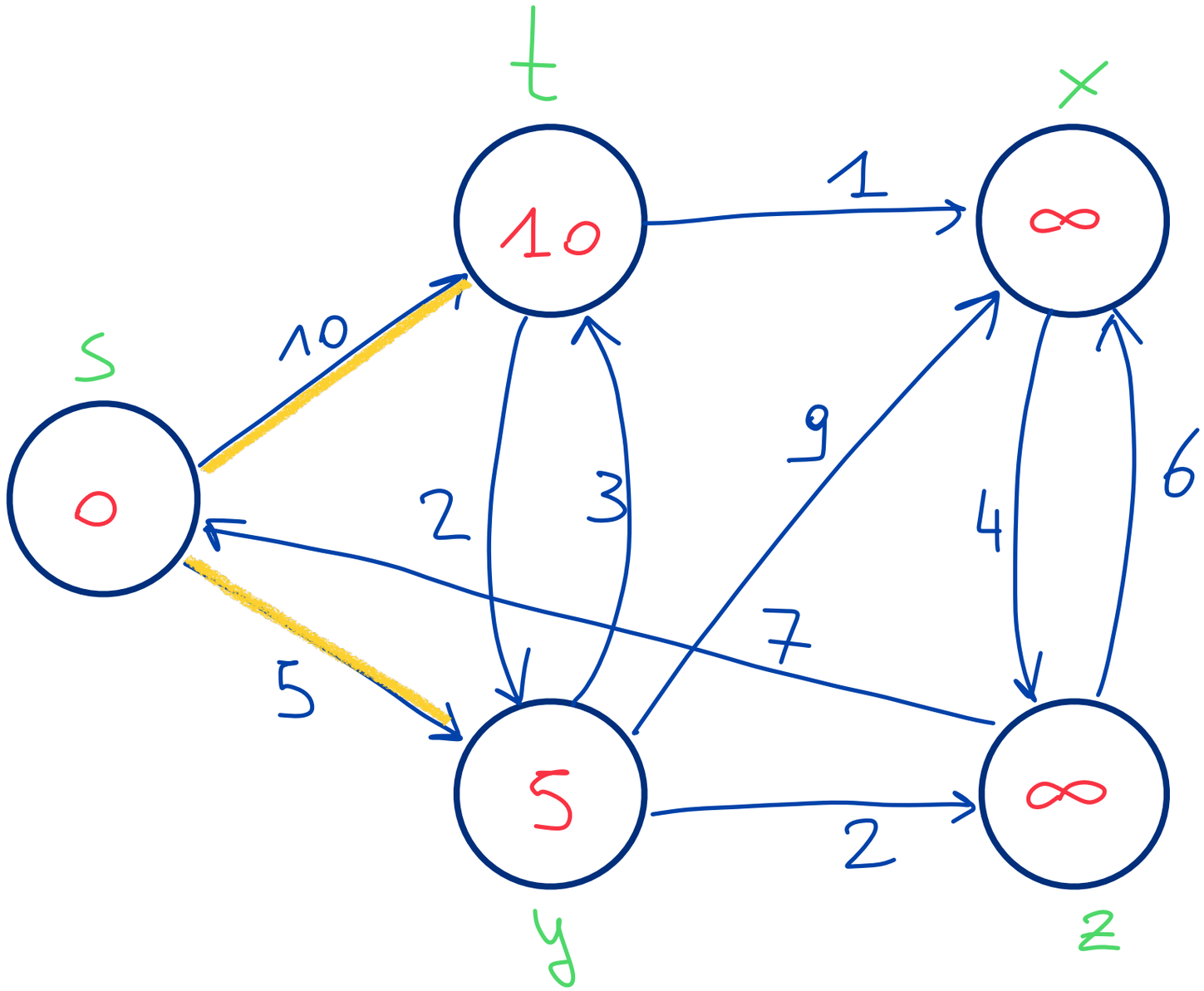
RELAX (s, t, w) =

if $t.d > s.d + w(s,t)$

⇒ $\infty > 0 + 10$

⇒ $t.d = 0 + 10$

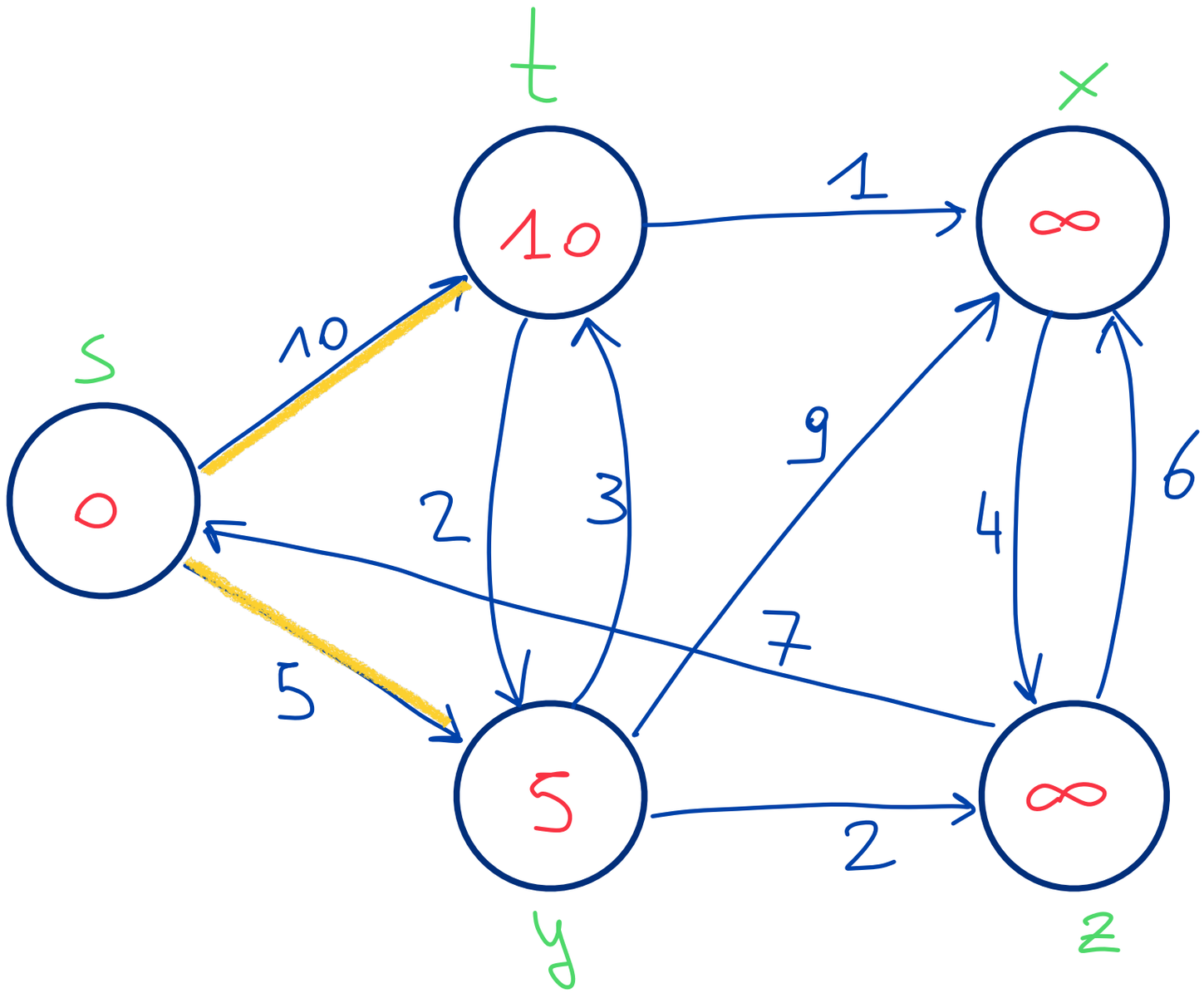
Dijkstra exercise



$Q = [t, y, x, z]$

$10 \quad 5 \quad \infty \quad \infty$

Dijkstra exercise



$Q = [t, y, x, z]$

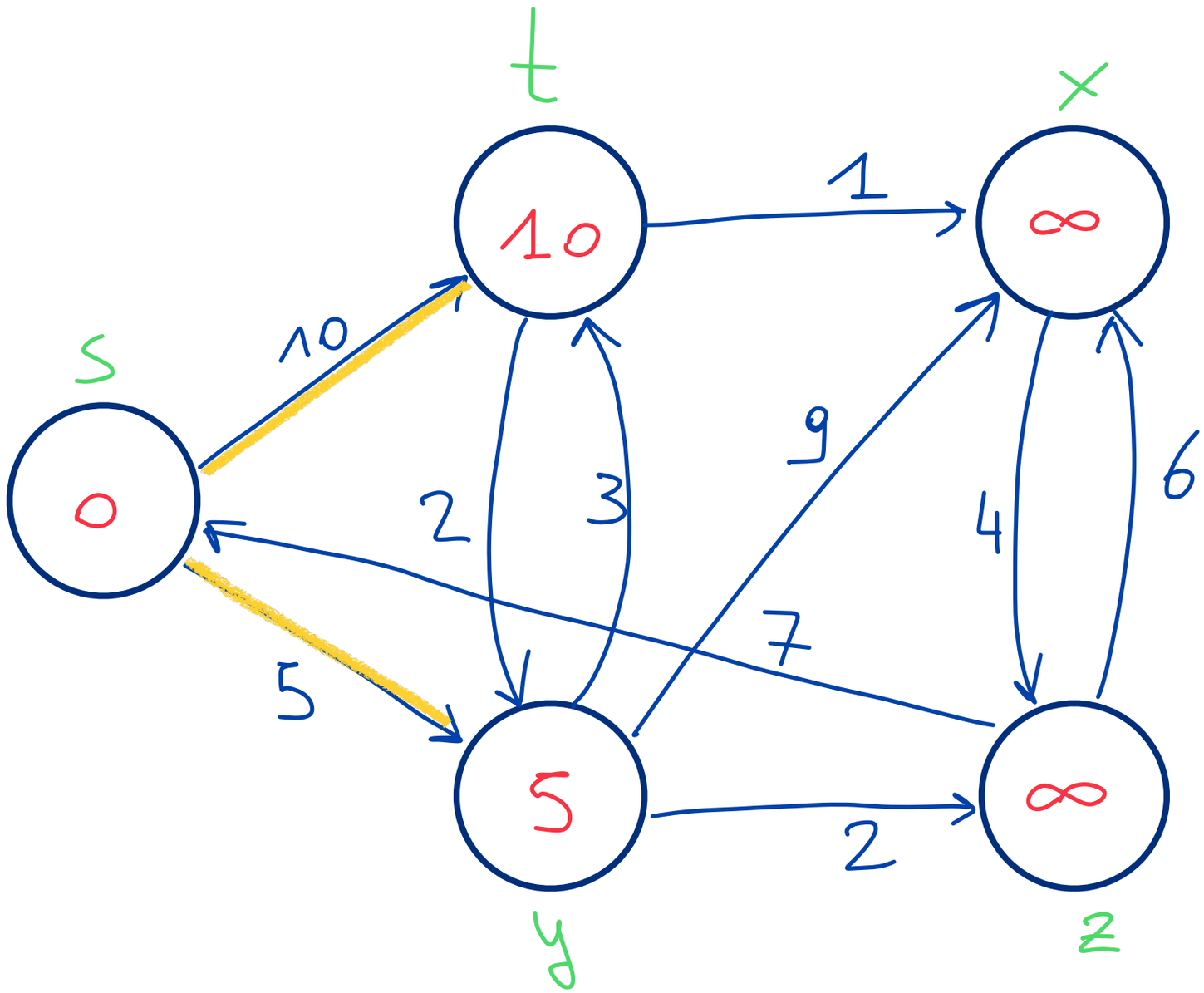
10 5 ∞ ∞

↓

extract-min

$adj(y) = \{t, z, x\}$

Dijkstra exercise



$Q = [t, y, x, z]$

$10 \quad 5 \quad \infty \quad \infty$

↓

extract-min

$adj(y) = \{t, z, x\}$

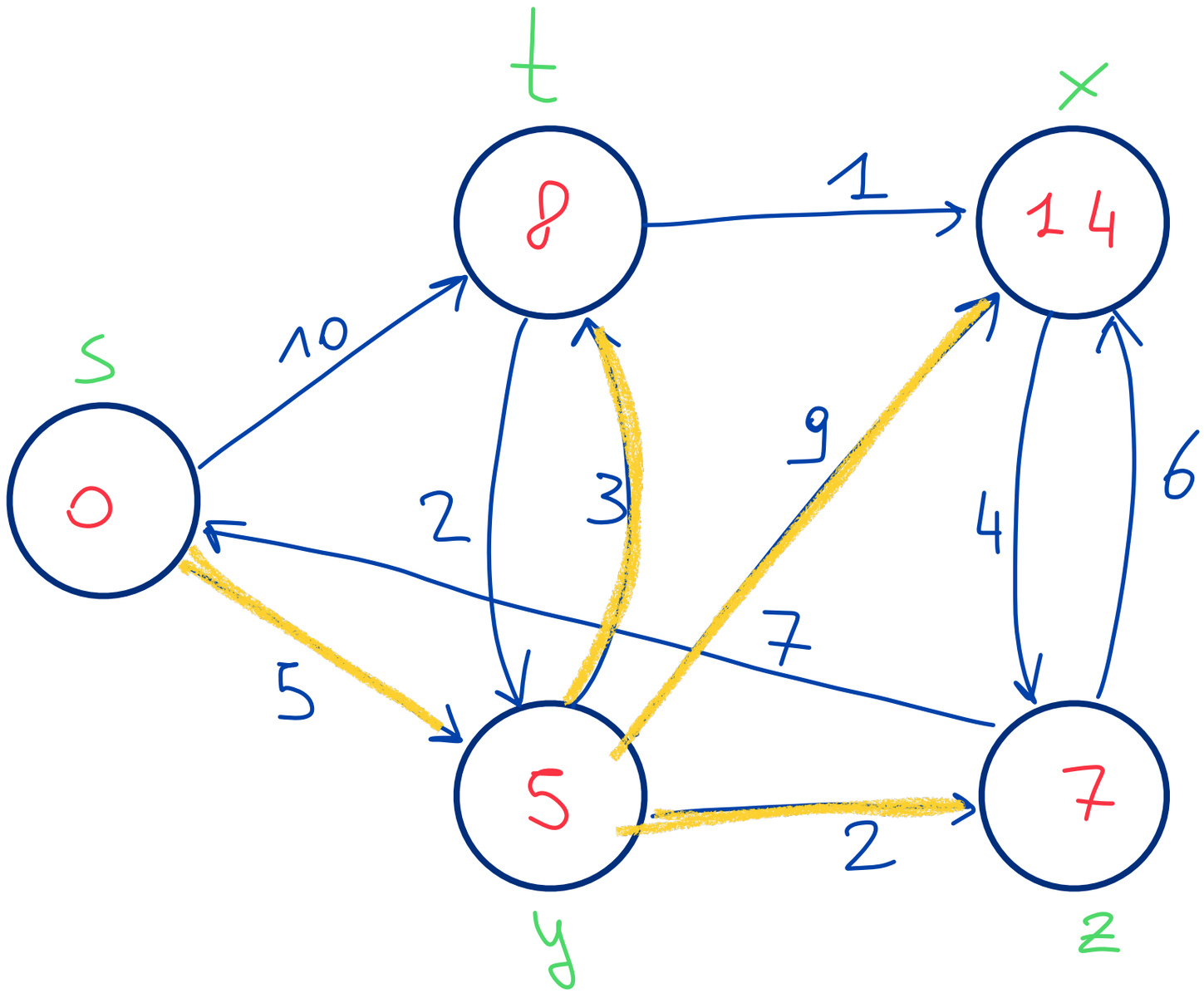
if $t.d > y.d + w(y,t)$

$10 > 5 + 3$

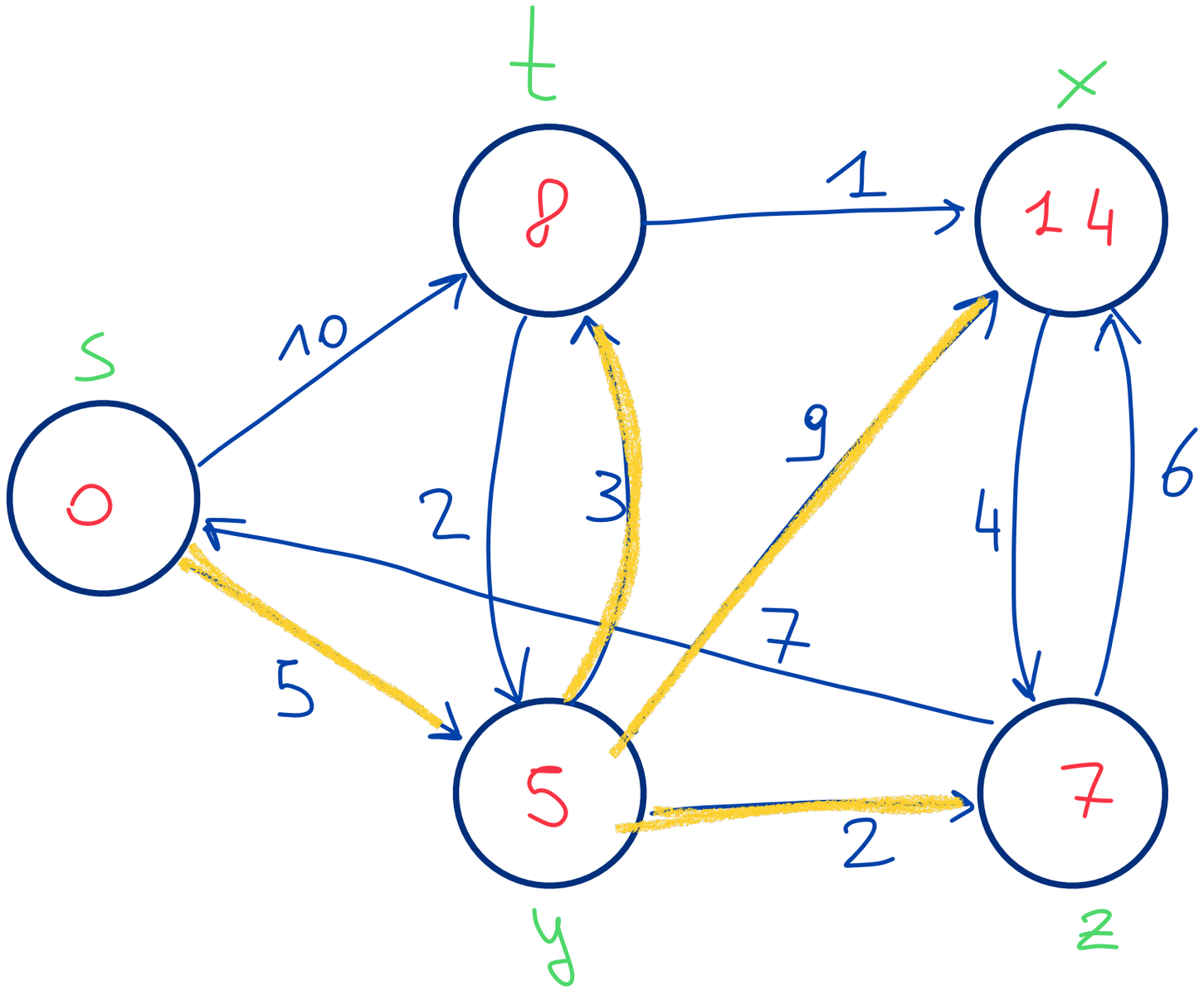
$\Rightarrow t.d = 8$

Dijkstra exercise

$Q = [\ t \quad x \quad z\ ]$
$\quad\quad 8 \quad 14 \quad 7$



Dijkstra exercise



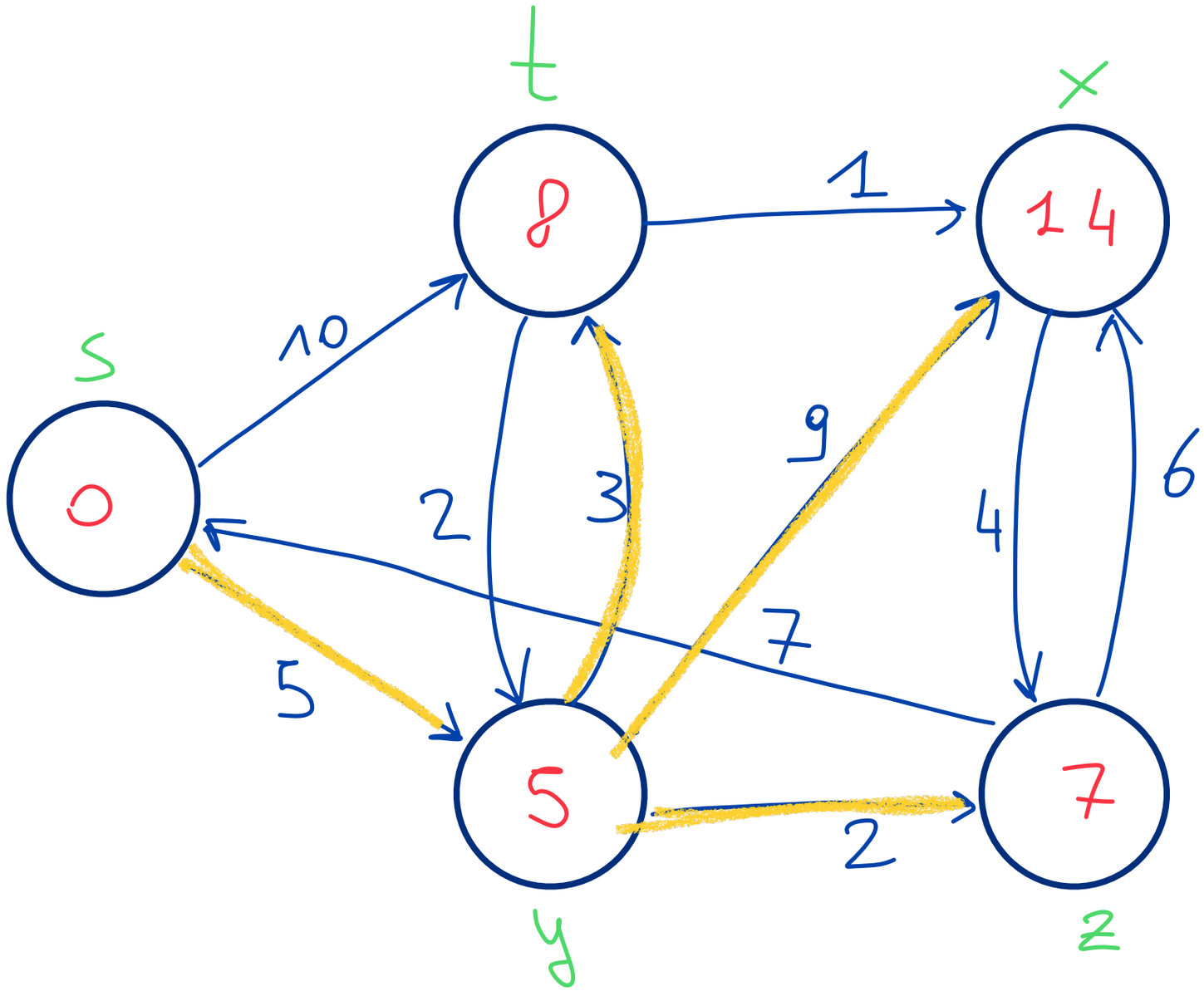
$Q = [\ t \quad x \quad z\ ]$

$8 \quad 14 \quad 7$

↓ extract-min

$adj(z) = \{x, s\}$

Dijkstra exercise



$Q = [\ t \quad x \quad z\ ]$

$8 \quad 14 \quad 7$

↓ extract-min

$adj(z) = \{x, s\}$

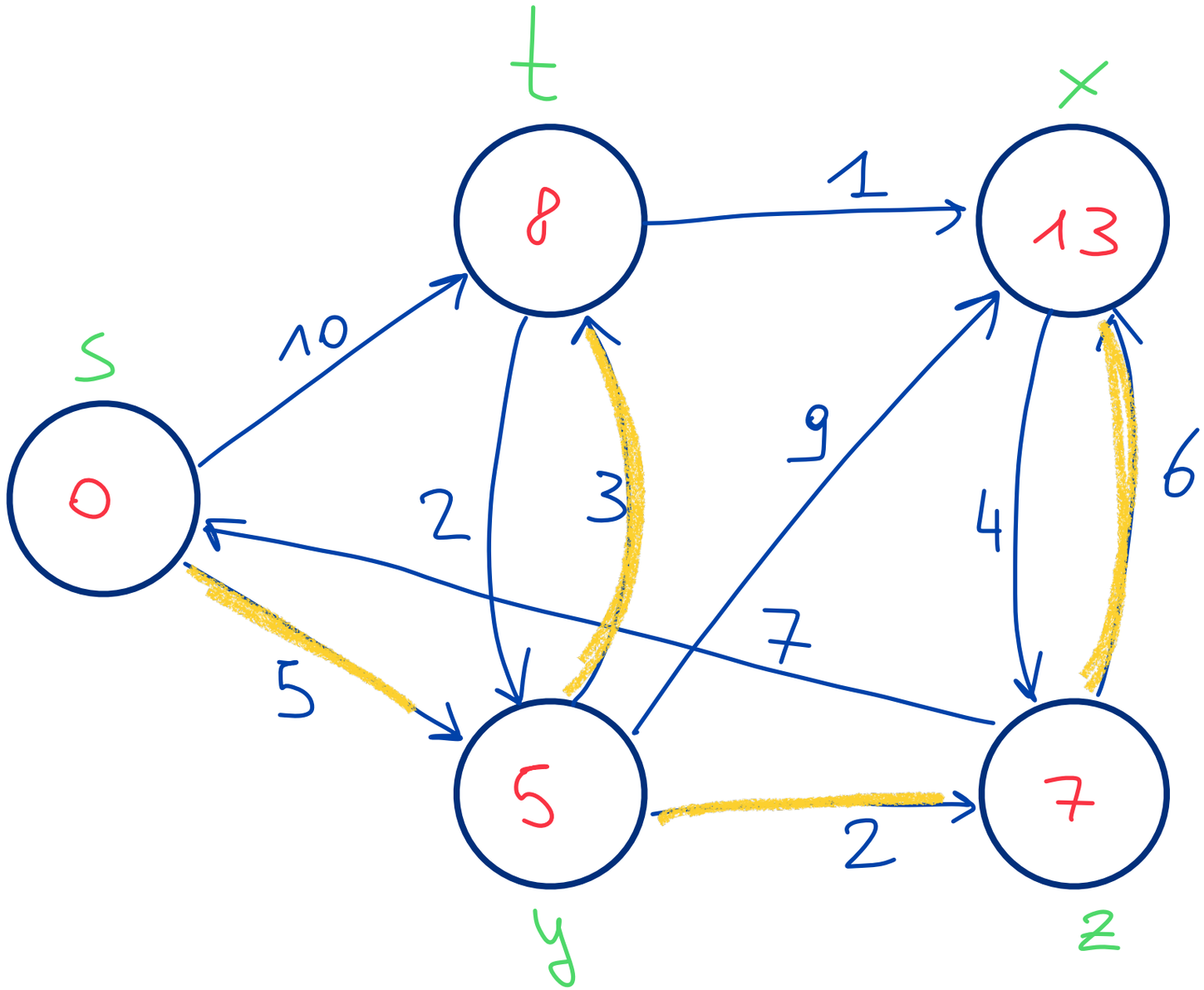
if $14 > 7+6$

TRUE => $x.d = 13$

if $0 > 7+7$

FALSE => DO NOTHING

Dijkstra exercise



$Q = [\ t\ ,\ x\ ]$
$\quad\ 8 \quad 13$

↓

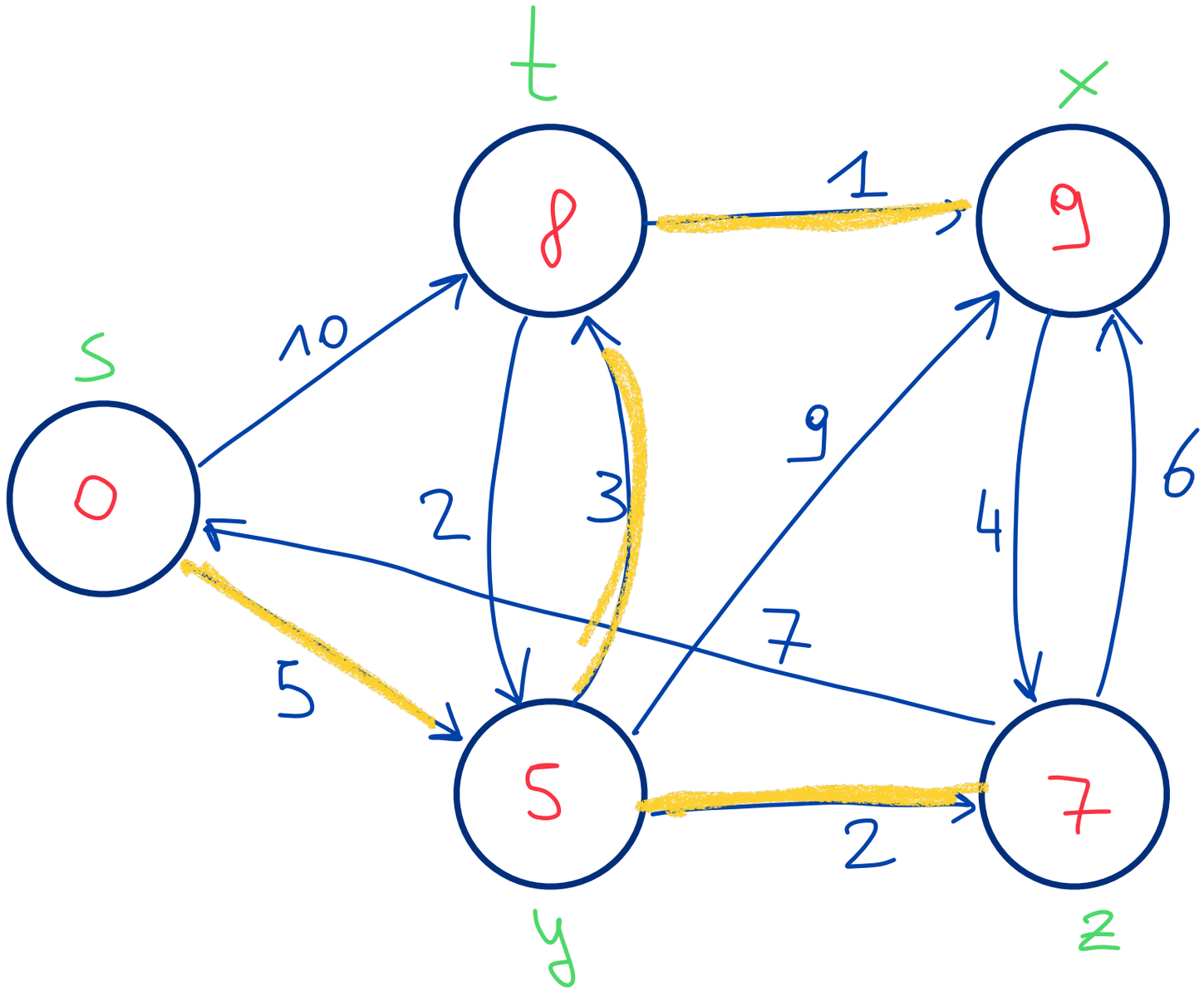
extract-min

$adj(t) = \{x, y\}$

$y.d > t.d + w(t,y)$?
→ FALSE

$x.d > t.d + w(t,x)$?
→ TRUE
⇒ $x.d = 8 + 1 = 9$

Dijkstra exercise



$Q = [\ x\ ,\ y\ ]$

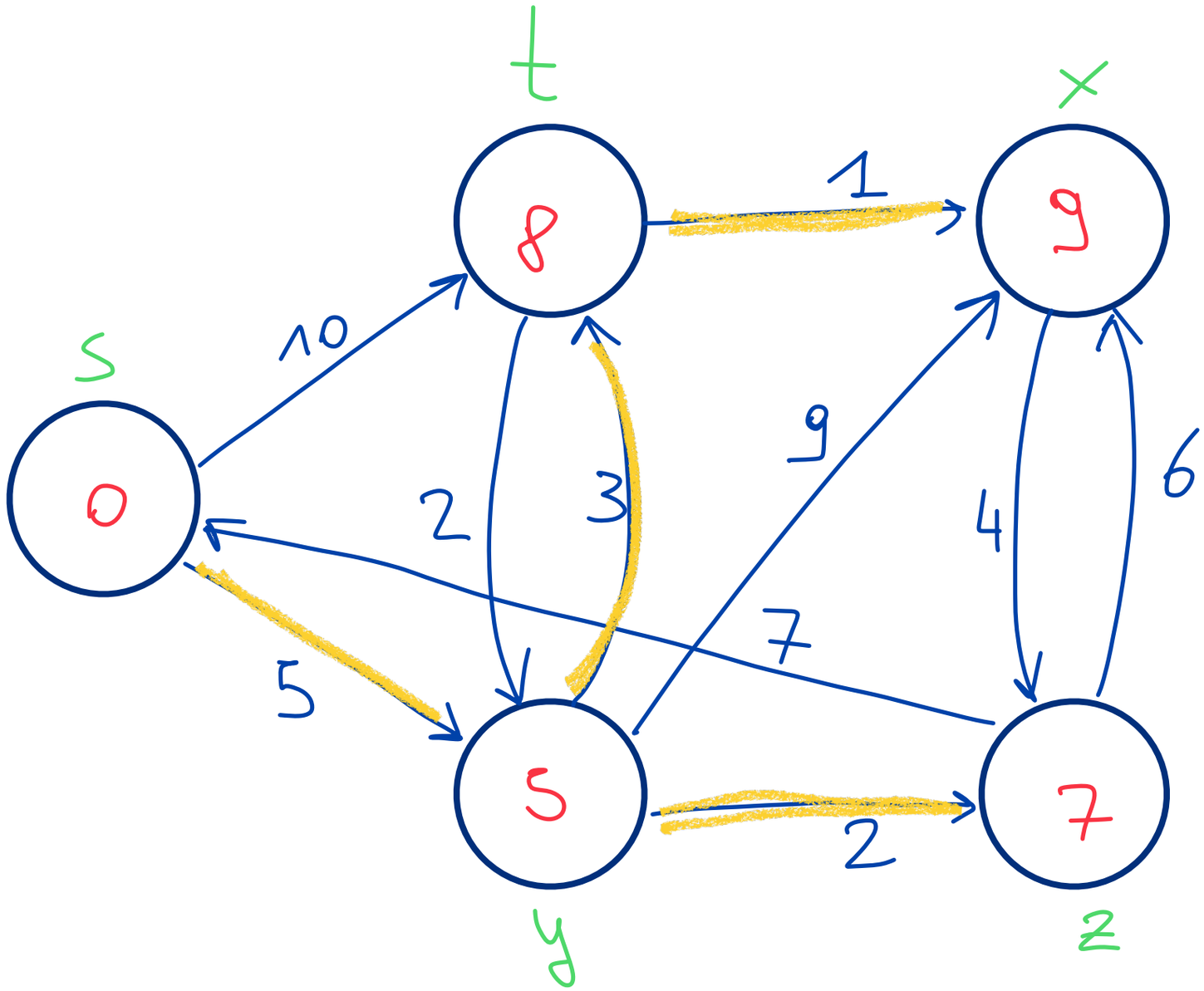
↓ extract-min

$adj(x) = \{z\}$

$x.d > z.d + w(x,z)$

⇒ DO NOTHING

Dijkstra exercise



