CS 4102: Algorithms

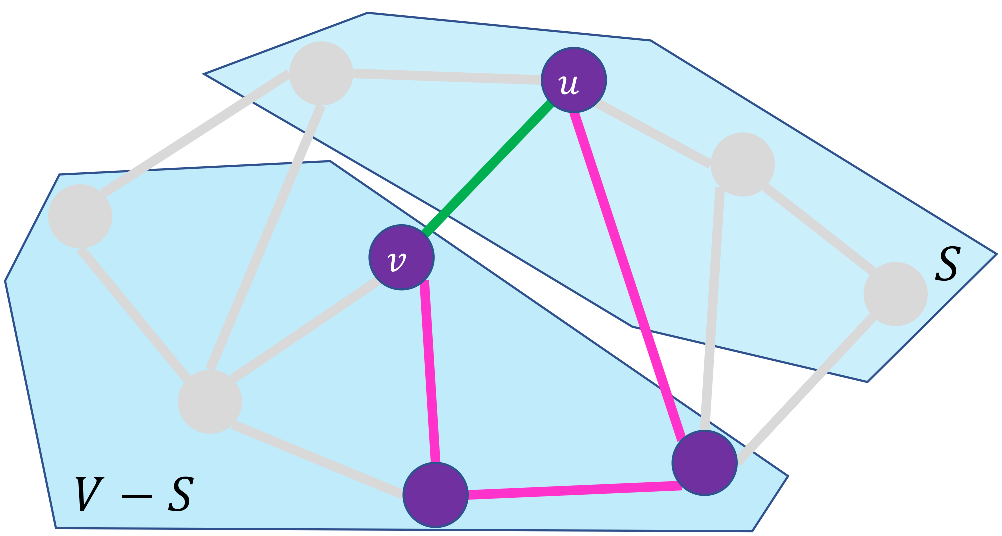
Shortest Path Algorithms

Tom Horton and Robbie Hott

Spring 2020

Warm-Up

Show that no cycle crosses a cut exactly once



- Consider an edge $e = (u, v)$ that crosses the cut
- After removing the edge e from the graph, there is still a path from $u \in S$ to $v \notin S$
- At least one edge along the path from cross the cut

Today’s Keywords

Graphs

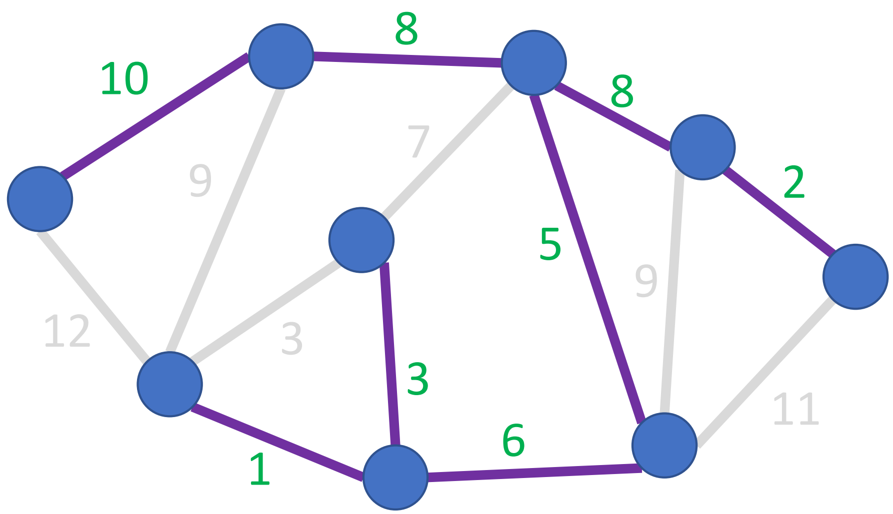
Shortest paths algorithms

Dijkstra’s algorithm

Breadth-first search (BFS)

CLRS Readings: Chapter 22, 23

Minimum Spanning Tree

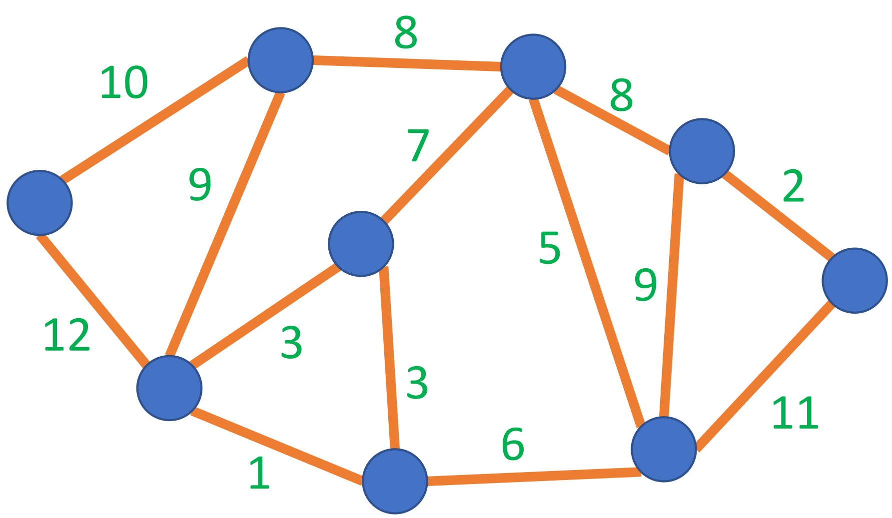


$$\text{Cost}(T) = \sum_{e \in E_T} w(e)$$

A tree $T = (V_T, E_T)$ is a **minimum spanning tree** for an undirected graph $G = (V, E)$ if T is a spanning tree of minimal cost

Minimum Spanning Tree

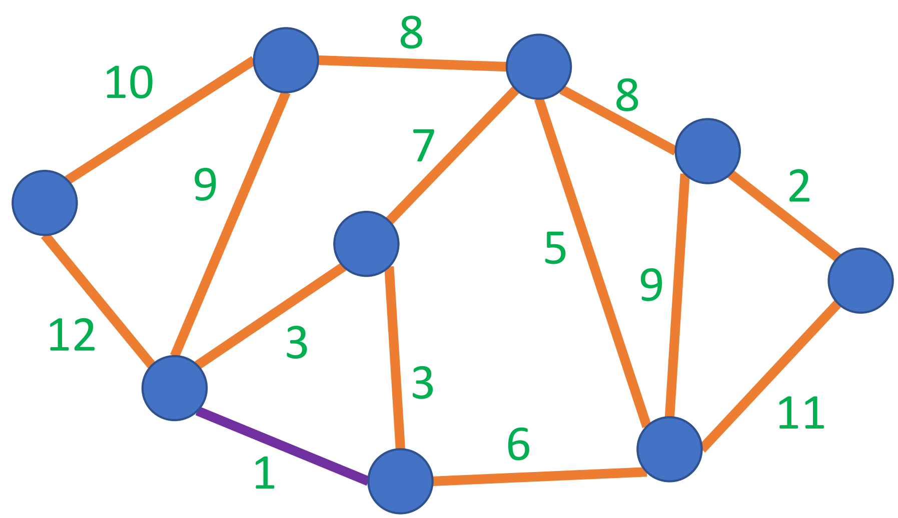
Reminder: **Kruskal's** is the first of two greedy algorithms!



Kruskal: add <u>minimum-weight edge</u> that does not introduce a cycle

Minimum Spanning Tree

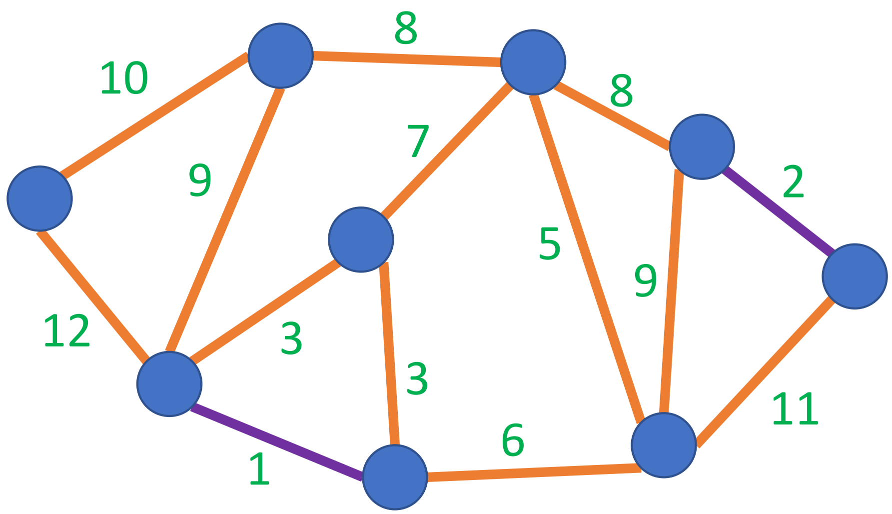
Two greedy algorithms:



Kruskal: add minimum-weight edge that does not introduce a cycle

Minimum Spanning Tree

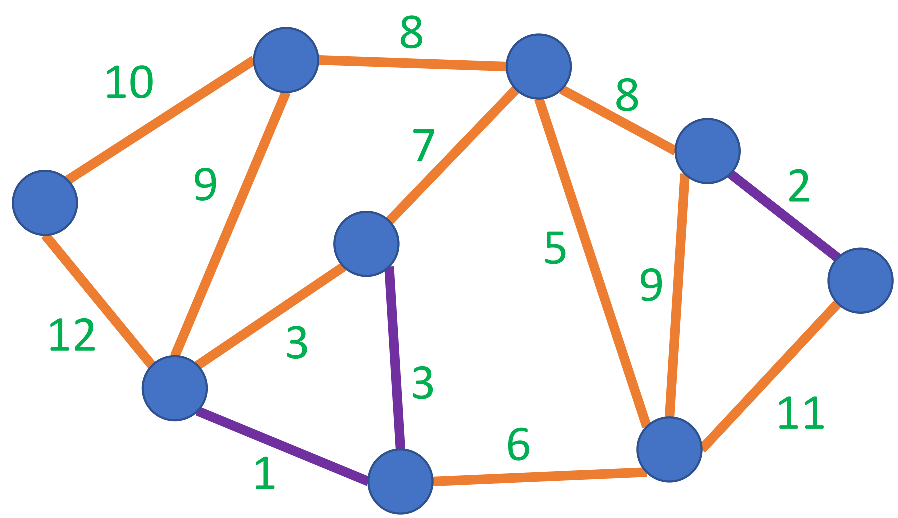
Reminder: **Kruskal's** is the first of two greedy algorithms!