Q = [ ]

→ EMPTY

↓

EXIT

Single Source Shortest Path Properties

Single Source Shortest Path Properties

- **Triangle Inequality**
  - For any edge $(u,v) \in E$, we have $\delta(s,v) \leq \delta(s,u) + w(u,v)$

Single Source Shortest Path Properties

- **Triangle Inequality**
  - For any edge $(u,v) \in E$, we have $\delta(s,v) \leq \delta(s,u) + w(u,v)$
- **Upper-bound property**
  - We always have $v.d > \delta(s,v)$ for all vertices $v \in V$, and once $v.d$ achieves the value $\delta(s,v)$, it never changes

Single Source Shortest Path Properties

- **Triangle Inequality**
  - For any edge $(u,v) \in E$, we have $\delta(s,v) \leq \delta(s,u) + w(u,v)$
- **Upper-bound property**
  - We always have $v.d > \delta(s,v)$ for all vertices $v \in V$, and once $v.d$ achieves the value $\delta(s,v)$, it never changes
- **No-path property**
  - If there is no path from s and v, then we always have $v.d = \delta(s,v) = \infty$

Single Source Shortest Path Properties

- **Triangle Inequality**
  - For any edge $(u,v) \in E$, we have $\delta(s,v) \leq \delta(s,u) + w(u,v)$
- **Upper-bound property**
  - We always have $v.d > \delta(s,v)$ for all vertices $v \in V$, and once $v.d$ achieves the value $\delta(s,v)$, it never changes
- **No-path property**
  - If there is no path from s and v, then we always have $v.d = \delta(s,v) = \infty$
- **Convergence property**
  - If $s \rightsquigarrow u \rightarrow v$ is a shortest path in G for some $u,v \in V$, and if $u.d = \delta(s,u)$ at any time prior to relaxing edge (u,v), then $v.d = \delta(s,v)$ at all times afterwards

Minimum Spanning Tree

Minimum Spanning Tree

- Let G(V,E) a weighted graph s.t. there exists w:E → **R**

Minimum Spanning Tree

- Let G(V,E) a weighted graph s.t. there exists w:E → **R**
- The Minimum Spanning Tree algorithm starts from a given node and finds all its reachable nodes and the set of relationships that connect the nodes together with the minimum possible weight

Minimum Spanning Tree

- Let G(V,E) a weighted graph s.t. there exists w:E → **R**
- The Minimum Spanning Tree algorithm starts from a given node and finds all its reachable nodes and the set of relationships that connect the nodes together with the minimum possible weight
- The first known Minimum Weight Spanning Tree algorithm was developed by the Czech scientist Otakar Borůvka in 1926. **Prim**'s algorithm, invented in 1957, is the simplest and best known.

Minimum Spanning Tree

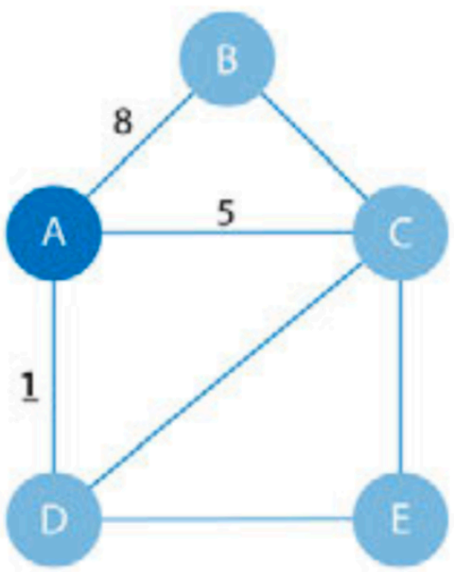
- Let G(V,E) a weighted graph s.t. there exists w:E → **R**
- The Minimum Spanning Tree algorithm starts from a given node and finds all its reachable nodes and the set of relationships that connect the nodes together with the minimum possible weight
- The first known Minimum Weight Spanning Tree algorithm was developed by the Czech scientist Otakar Borůvka in 1926. **Prim**'s algorithm, invented in 1957, is the simplest and best known.