Kruskal: add minimum-weight edge that does not introduce a cycle

Minimum Spanning Tree

Reminder: **Kruskal's** is the first of two greedy algorithms!

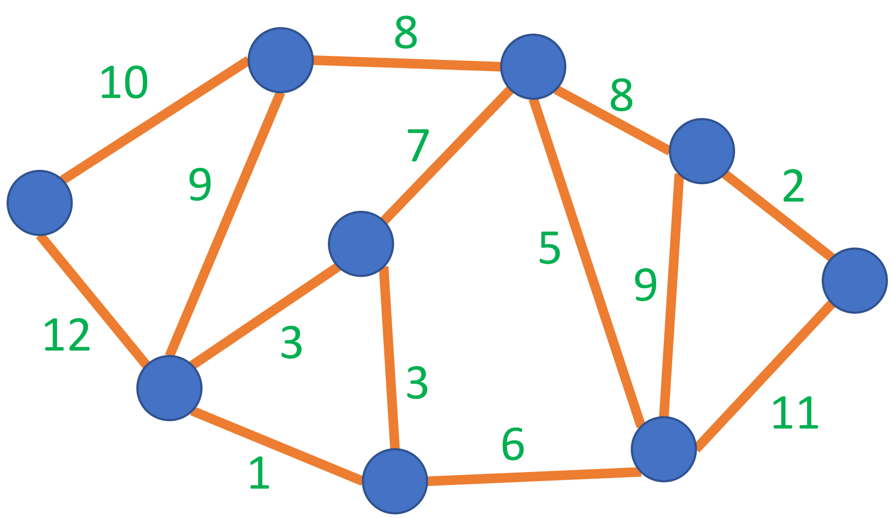


And so on…
See previous lecture slides

Kruskal: add <u>minimum-weight edge</u> that does not introduce a cycle

Minimum Spanning Tree

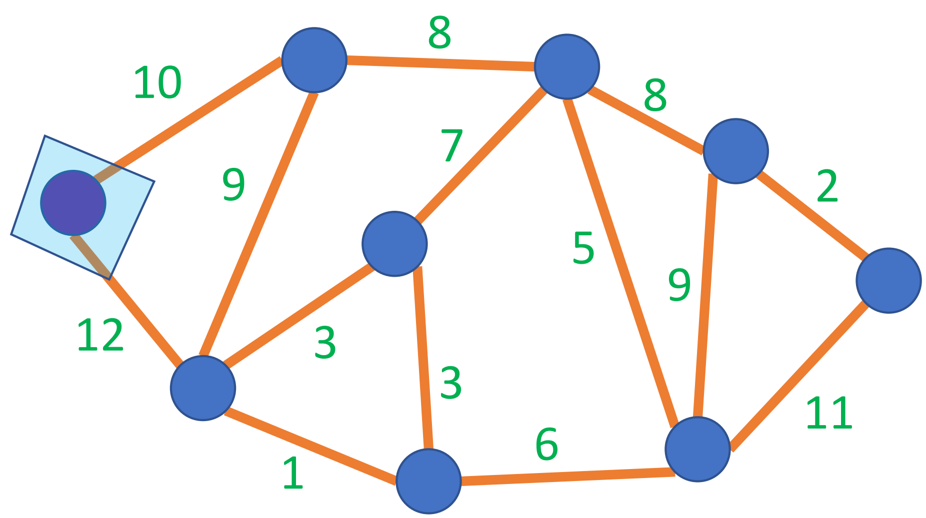
Reminder: **Prim's** is the second of two greedy algorithms!



Prim: "grow" a tree by adding <u>minimum-weight edge</u>
from the tree to an external node

Minimum Spanning Tree

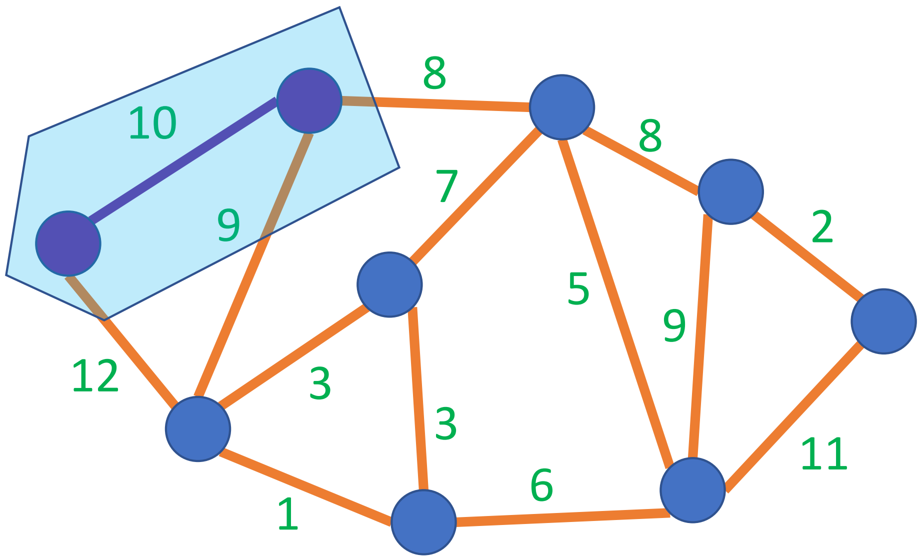
Reminder: **Prim's** is the second of two greedy algorithms!



Prim: "grow" a tree by adding <u>minimum-weight edge</u>
from the tree to an external node

Minimum Spanning Tree

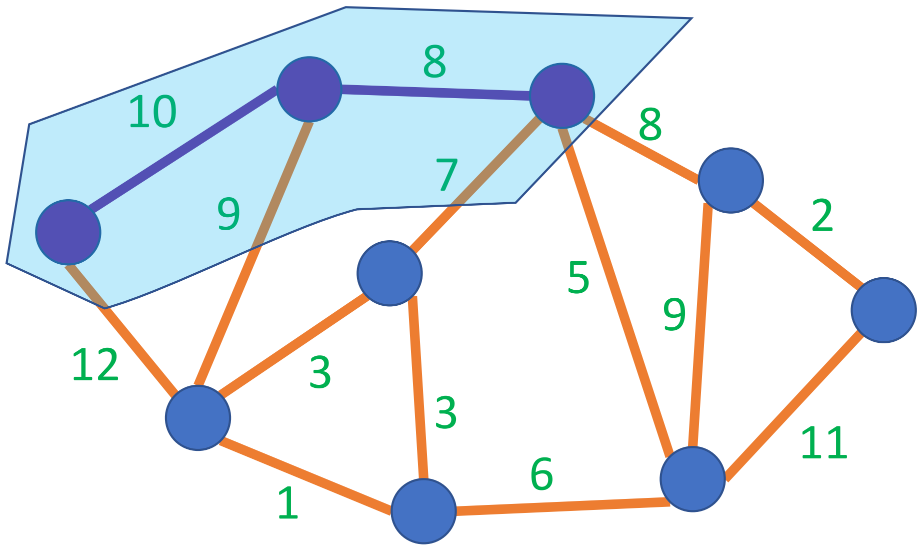
Reminder: **Prim's** is the second of two greedy algorithms!



Prim: "grow" a tree by adding minimum-weight edge from the tree to an external node

Minimum Spanning Tree

Reminder: **Prim’s** is the second of two greedy algorithms!



And so on…
See previous
lecture slides

Prim: “grow” a tree by adding minimum-weight edge
from the tree to an external node

Prim's Algorithm Implementation

1. Start with an empty tree T and pick a start node and add it to T
2. Repeat $|V| - 1$ times:
   - Add the min-weight edge which connects a node in T with a node not in T

Implementation (with nodes in the priority queue):

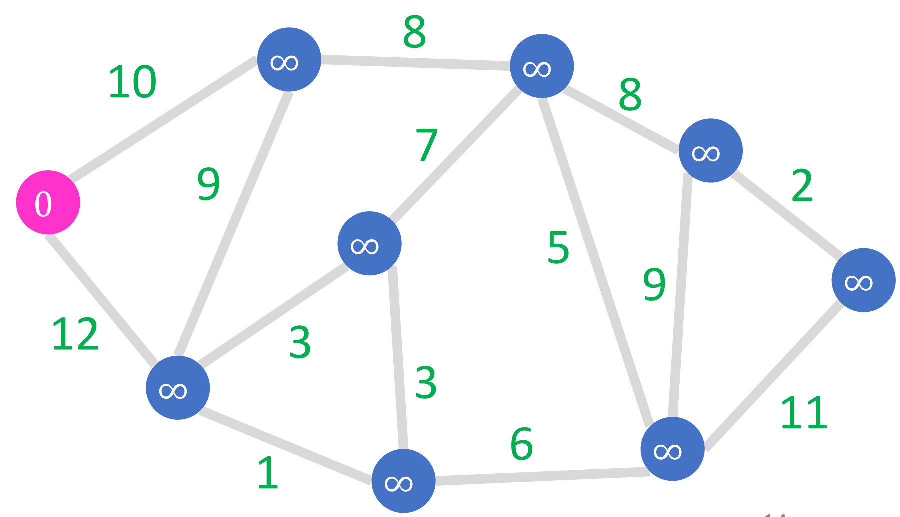
each node also maintains a parent, initially `NULL`

initialize $d_v = \infty$ for each node v
add all nodes $v \in V$ to the priority queue PQ, using d_v as the key
pick a starting node s and set $d_s = 0$
while PQ is not empty:
    $v = \mathrm{PQ}.\mathrm{extractMin}()$
    for each $u \in V$ such that $(v, u) \in E$:
        if $u \in \mathrm{PQ}$ and $w(v, u) < d_u$:
            $\mathrm{PQ}.\mathrm{decreaseKey}(u, w(v, u))$
            $u.\mathrm{parent} = v$

Prim's Algorithm Implementation

Implementation (with nodes in the priority queue):

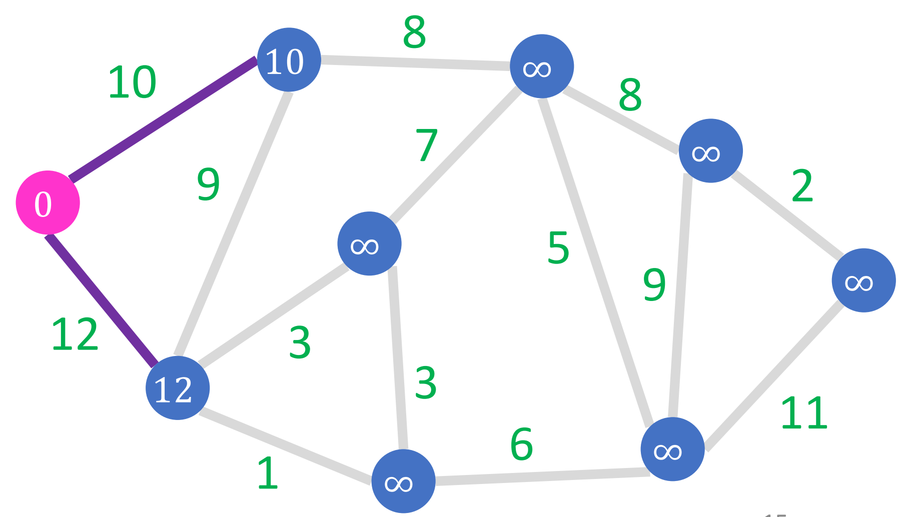
initialize $d_v = \infty$ for each node v
add all nodes $v \in V$ to the priority queue PQ, using d_v as the key
pick a starting node s and set $d_s = 0$
while PQ is not empty:
  $v = \mathrm{PQ}.\mathrm{extractMin}()$
  for each $u \in V$ such that $(v, u) \in E$:
    if $u \in \mathrm{PQ}$ and $w(v, u) < d_u$:
      $\mathrm{PQ}.\mathrm{decreaseKey}\big(u, w(v, u)\big)$
      $u.\mathrm{parent} = v$



Prim's Algorithm Implementation

Implementation (with nodes in the priority queue):

initialize $d_v = \infty$ for each node v
add all nodes $v \in V$ to the priority queue PQ, using d_v as the key
pick a starting node s and set $d_s = 0$
while PQ is not empty:
    $v = \mathrm{PQ}.\mathrm{extractMin}()$
    for each $u \in V$ such that $(v, u) \in E$:
        if $u \in \mathrm{PQ}$ and $w(v, u) < d_u$:
            $\mathrm{PQ}.\mathrm{decreaseKey}\big(u, w(v, u)\big)$
            $u.\mathrm{parent} = v$



Prim's Algorithm Implementation

Implementation (with nodes in the priority queue):

initialize $d_v = \infty$ for each node v

add all nodes $v \in V$ to the priority queue PQ, using d_v as the key

pick a starting node s and set $d_s = 0$

while PQ is not empty:

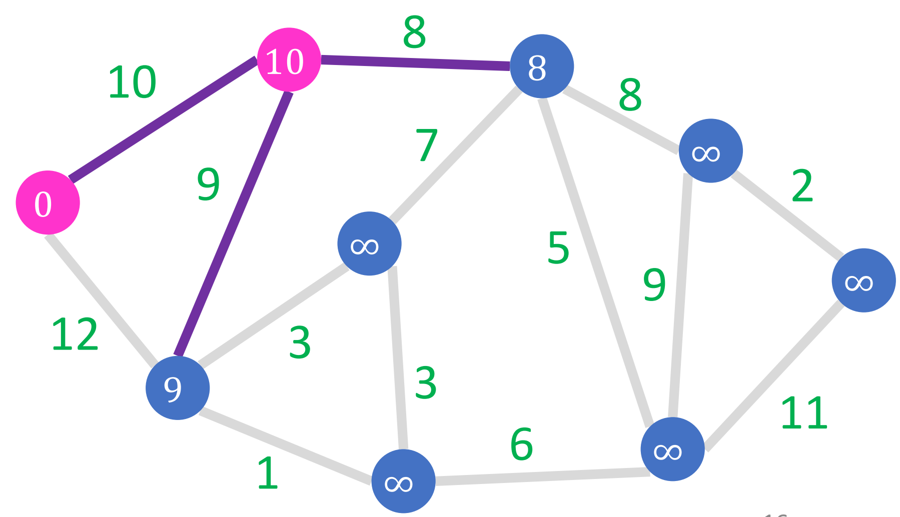
  $v = \mathrm{PQ.extractMin}()$

  for each $u \in V$ such that $(v, u) \in E$:

    if $u \in \mathrm{PQ}$ and $w(v, u) < d_u$:

      $\mathrm{PQ.decreaseKey}(u, w(v, u))$

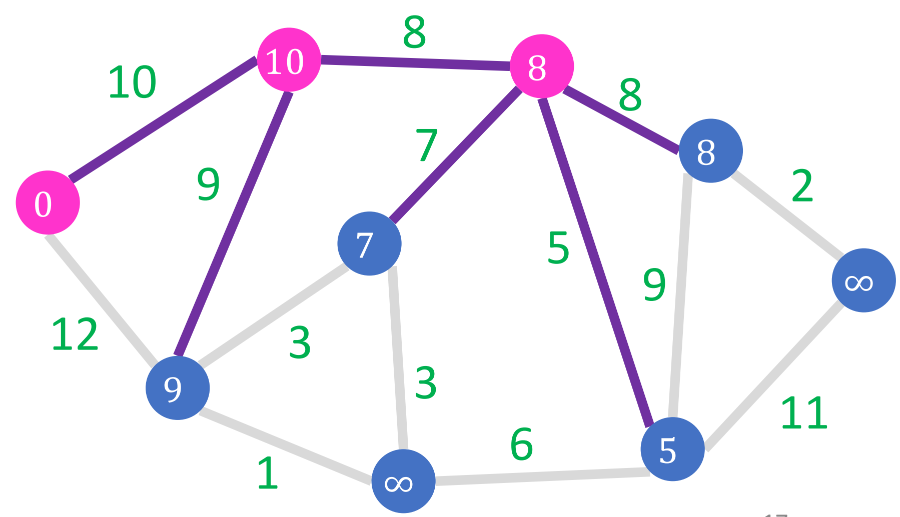
      $u.\mathrm{parent} = v$



Prim's Algorithm Implementation

Implementation (with nodes in the priority queue):

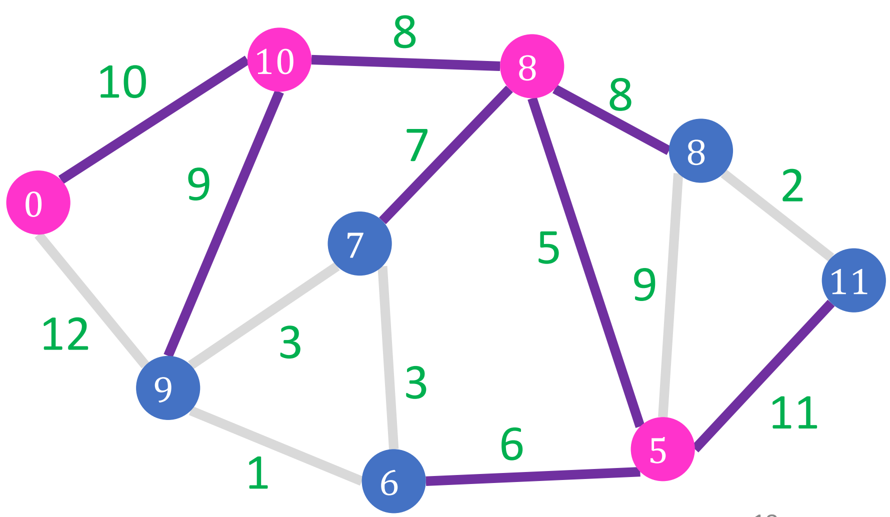
initialize $d_v = \infty$ for each node v
add all nodes $v \in V$ to the priority queue PQ, using d_v as the key
pick a starting node s and set $d_s = 0$
while PQ is not empty:
    $v = \mathrm{PQ.extractMin}()$
    for each $u \in V$ such that $(v, u) \in E$:
        if $u \in \mathrm{PQ}$ and $w(v, u) < d_u$:
            $\mathrm{PQ.decreaseKey}(u, w(v, u))$
            $u.\mathrm{parent} = v$



Prim's Algorithm Implementation

Implementation (with nodes in the priority queue):