- Prim's algorithm is similar to Dijkstra's Shortest Path algorithm, but rather than minimizing the total length of a path ending at each relationship, it minimizes the length of each relationship individually.
    - Negative weights are allowed

Prim's MST: Main Steps

- It begins with a tree containing only one node.
- The relationship with smallest weight coming from that node is selected and added to the tree (along with its connected node).
- This process is repeated, always choosing the minimal-weight relationship that joins any node not already in the tree.
- When there are no more nodes to add, the tree is a minimum spanning tree.

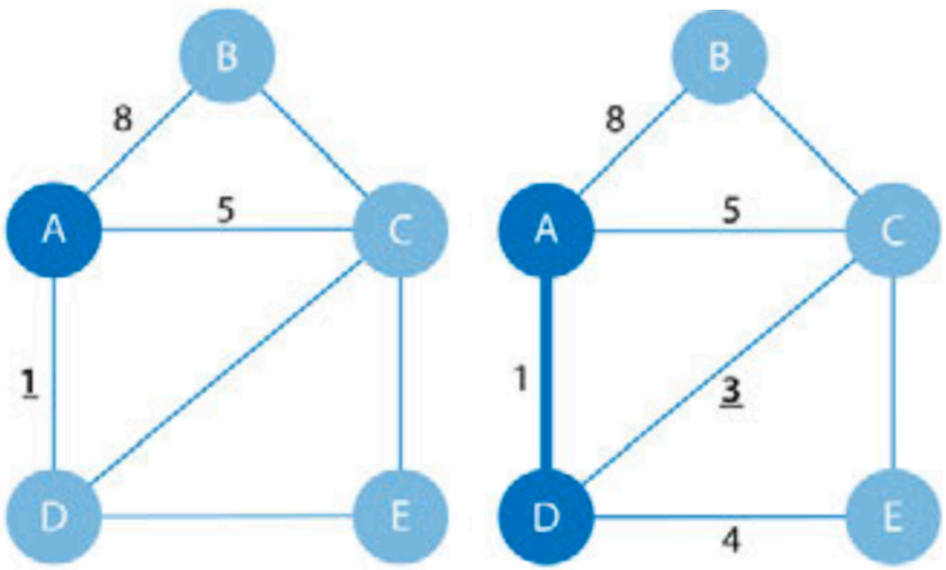
Prim's MST: Main Steps

- It begins with a tree containing only one node.
- The relationship with smallest weight coming from that node is selected and added to the tree (along with its connected node).
- This process is repeated, always choosing the minimal-weight relationship that joins any node not already in the tree.
- When there are no more nodes to add, the tree is a minimum spanning tree.



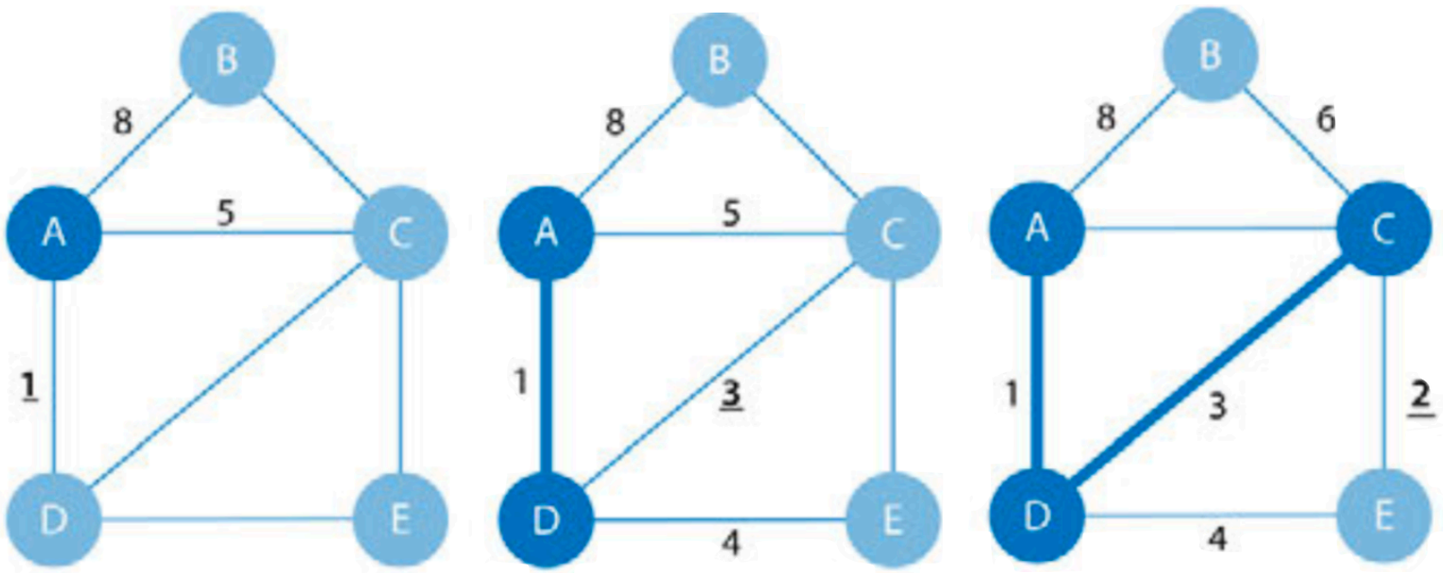
Prim's MST: Main Steps

- It begins with a tree containing only one node.
- The relationship with smallest weight coming from that node is selected and added to the tree (along with its connected node).
- This process is repeated, always choosing the minimal-weight relationship that joins any node not already in the tree.
- When there are no more nodes to add, the tree is a minimum spanning tree.