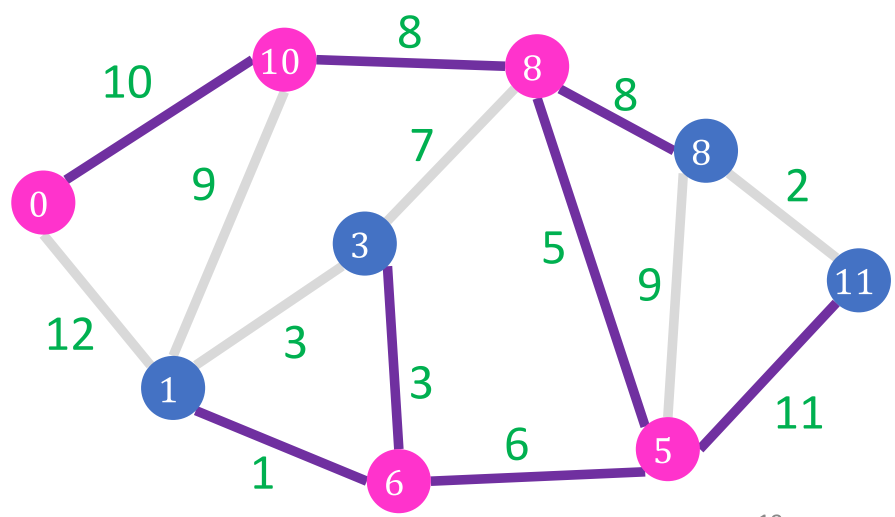
initialize $d_v = \infty$ for each node v
add all nodes $v \in V$ to the priority queue PQ, using d_v as the key
pick a starting node s and set $d_s = 0$
while PQ is not empty:
  $v = \mathrm{PQ}.\mathrm{extractMin}()$
  for each $u \in V$ such that $(v, u) \in E$:
    if $u \in \mathrm{PQ}$ and $w(v, u) < d_u$:
      $\mathrm{PQ}.\mathrm{decreaseKey}\big(u, w(v, u)\big)$
      $u.\mathrm{parent} = v$



Prim's Algorithm Implementation

Implementation (with nodes in the priority queue):

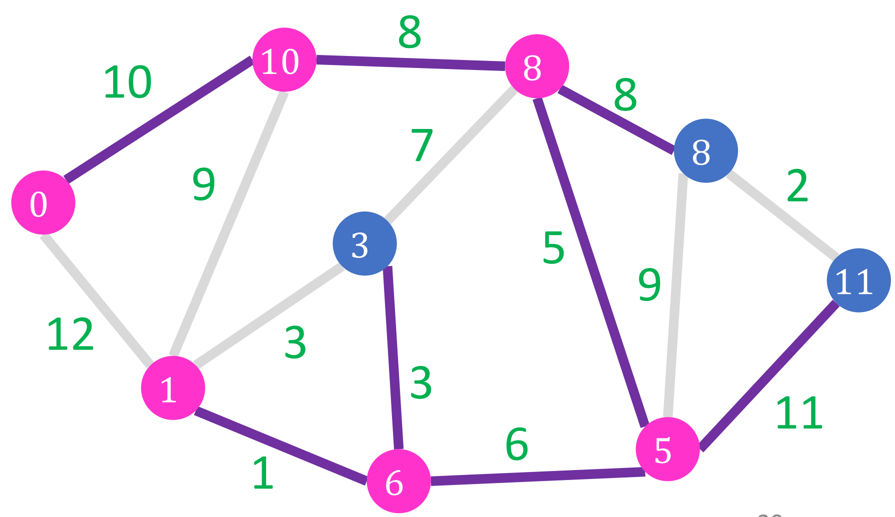
initialize $d_v = \infty$ for each node v
add all nodes $v \in V$ to the priority queue PQ, using d_v as the key
pick a starting node s and set $d_s = 0$
while PQ is not empty:
    $v = \mathrm{PQ.extractMin}()$
    for each $u \in V$ such that $(v, u) \in E$:
        if $u \in \mathrm{PQ}$ and $w(v, u) < d_u$:
            $\mathrm{PQ.decreaseKey}(u, w(v, u))$
            $u.\mathrm{parent} = v$



Prim's Algorithm Implementation

Implementation (with nodes in the priority queue):

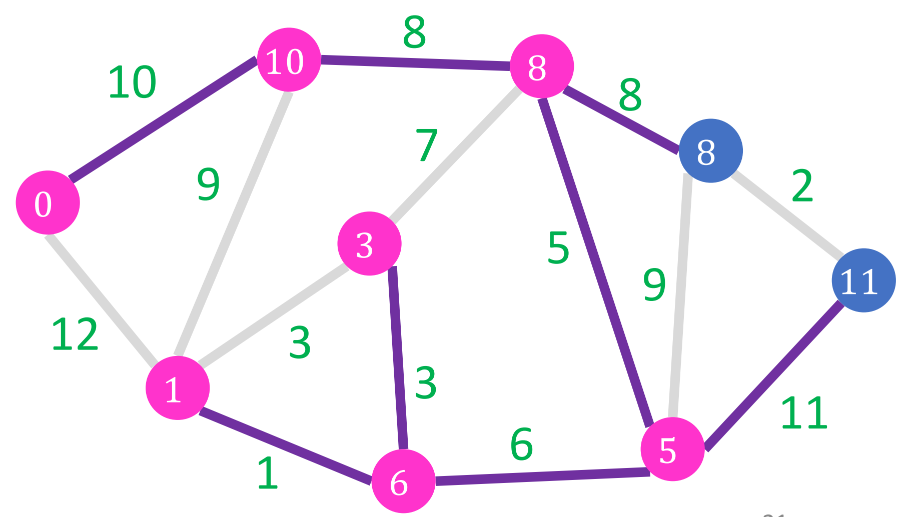
initialize $d_v = \infty$ for each node v
add all nodes $v \in V$ to the priority queue PQ, using d_v as the key
pick a starting node s and set $d_s = 0$
while PQ is not empty:
    $v = \mathrm{PQ}.\mathrm{extractMin}()$
    for each $u \in V$ such that $(v, u) \in E$:
        if $u \in \mathrm{PQ}$ and $w(v, u) < d_u$:
            $\mathrm{PQ}.\mathrm{decreaseKey}\big(u, w(v, u)\big)$
            $u.\mathrm{parent} = v$



Prim's Algorithm Implementation

Implementation (with nodes in the priority queue):

initialize $d_v = \infty$ for each node v
add all nodes $v \in V$ to the priority queue PQ, using d_v as the key
pick a starting node s and set $d_s = 0$
while PQ is not empty:
  $v = \text{PQ.extractMin}()$
  for each $u \in V$ such that $(v, u) \in E$:
    if $u \in$ PQ and $w(v, u) < d_u$:
      $\text{PQ.decreaseKey}(u, w(v, u))$
      $u.\text{parent} = v$



Prim's Algorithm Implementation

Implementation (with nodes in the priority queue):

initialize $d_v = \infty$ for each node v

add all nodes $v \in V$ to the priority queue PQ, using d_v as the key

pick a starting node s and set $d_s = 0$

while PQ is not empty:

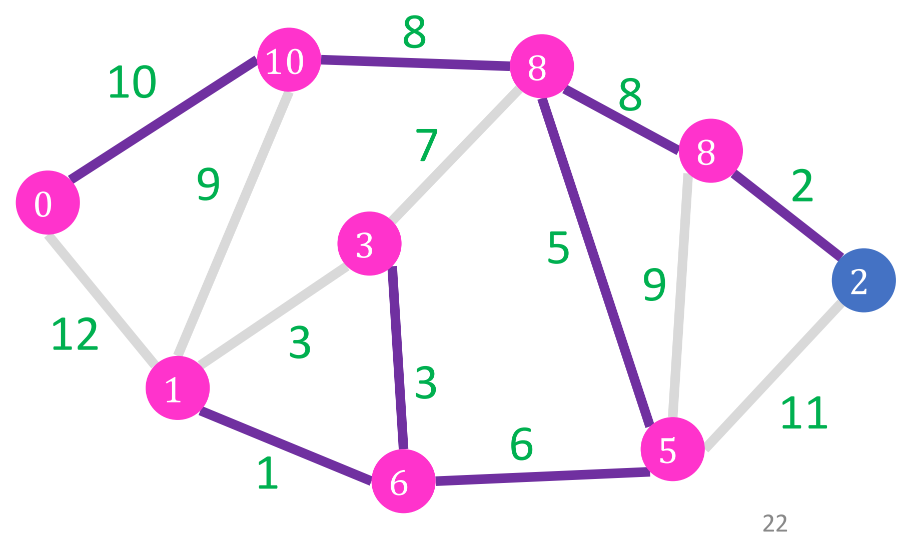
  $v = \mathrm{PQ}.\mathrm{extractMin}()$

  for each $u \in V$ such that $(v, u) \in E$:

    if $u \in \mathrm{PQ}$ and $w(v, u) < d_u$:

      $\mathrm{PQ}.\mathrm{decreaseKey}\big(u, w(v, u)\big)$

      $u.\mathrm{parent} = v$



Prim's Algorithm Implementation

Implementation (with nodes in the priority queue):

initialize $d_v = \infty$ for each node v

add all nodes $v \in V$ to the priority queue PQ, using d_v as the key

pick a starting node s and set $d_s = 0$

while PQ is not empty:

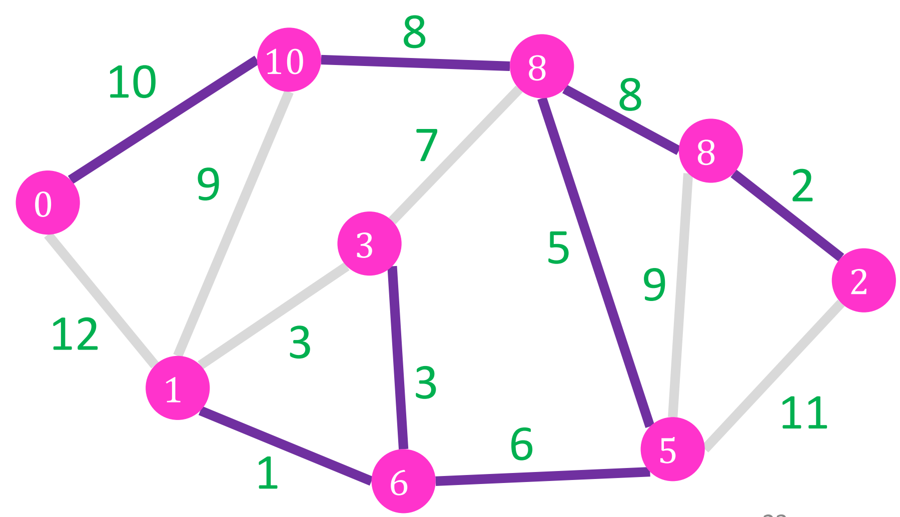
  $v = \mathrm{PQ}.\mathrm{extractMin}()$

  for each $u \in V$ such that $(v, u) \in E$:

    if $u \in \mathrm{PQ}$ and $w(v, u) < d_u$:

      $\mathrm{PQ}.\mathrm{decreaseKey}(u, w(v, u))$

      $u.\mathrm{parent} = v$



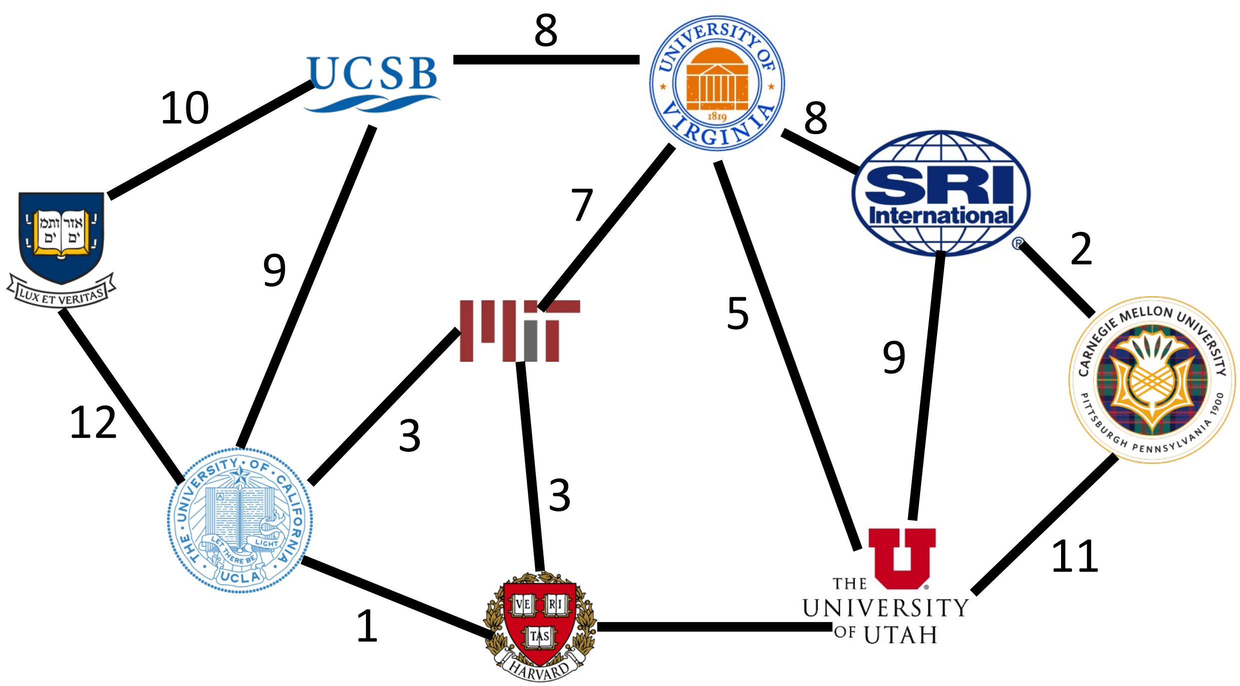
Prim's Algorithm Running Time

Implementation (with nodes in the priority queue):

initialize $d_v = \infty$ for each node v add all nodes $v \in V$ to the priority queue PQ, using d_v as the key pick a starting node s and set $d_s = 0$	Initialization: $O(\lvert V \rvert)$
while PQ is not empty:	$\lvert V \rvert$ iterations
$v = \mathrm{PQ.extractMin}()$	$O(\log\lvert V \rvert)$
for each $u \in V$ such that $(v, u) \in E$:	$\lvert E \rvert$ iterations total
if $u \in \mathrm{PQ}$ and $w(v, u) < d_u$:	
$\mathrm{PQ.decreaseKey}(u, w(v, u))$	$O(\log\lvert V \rvert)$
$u.\mathrm{parent} = v$	

Overall running time: $O(\lvert V \rvert \log\lvert V \rvert + \lvert E \rvert \log\lvert V \rvert) = O(\lvert E \rvert \log\lvert V \rvert)$

Single-Source Shortest Path



Find the <u>shortest path</u> from UVA to each of these other places

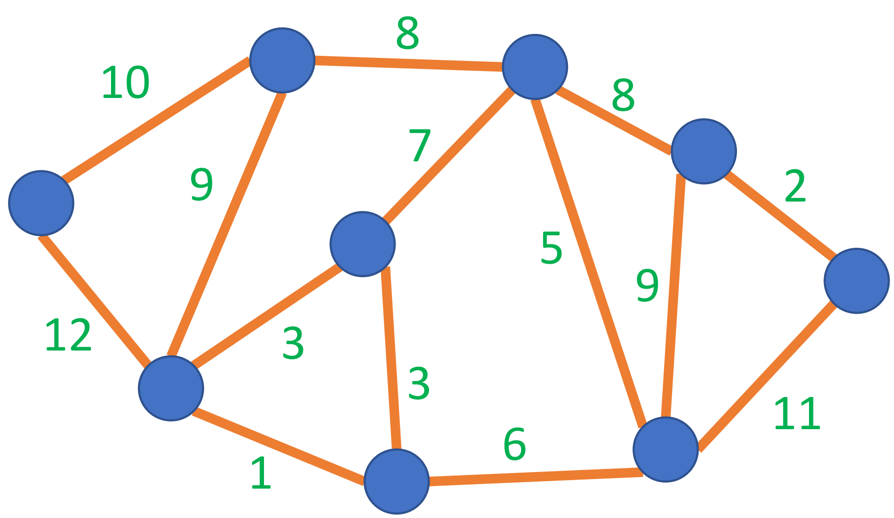
Given a graph $G = (V, E)$ and a start node (i.e., source) $s \in V$,

for each $v \in V$ find the minimum-weight path from $s \to v$ (call this weight $\delta(s, v)$)

Assumption (for now): all edge weights are positive

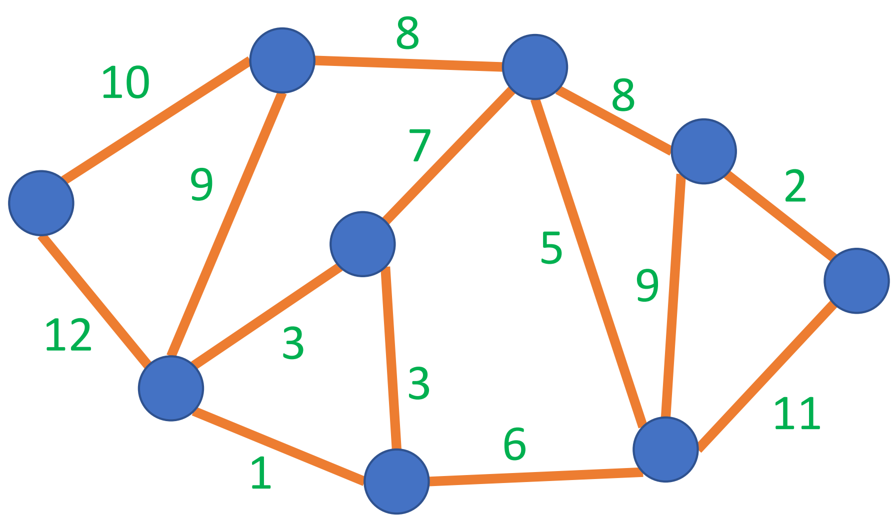
Dijkstra's SP Algorithm

1. Start with an empty tree T and add the source to T
2. Repeat $|V| - 1$ times:
   - Add the node nearest to the source that's not yet in T to T



Prim’s MST Algorithm

1. Start with an empty tree T and pick a start node and add it to T
2. Repeat $|V| - 1$ times:
   - Add the min-weight edge which connects a node in T with a node not in T



Prim's MST Algorithm Implementation

1. Start with an empty tree T and pick a start node and add it to T
2. Repeat $|V| - 1$ times:
   - Add the min-weight edge which connects a node in T with a node not in T

Implementation:

initialize $d_v = \infty$ for each node v
add all nodes $v \in V$ to the priority queue PQ, using d_v as the key
pick a starting node s and set $d_s = 0$
while PQ is not empty:
    $v = \mathrm{PQ.extractMin}()$
    for each $u \in V$ such that $(v, u) \in E$:
        if $u \in \mathrm{PQ}$ and $w(v, u) < d_u$:
            $\mathrm{PQ.decreaseKey}(u, w(v, u))$
            $u.\mathrm{parent} = v$

each node also maintains a parent, initially NULL

PQ's key: weight of a single connecting edge

Dijkstra's SP Algorithm Implementation

1. Start with an empty tree T and add the source to T
2. Repeat $|V| - 1$ times:
   - Add the "nearest" node not yet in T to T