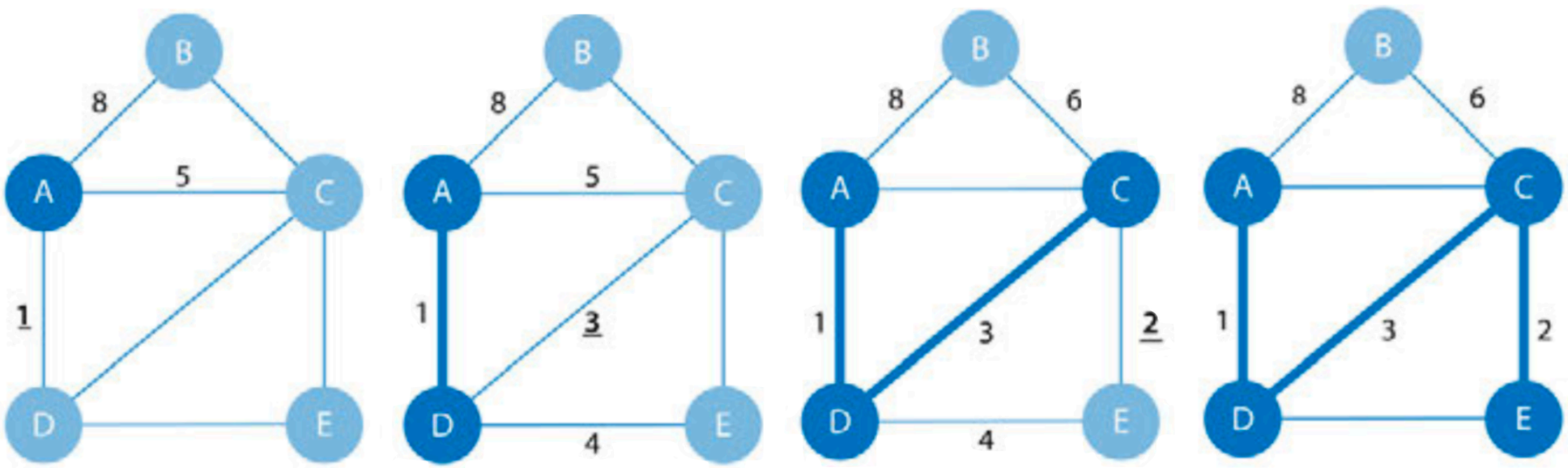
Prim's MST: Main Steps

- It begins with a tree containing only one node.
- The relationship with smallest weight coming from that node is selected and added to the tree (along with its connected node).
- This process is repeated, always choosing the minimal-weight relationship that joins any node not already in the tree.
- When there are no more nodes to add, the tree is a minimum spanning tree.



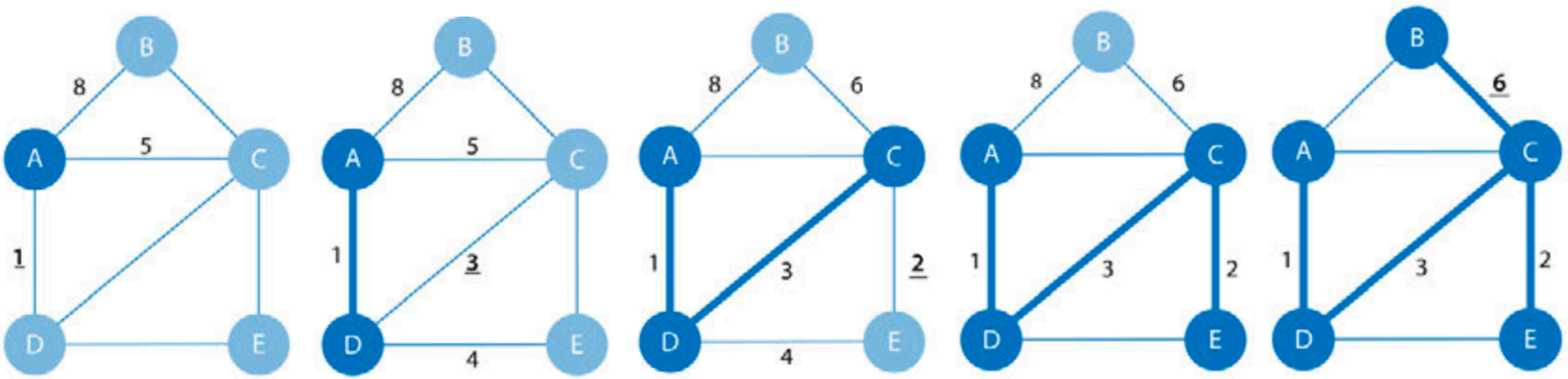
Prim's MST: Main Steps

- It begins with a tree containing only one node.
- The relationship with smallest weight coming from that node is selected and added to the tree (along with its connected node).
- This process is repeated, always choosing the minimal-weight relationship that joins any node not already in the tree.
- When there are no more nodes to add, the tree is a minimum spanning tree.



Prim's MST: Main Steps

- It begins with a tree containing only one node.
- The relationship with smallest weight coming from that node is selected and added to the tree (along with its connected node).
- This process is repeated, always choosing the minimal-weight relationship that joins any node not already in the tree.
- When there are no more nodes to add, the tree is a minimum spanning tree.



Prim's MST: Pseudo-code

```
PRIM(G, w, r)
  Q = ∅
  for each u ∈ G.V
        u.key = ∞
        u.π = NIL
        INSERT(Q, u)
  DECREASE-KEY(Q, r, 0)        // r.key = 0
  while Q ≠ ∅
        u = EXTRACT-MIN(Q)
        for each v ∈ G.Adj[u]
              if v ∈ Q and w(u, v) < v.key
                    v.π = u
                    DECREASE-KEY(Q, v, w(u, v))
```

Example based on CLRS book.

When to use MST

- Use Minimum Spanning Tree when you need the best route to visit all nodes. Because the route is chosen based on the cost of each next step, it's useful when you must visit all nodes in a single walk.
- It's also employed to approximate some problems with unknown compute times, such as the Traveling Salesman Problem.
- Although it may not always find the absolute optimal solution, this algorithm makes potentially complicated and compute-intensive analysis much more approachable.

All Pairs Shortest Path

- The All Pairs Shortest Path (APSP) algorithm calculates the shortest (weighted) path between all pairs of nodes. It's more efficient than running the Single Source Shortest Path algorithm for every pair of nodes in the graph.
- APSP optimizes operations by keeping track of the distances calculated so far and running on nodes in parallel. Those known distances can then be reused when calculating the shortest path to an unseen node.
- All Pairs Shortest Path is commonly used for understanding alternate routing when the shortest route is blocked or becomes suboptimal. For example, this algorithm is used in logical route planning to ensure the best multiple paths for diversity routing.

Centrality Algorithms

- Centrality algorithms are used to understand the roles of particular nodes in a graph and their impact on that network.
- They’re useful because they identify the most important nodes and help us understand group dynamics such as credibility, accessibility, the speed at which things spread, and bridges between groups.

Centrality Algorithms

Degree

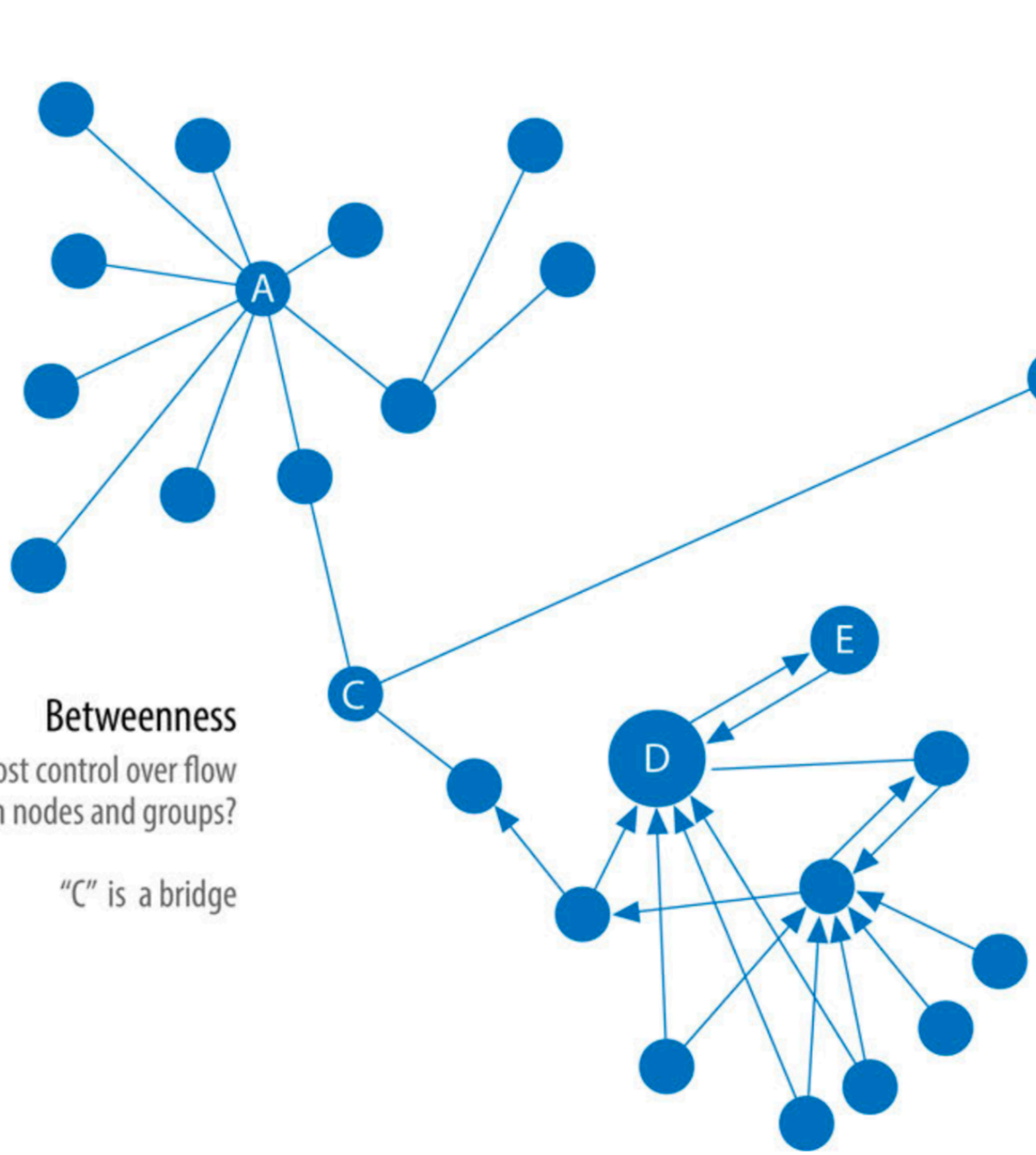
Number of connections?

"A" has a high degree

Betweenness

Which node has the most control over flow between nodes and groups?

"C" is a bridge



Closeness

Which node can most easily reach all other nodes in a graph or subgraph?

"B" is closest with the fewest hops in its subgraph

PageRank

Which node is the most important?

"D" is foremost based on number & weighting of in-links

"E" is next, due to the influence of D's link

From Needham and Hodler 2019, page 96

Centrality Algorithms

- **Degree centrality**: Measures the number of relationships a node has.
  - Use: Estimating a person's popularity by looking at their in-degree and using their out-degree to estimate gregariousness.

Centrality Algorithms

- **Degree centrality**: Measures the number of relationships a node has.
  - Use: Estimating a person's popularity by looking at their in-degree and using their out-degree to estimate gregariousness.
- **Closeness centrality**: Calculates which nodes have the shortest paths to all other nodes.
  - Use: Finding the optimal location of new public services for maximum accessibility.

Centrality Algorithms

- **Degree centrality**: Measures the number of relationships a node has.
  - Use: Estimating a person's popularity by looking at their in-degree and using their out-degree to estimate gregariousness.
- **Closeness centrality**: Calculates which nodes have the shortest paths to all other nodes.
  - Use: Finding the optimal location of new public services for maximum accessibility.
- **Betweenness centrality**: Measures the number of shortest paths that pass through a node.
  - Use: Improving drug targeting by finding the control genes for specific diseases

Centrality Algorithms

- **Degree centrality**: Measures the number of relationships a node has.
  - Use: Estimating a person's popularity by looking at their in-degree and using their out-degree to estimate gregariousness.
- **Closeness centrality**: Calculates which nodes have the shortest paths to all other nodes.
  - Use: Finding the optimal location of new public services for maximum accessibility.
- **Betweenness centrality**: Measures the number of shortest paths that pass through a node.
  - Use: Improving drug targeting by finding the control genes for specific diseases
- **Page rank**: Estimates a current node's importance from its linked neighbors and their neighbors.
  - Use: Finding the most influential features for extraction in machine learning

Degree Centrality

- It counts the number of incoming and outgoing relationships from a node, and is used to find popular nodes in a graph.
- The average degree of a network is simply the total number of relationships divided by the total number of nodes; it can be heavily skewed by high degree nodes.
- The degree distribution is the probability that a randomly selected node will have a certain number of relationships.
- Use Degree Centrality if you're attempting to analyze influence by looking at the number of incoming and outgoing relationships, or find the "popularity" of individual nodes.

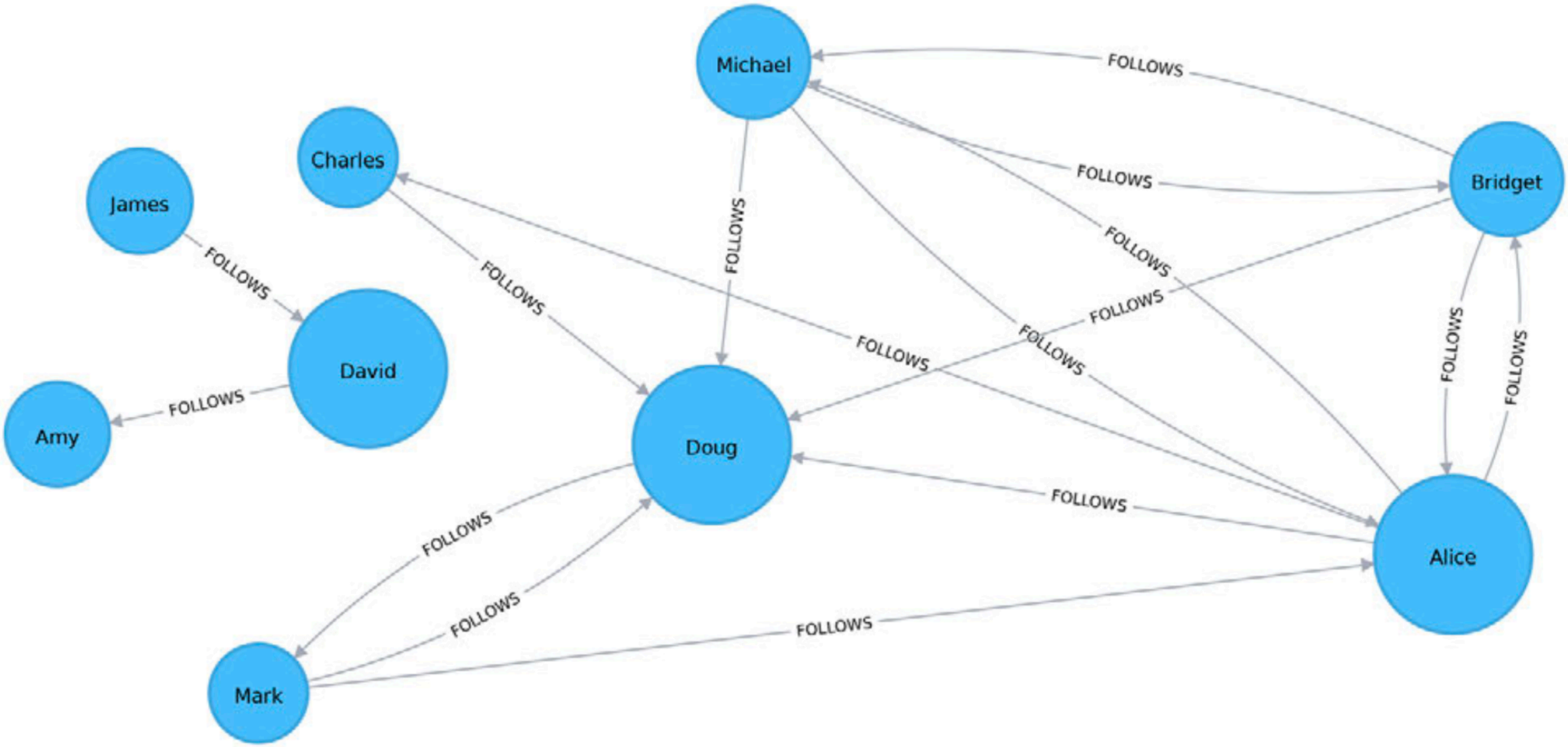
Closeness Centrality

- Closeness Centrality is a way of detecting nodes that are able to spread information efficiently through a subgraph.
- The measure of a node's centrality is its average farness (inverse distance) to all other nodes. Nodes with a high closeness score have the shortest distances from all other nodes.
- For each node, the Closeness Centrality algorithm calculates the sum of its distances to all other nodes, based on calculating the shortest paths between all pairs of nodes. The resulting sum is then inverted to determine the closeness centrality score for that node.

Normalized Closeness Centrality

Given a graph $G(V, E)$, where $|V| = n$ the normalized closeness centrality of a node $u \in V$ is defined as: $C_{norm}(u) = \frac{n-1}{\sum_{v=1}^{n-1} \delta(u,v)}$ where $\delta(u, v)$ is the shortest path between any $\{u, v\} \in V$

Closeness Centrality



Normalized Closeness: Alice = 1; David = 1; Doug = 1; Bridget = 0.714; Michael = 0.714; Amy = 0.666; James = 0.666; Charles = 0.625; Mark = 0.625

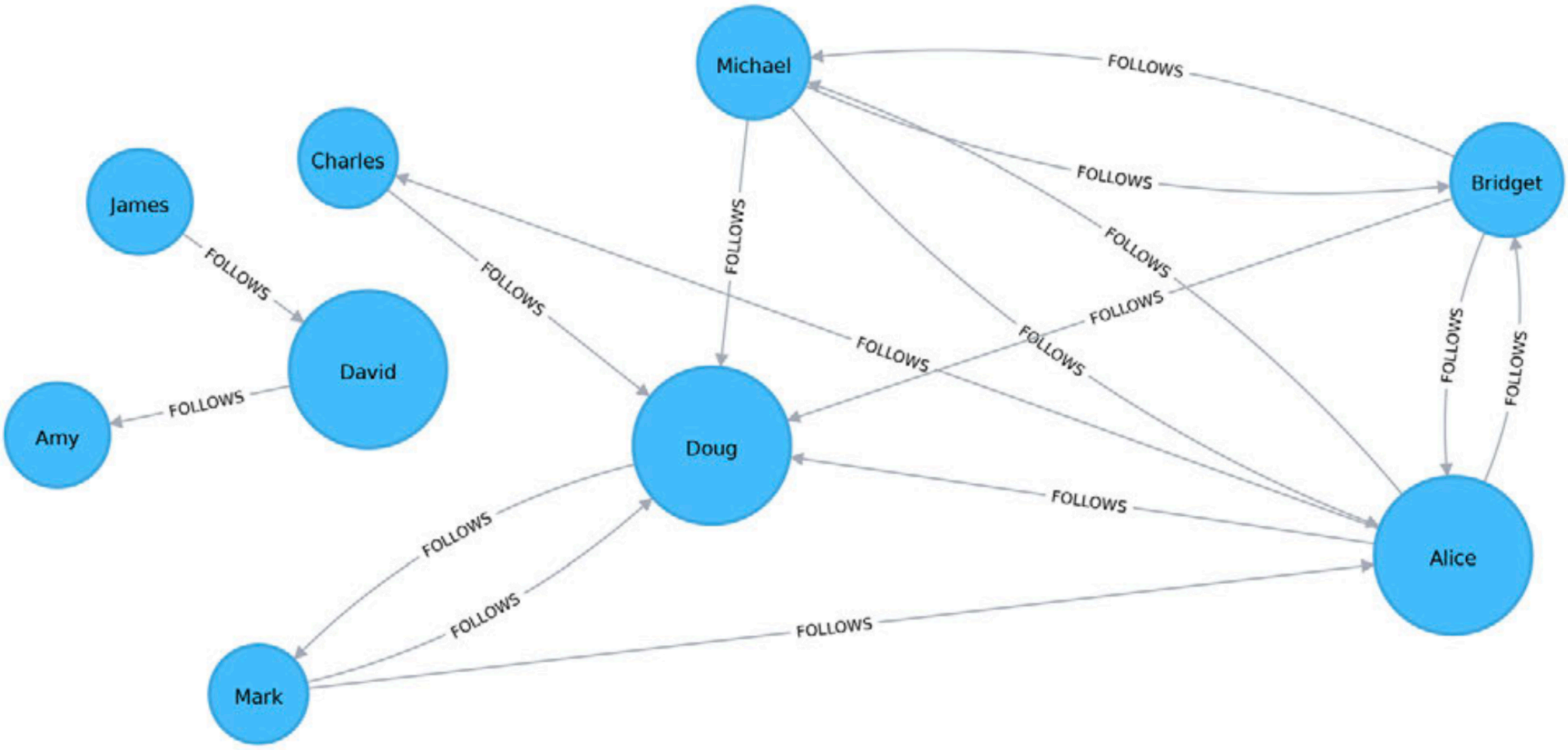
Note that Alice and David have the same closeness centrality

Wasserman and Faust Closeness Centrality

- Stanley Wasserman and Katherine Faust came up with an improved formula for calculating closeness for graphs with multiple subgraphs without connections between those groups.
- The result of this formula is a ratio of the fraction of nodes in the group that are reachable to the average distance from the reachable nodes.