Implementation:

initialize $d_v = \infty$ for each node v
add all nodes $v \in V$ to the priority queue PQ, using d_v as the key
set $d_s = 0$
while PQ is not empty:
  $v = \mathrm{PQ.extractMin}()$
  for each $u \in V$ such that $(v, u) \in E$:
    if $u \in \mathrm{PQ}$ and $d_v + w(v, u) < d_u$:
      $\mathrm{PQ.decreaseKey}(u, d_v + w(v, u))$
      $u.\mathrm{parent} = v$

each node also maintains a parent, initially `NULL`

PQ's key: length of shortest path $s \to u$ using nodes in PQ

Dijkstra's Algorithm Implementation

Implementation:

initialize $d_v = \infty$ for each node v

add all nodes $v \in V$ to the priority queue PQ, using d_v as the key

set $d_s = 0$

while PQ is not empty:

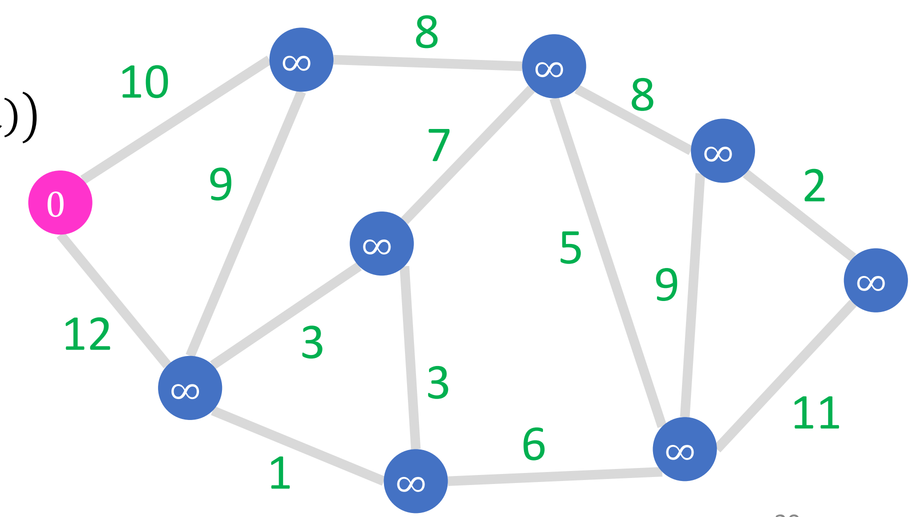
  $v = \text{PQ.extractMin}()$

  for each $u \in V$ such that $(v, u) \in E$:

    if $u \in$ PQ and $d_v + w(v, u) < d_u$:

      $\text{PQ.decreaseKey}\big(u, d_v + w(v, u)\big)$

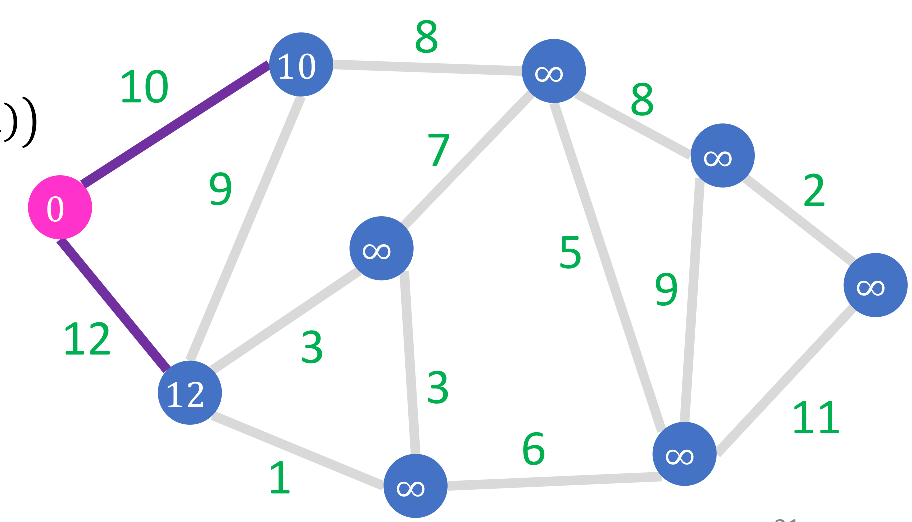
      $u.\text{parent} = v$



Dijkstra's Algorithm Implementation

Implementation:

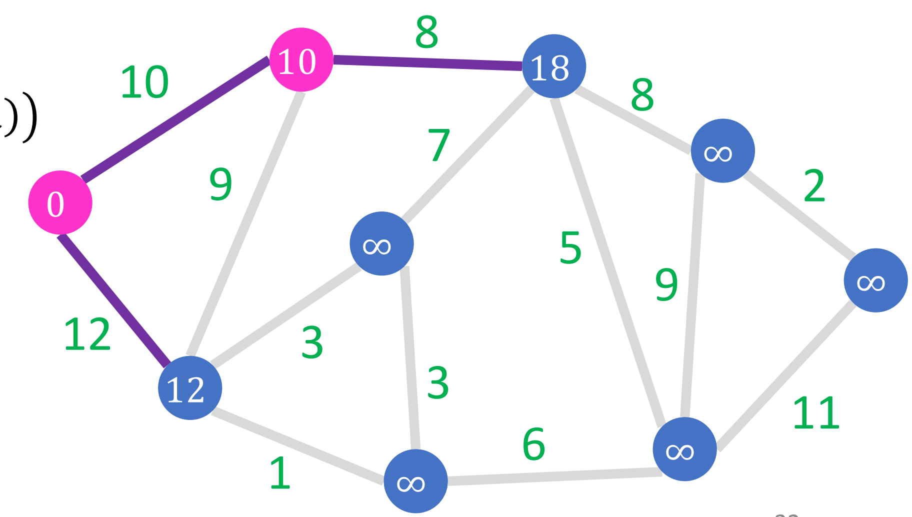
initialize $d_v = \infty$ for each node v
add all nodes $v \in V$ to the priority queue PQ, using d_v as the key
set $d_s = 0$
while PQ is not empty:
    $v = \mathrm{PQ.extractMin}()$
    for each $u \in V$ such that $(v, u) \in E$:
        if $u \in \mathrm{PQ}$ and $d_v + w(v, u) < d_u$:
            $\mathrm{PQ.decreaseKey}\big(u, d_v + w(v, u)\big)$
            $u.\mathrm{parent} = v$



Dijkstra's Algorithm Implementation

Implementation:

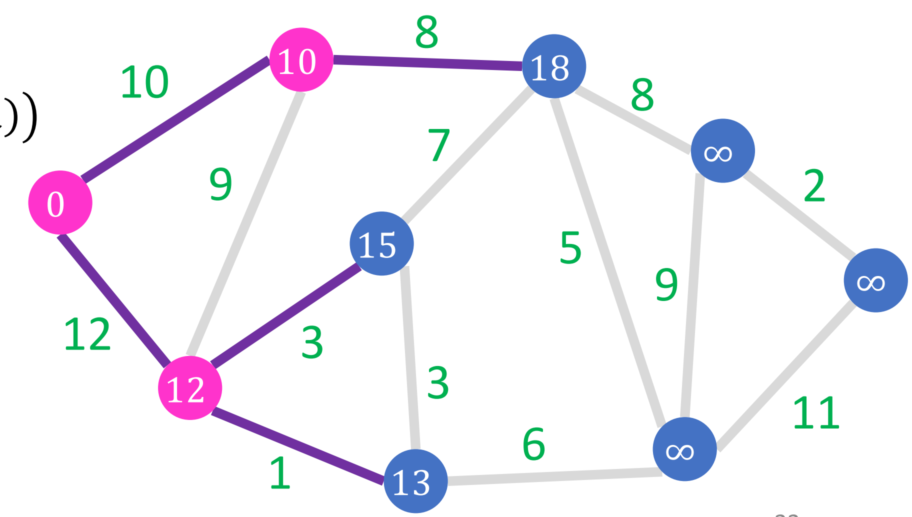
initialize $d_v = \infty$ for each node v
add all nodes $v \in V$ to the priority queue PQ, using d_v as the key
set $d_s = 0$
while PQ is not empty:
    $v = \mathrm{PQ.extractMin}()$
    for each $u \in V$ such that $(v, u) \in E$:
        if $u \in \mathrm{PQ}$ and $d_v + w(v, u) < d_u$:
            $\mathrm{PQ.decreaseKey}(u, d_v + w(v, u))$
            $u.\mathrm{parent} = v$



Dijkstra's Algorithm Implementation

Implementation:

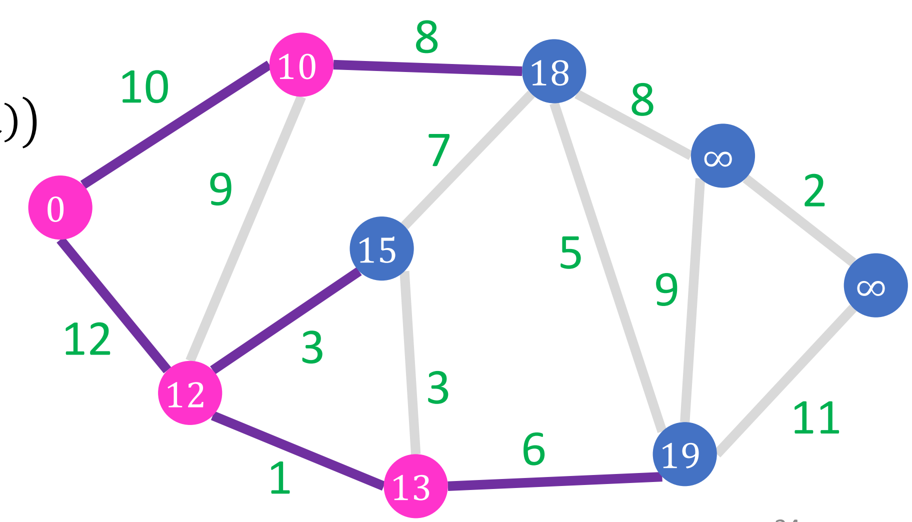
initialize $d_v = \infty$ for each node v
add all nodes $v \in V$ to the priority queue PQ, using d_v as the key
set $d_s = 0$
while PQ is not empty:
    $v = \text{PQ.extractMin}()$
    for each $u \in V$ such that $(v, u) \in E$:
        if $u \in \text{PQ}$ and $d_v + w(v, u) < d_u$:
            $\text{PQ.decreaseKey}(u, d_v + w(v, u))$
            $u.\text{parent} = v$



Dijkstra's Algorithm Implementation

Implementation:

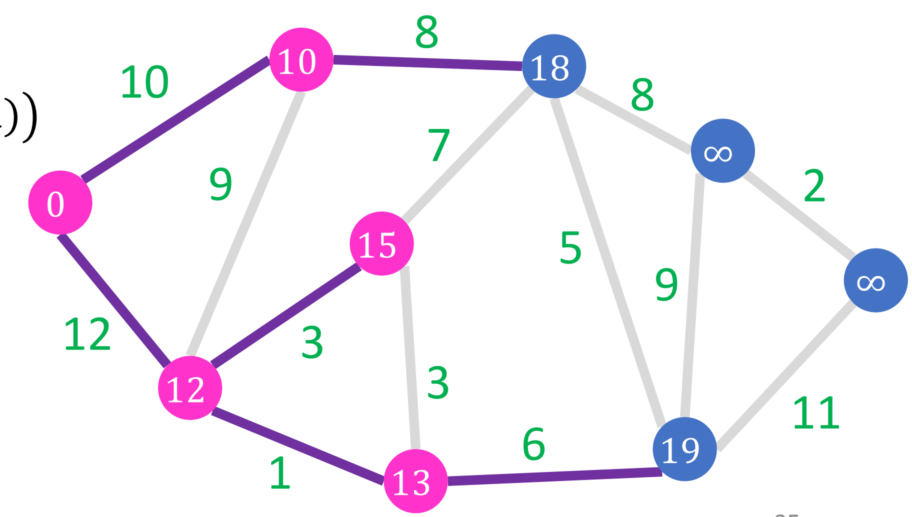
initialize $d_v = \infty$ for each node v
add all nodes $v \in V$ to the priority queue PQ, using d_v as the key
set $d_s = 0$
while PQ is not empty:
    $v = \mathrm{PQ}.\mathrm{extractMin}()$
    for each $u \in V$ such that $(v, u) \in E$:
        if $u \in \mathrm{PQ}$ and $d_v + w(v,u) < d_u$:
            $\mathrm{PQ}.\mathrm{decreaseKey}\big(u, d_v + w(v,u)\big)$
            $u.\mathrm{parent} = v$



Dijkstra's Algorithm Implementation

Implementation:

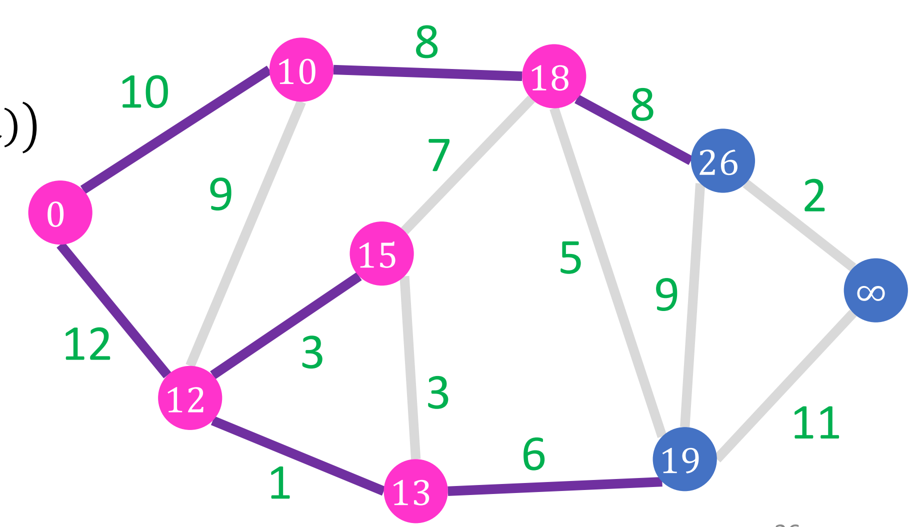
initialize $d_v = \infty$ for each node v
add all nodes $v \in V$ to the priority queue PQ, using d_v as the key
set $d_s = 0$
while PQ is not empty:
    $v = \mathrm{PQ.extractMin}()$
    for each $u \in V$ such that $(v, u) \in E$:
        if $u \in \mathrm{PQ}$ and $d_v + w(v, u) < d_u$:
            $\mathrm{PQ.decreaseKey}\big(u, d_v + w(v, u)\big)$
            $u.\mathrm{parent} = v$



Dijkstra's Algorithm Implementation

Implementation:

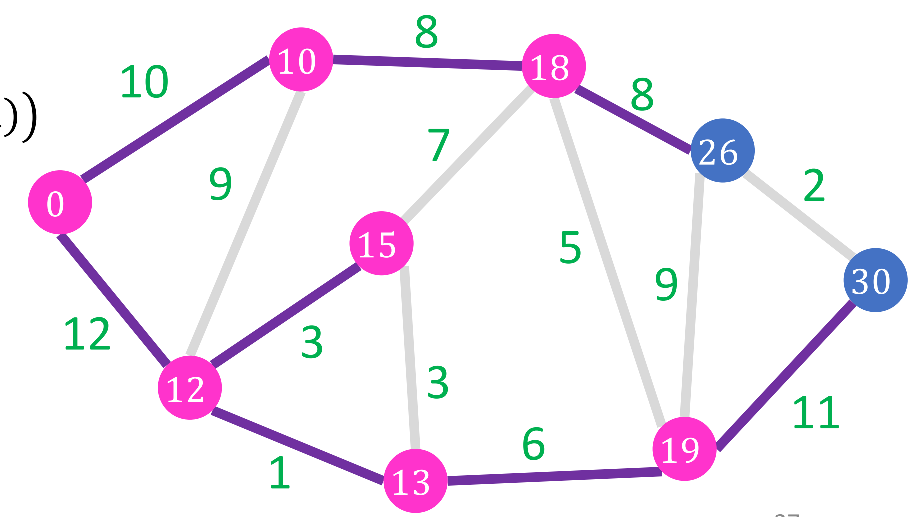
initialize $d_v = \infty$ for each node v
add all nodes $v \in V$ to the priority queue PQ, using d_v as the key
set $d_s = 0$
while PQ is not empty:
  $v = \mathrm{PQ.extractMin()}$
  for each $u \in V$ such that $(v, u) \in E$:
    if $u \in \mathrm{PQ}$ and $d_v + w(v, u) < d_u$:
      $\mathrm{PQ.decreaseKey}\big(u, d_v + w(v, u)\big)$
      $u.\mathrm{parent} = v$



Dijkstra's Algorithm Implementation

Implementation:

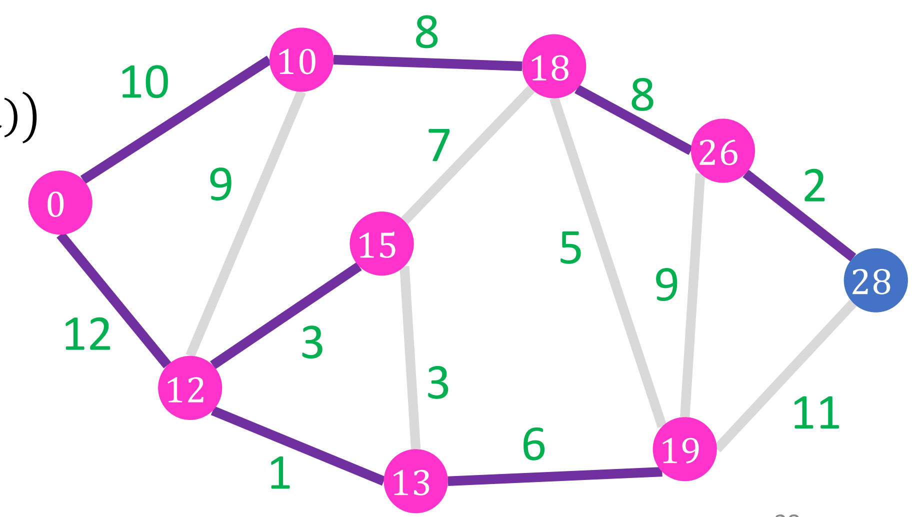
initialize $d_v = \infty$ for each node v
add all nodes $v \in V$ to the priority queue PQ, using d_v as the key
set $d_s = 0$
while PQ is not empty:
  $v = \mathrm{PQ.extractMin}()$
  for each $u \in V$ such that $(v, u) \in E$:
    if $u \in \mathrm{PQ}$ and $d_v + w(v, u) < d_u$:
      $\mathrm{PQ.decreaseKey}\big(u, d_v + w(v, u)\big)$
      $u.\mathrm{parent} = v$