Given a graph $G(V, E)$, where $|V| = N$ and a node $u \in V$ where n is the number of nodes in the same component as u, the WF closeness centrality of a node $u \in V$ is defined as: $C_{WF}(u) = \frac{n-1}{N-1}\left(\frac{n-1}{\sum_{v=1}^{n-1} \delta(u,v)}\right)$ where $\delta(u, v)$ is the shortest path between any $\{u, v\} \in V$

Wasserman and Faust Closeness Centrality



WF Closeness: Alice = 0.5 Doug= 0.5 Bridget=0.357 Michael=0.357 Charles= 0.3125 Mark= 0.3125 David= 0.125 Amy=0.0833 James=0.0833

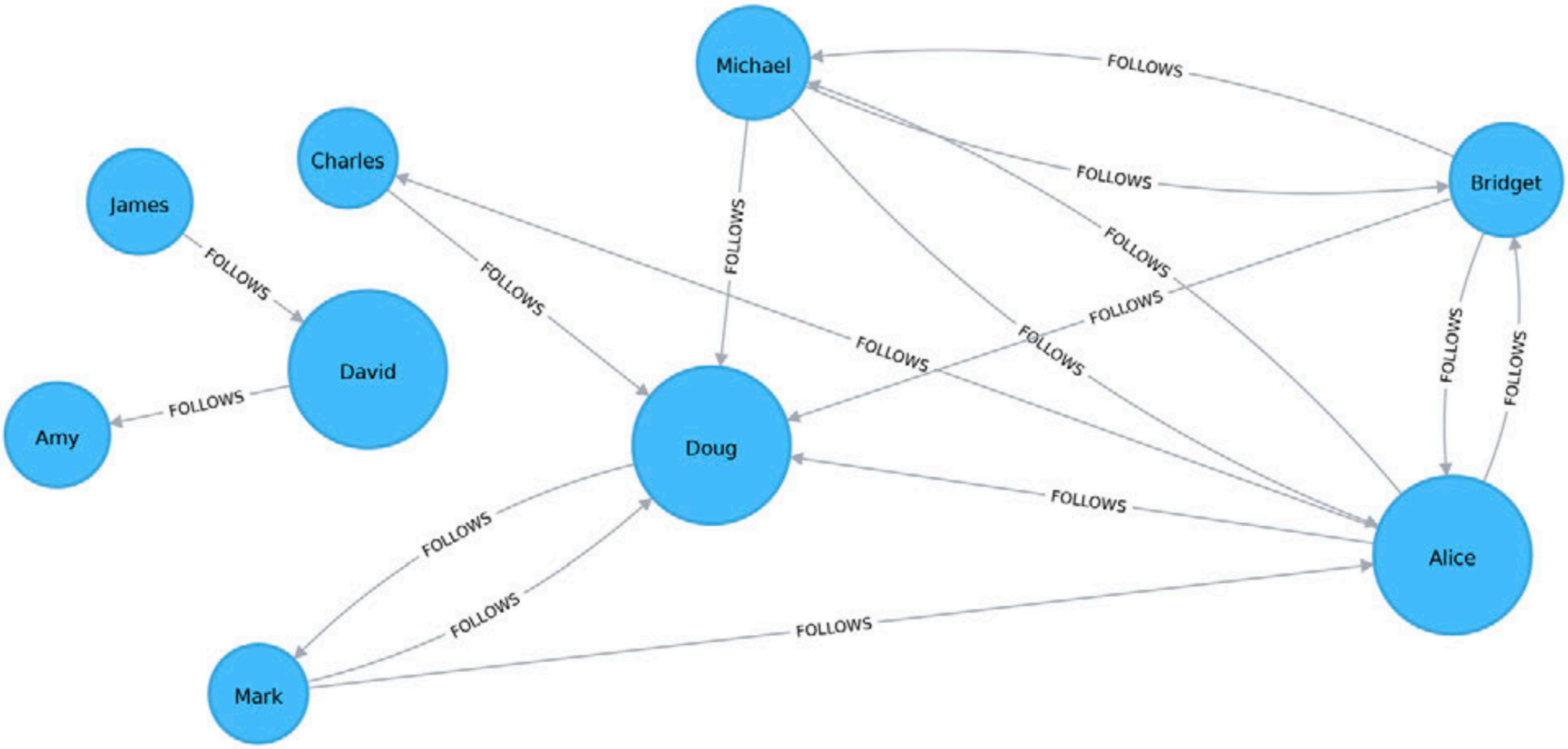
Note that Alice and David have very different closeness centrality

Harmonic Centrality

- Harmonic Centrality (also known as Valued Centrality) solves the original problem with unconnected graphs. In “Harmony in a Small World”, M. Marchiori and V. Latora proposed this concept as a practical representation of an average shortest path.

Given a graph $G(V, E)$, where $|V| = N$ the harmonic closeness centrality of a node $u \in V$ is defined as: $H(u) = \frac{\sum_{v=1}^{n-1} \frac{1}{\delta(u,v)}}{N-1}$ where $\delta(u, v)$ is the shortest path between any $\{u, v\} \in V$

Harmonic Centrality



Harmonic Closeness: Alice = 0.625 Doug= 0.625 Bridget=0.5 Michael=0.5 Charles= 0.437 Mark= 0.437 David= 0.25 Amy=0.1875 James=0.1875

The results from this algorithm differ from those of the original Closeness Centrality algorithm but are similar to those from the Wasserman and Faust formula.

Betweenness Centrality

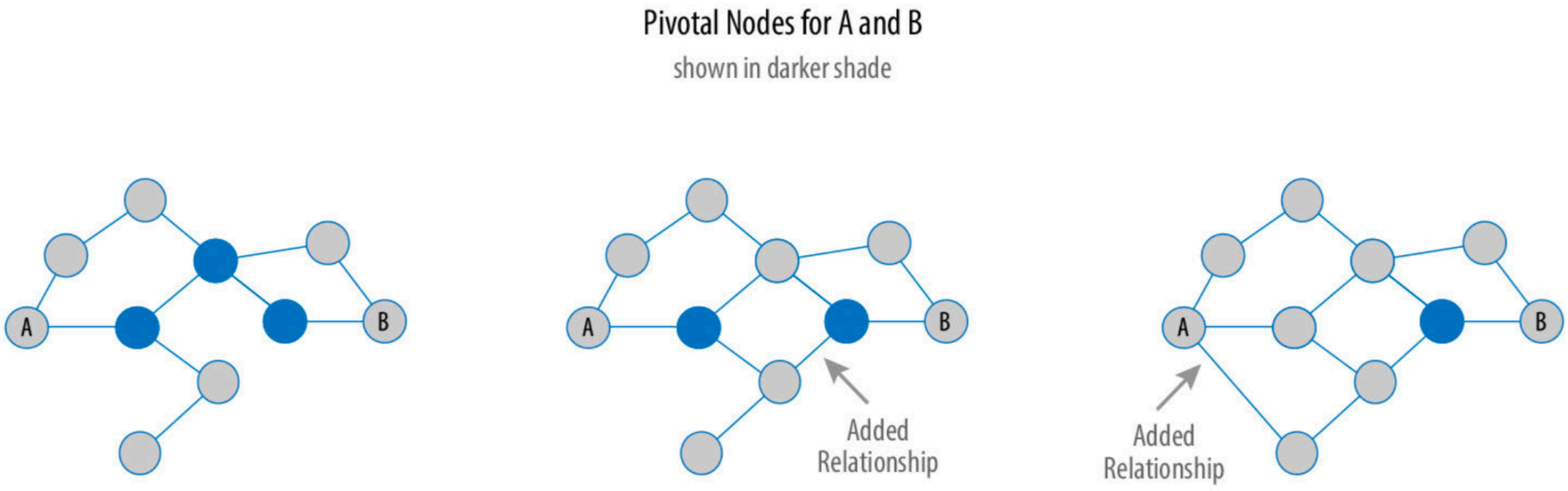
- Betweenness Centrality is a way of detecting the amount of influence a node has over the flow of information or resources in a graph.
- It is typically used to find nodes that serve as a bridge from one part of a graph to another.
- The Betweenness Centrality algorithm first calculates the shortest (weighted) path between every pair of nodes in a connected graph.
  - Each node receives a score, based on the number of these shortest paths that pass through the node.
  - The more shortest paths that a node lies on, the higher its score.

Betweenness Centrality

- A *bridge* in a graph can be a node or a relationship.
  - In a very simple graph, you can find them by looking for the node or relationship that, if removed, would cause a section of the graph to become disconnected.
- A node is considered *pivotal* for two other nodes if it lies on every shortest path between those nodes
- Pivotal nodes play an important role in connecting other nodes—if you remove a pivotal node, the new shortest path for the original node pairs will be longer or more costly. This can be a consideration for evaluating single points of vulnerability.

Betweenness Centrality

- A *bridge* in a graph can be a node or a relationship.
  - In a very simple graph, you can find them by looking for the node or relationship that, if removed, would cause a section of the graph to become disconnected.
- A node is considered *pivotal* for two other nodes if it lies on every shortest path between those nodes
-



Risk mitigation

Betweenness Centrality

Given a graph $G(V, E)$, where $|V| = N$ the betweenness centrality of a node $u \in V$ is defined as: $B(u) = \sum_{s \neq u \neq t} \frac{p(u)}{p}$ where $p(u)$ is the number of shortest paths between $\{s, t\} \in V$ passing through u and p is the total number of shortest path between $\{s, t\} \in V$.

- For each node, find the shortest paths that go through it.
- For each shortest path in step 1, calculate its percentage of the total possible shortest paths for that pair.
- Add together all the values in step 2 to find a node's betweenness centrality score.
- Repeat the process for each node.

Betweenness Centrality

- Uses:
  - Identifying influencers in various organizations. Powerful individuals are not necessarily in management positions, but can be found in “brokerage positions” using Betweenness Centrality. Removal of such influencers can seriously destabilize the organization.
  - Uncovering key transfer points in networks such as electrical grids. Counterintuitively, removal of specific bridges can actually improve overall robustness by “islanding” disturbances.
  - Helping microbloggers spread their reach on Twitter, with a recommendation engine for targeting influencers.

Page Rank

- PageRank is the best known of the centrality algorithms. It measures the transitive (or directional) influence of nodes.
- All the other centrality algorithms we discuss measure the direct influence of a node, whereas PageRank considers the influence of a node's neighbors, and their neighbors.
- PageRank is named after Google cofounder Larry Page, who created it to rank websites in Google's search results. The basic assumption is that a page with more incoming and more influential incoming links is more likely a credible source.
  - PageRank measures the number and quality of incoming relationships to a node to determine an estimation of how important that node is. Nodes with more sway over a network are presumed to have more incoming relationships from other influential nodes.

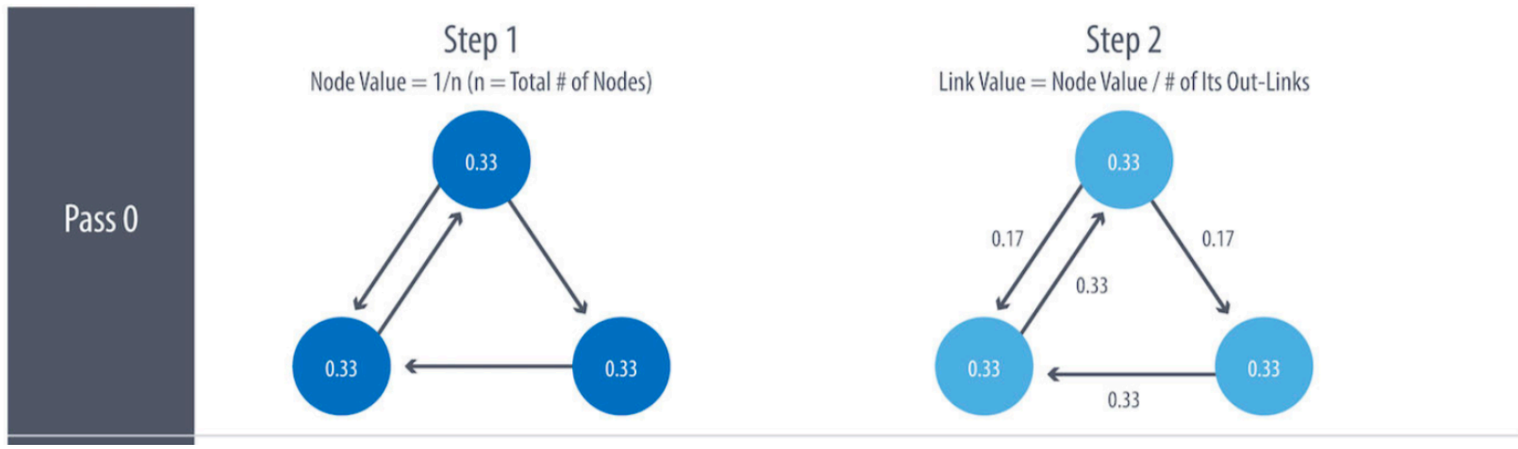
Page Rank

$$PR(u) = (1-d) + d\left(\frac{PR(T1)}{C(T1)} + \ldots + \frac{PR(Tn)}{C(Tn)}\right)$$

- We assume that a page *u* has citations from pages *T1* to *Tn.*
- *d* is a dumping factor which is set between 0 and 1 (usually set to 0.85). This can be seen as the probability that a user will continue clicking.
- *1-d* is the probability that a node is reached directly without following any relationships.
- *C(Tn)* is defined as the out-degree of a node *T*.

Page Rank

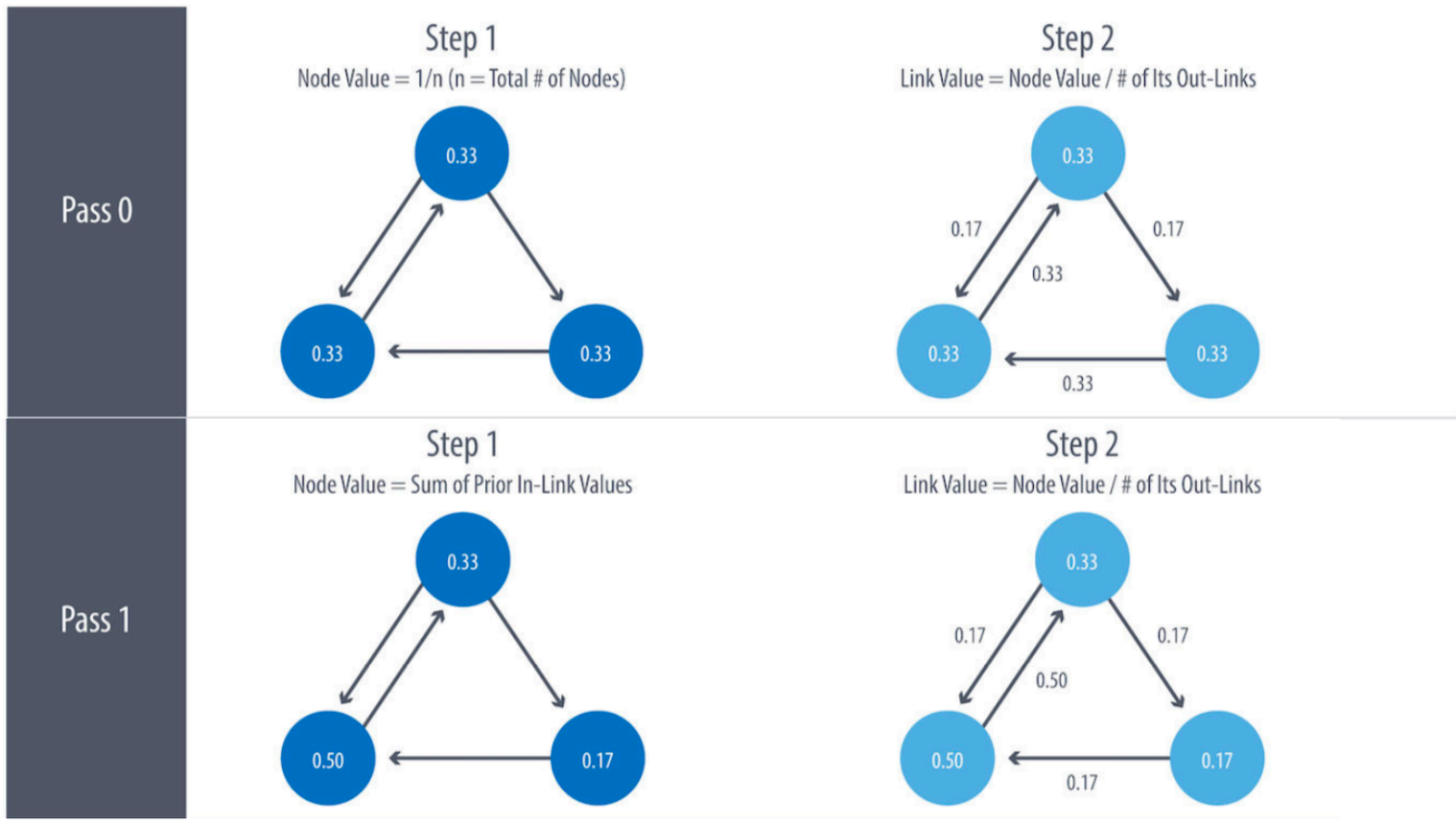
- PageRank will continue to update the rank of a node until it converges or meets the set number of iterations



Reference: Chapter 4
of Needham and Hodler, Graph algorithms, O'Reilly, 2019

Page Rank

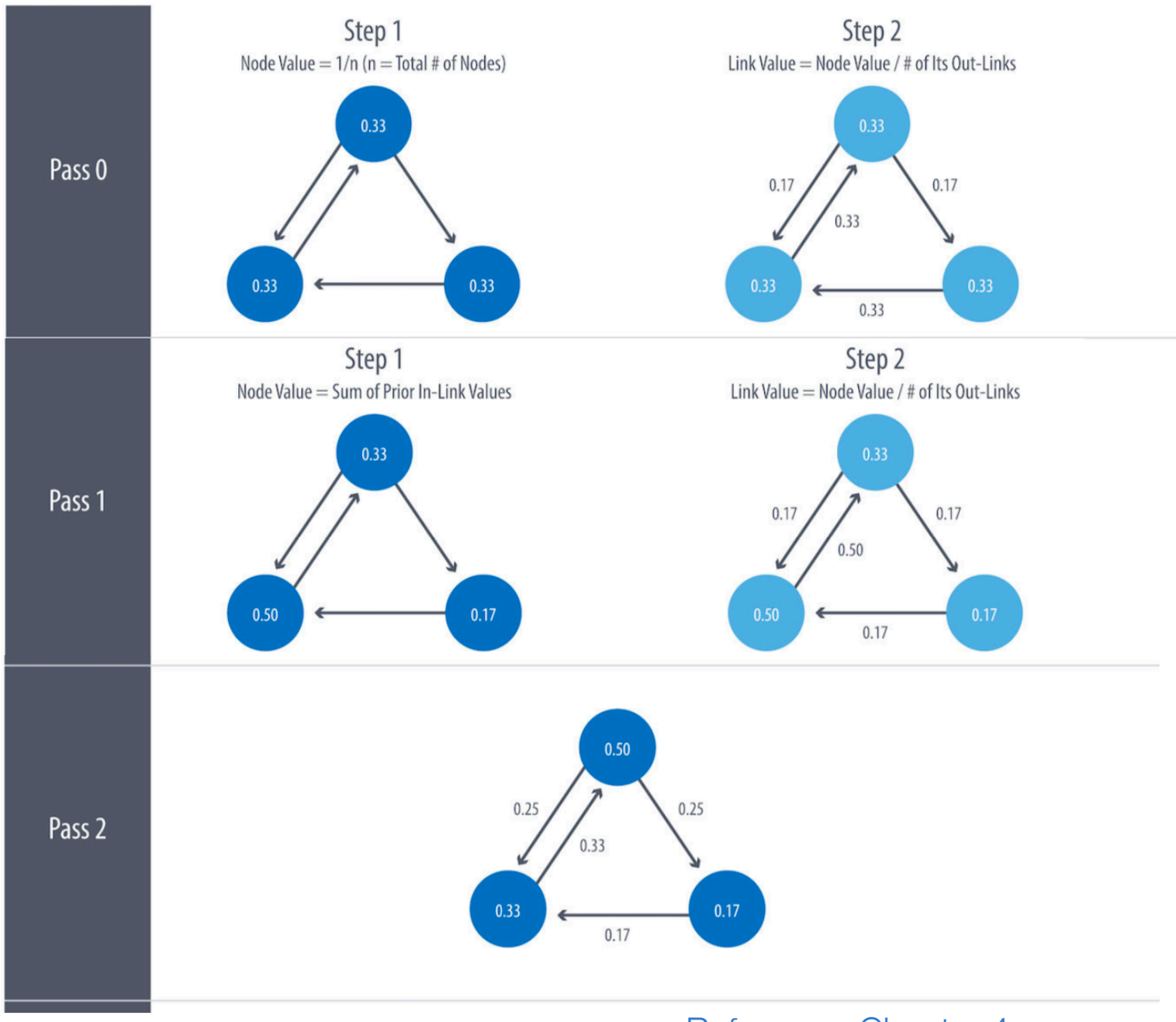
- PageRank will continue to update the rank of a node until it converges or meets the set number of iterations



Reference: Chapter 4
of Needham and Hodler, Graph algorithms, O'Reilly, 2019

Page Rank

- PageRank will continue to update the rank of a node until it converges or meets the set number of iterations



Reference: Chapter 4
of Needham and Hodler, Graph algorithms, O'Reilly, 2019

Page Rank

- Conceptually, PageRank assumes there is a web surfer visiting pages by following links or by using a random URL.
- The damping factor *d* defines the probability that the next click will be through a link. A PageRank score represents the likelihood that a page is visited through an incoming link and not randomly.
- A node, or group of nodes, without outgoing relationships can monopolize the PageRank score by refusing to share.
  - This is known as a rank sink: a surfer that gets stuck on a page, or a subset of pages, with no way out.
- Another difficulty is created by nodes that point only to each other in a group. Circular references cause an increase in their ranks as the surfer bounces back and forth among the nodes.

Page Rank

- There are two strategies used to avoid rank sinks.
  - When a node is reached that has no outgoing relationships, PageRank assumes outgoing relationships to all nodes. Traversing these invisible links is sometimes called teleportation.
  - The damping factor provides another opportunity to avoid sinks by introducing a probability for direct link versus random node visitation. When you set d to 0.85, a completely random node is visited 15% of the time.