Dijkstra's Algorithm Implementation

Implementation:

initialize $d_v = \infty$ for each node v
add all nodes $v \in V$ to the priority queue PQ, using d_v as the key
set $d_s = 0$
while PQ is not empty:
  $v = \mathrm{PQ}.\mathrm{extractMin}()$
  for each $u \in V$ such that $(v, u) \in E$:
    if $u \in \mathrm{PQ}$ and $d_v + w(v, u) < d_u$:
      $\mathrm{PQ}.\mathrm{decreaseKey}\big(u, d_v + w(v, u)\big)$
      $u.\mathrm{parent} = v$



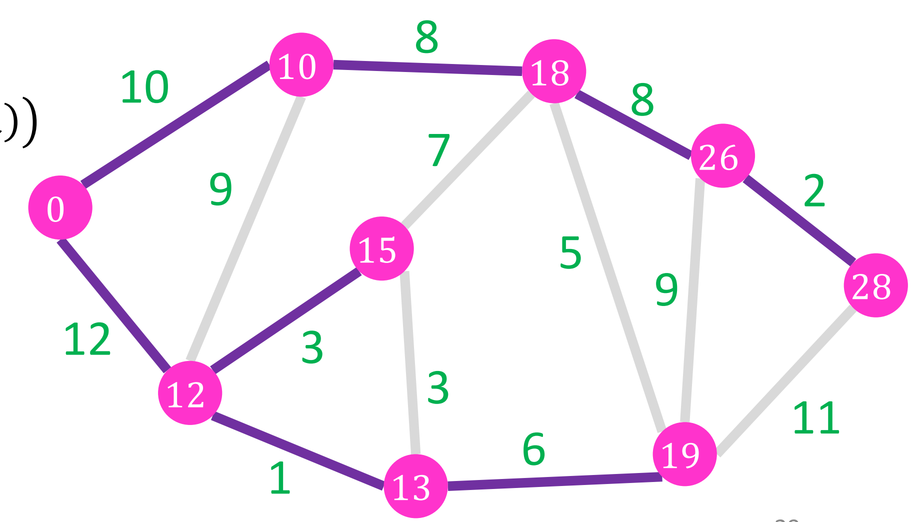
Dijkstra's Algorithm Implementation

Implementation:

initialize $d_v = \infty$ for each node v
add all nodes $v \in V$ to the priority queue PQ, using d_v as the key
set $d_s = 0$
while PQ is not empty:
    $v = \mathrm{PQ.extractMin}()$
    for each $u \in V$ such that $(v, u) \in E$:
        if $u \in \mathrm{PQ}$ and $d_v + w(v, u) < d_u$:
            $\mathrm{PQ.decreaseKey}\big(u, d_v + w(v, u)\big)$
            $u.\mathrm{parent} = v$

Every subpath of a shortest path is itself a shortest path (optimal substructure)

Observe: shortest paths from a source forms a <u>tree</u>, but **not** a minimum spanning tree



Dijkstra's Algorithm Running Time

Implementation:

initialize $d_v = \infty$ for each node v add all nodes $v \in V$ to the priority queue PQ, using d_v as the key set $d_s = 0$	Initialization: $O(\|V\|)$
while PQ is not empty:	$\|V\|$ iterations
$v = \mathrm{PQ}.\mathrm{extractMin}()$	$O(\log\|V\|)$
for each $u \in V$ such that $(v, u) \in E$:	$2\|E\|$ iterations <u>total</u>
if $u \in \mathrm{PQ}$ and $d_v + w(v, u) < d_u$:	
$\mathrm{PQ}.\mathrm{decreaseKey}\big(u, d_v + w(v, u)\big)$	$O(\log\|V\|)$
$u.\mathrm{parent} = v$	

Overall running time: $O(|V|\log|V| + |E|\log|V|) = O(|E|\log|V|)$

Dijkstra's Algorithm Proof Strategy

Proof by induction

Proof Idea: we will show that when node u is removed from the priority queue, $d_u = \delta(s, u)$

- **Claim 1:** There is a path of length d_u (as long as $d_u < \infty$) from s to u in G
- **Claim 2:** For every path $(s, \dots, u)$, $w(s, \dots, u) \geq d_u$

Correctness of Dijkstra's Algorithm

Inductive hypothesis: Suppose that nodes $v_1 = s, \dots, v_i$ have been removed from PQ, and for each of them $d_{v_i} = \delta(s, v_i)$, and there is a path from s to v_i with distance d_{v_i} (whenever $d_{v_i} < \infty$)

Base case:

- $i = 0$: $v_1 = s$
- Claim holds trivially

Correctness of Dijkstra's Algorithm: Claim 1

Let u be the $(i+1)^{\text{st}}$ node extracted

Claim 1: There is a path of length d_u (as long as $d_u < \infty$) from s to u in G

Proof:

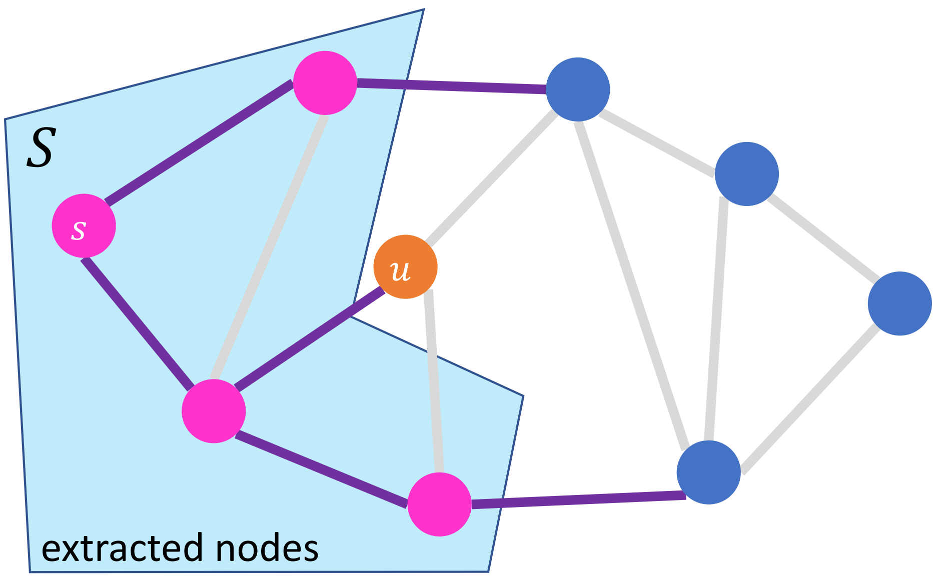
- Suppose $d_u < \infty$
- This means that PQ.decreaseKey was invoked on node u on an earlier iteration
- Consider the last time PQ.decreaseKey is invoked on node u
- PQ.decreaseKey is only invoked when there exists an edge $(v, u) \in E$ and node v was extracted from PQ in a previous iteration
- In this case, $d_u = d_v + w(v, u)$
- By the inductive hypothesis, there is a path $s \to v$ of length d_v in G and since there is an edge $(v, u) \in E$, there is a path $s \to u$ of length d_u in G

Correctness of Dijkstra's Algorithm: Claim 2

Let u be the $(i+1)^{\text{st}}$ node extracted

Claim 2: For every path $(s, \dots, u)$, $w(s, \dots, u) \geq d_u$

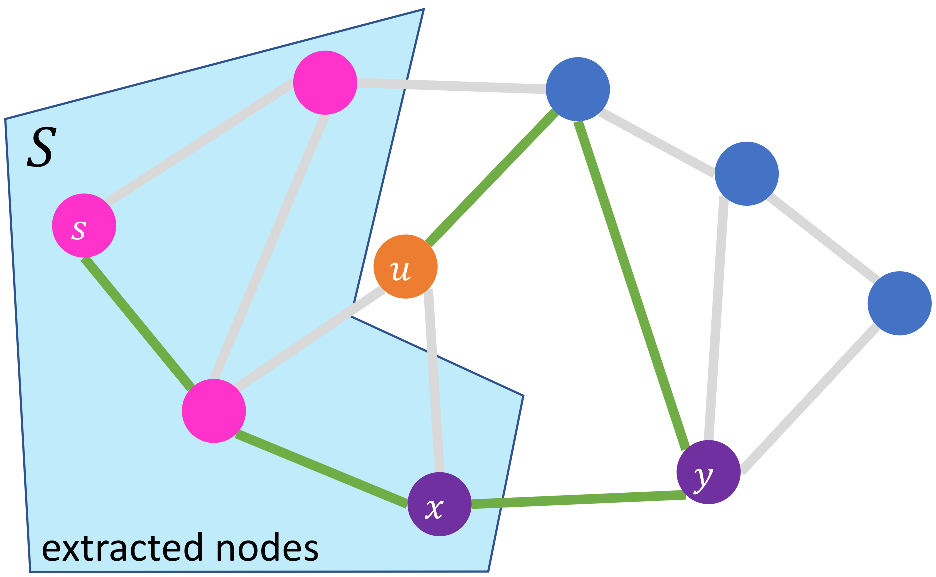
Extracted nodes define a cut $(S, V - S)$ of G



Correctness of Dijkstra's Algorithm: Claim 2

Let u be the $(i+1)^{\text{st}}$ node extracted

Claim 2: For every path $(s, \ldots, u)$, $w(s, \ldots, u) \geq d_u$



Extracted nodes define a cut $(S, V - S)$ of G

Take any path $(s, \ldots, u)$

Since $u \notin S$, $(s, \ldots, u)$ crosses the cut somewhere

- Let (x, y) be last edge in the path that crosses the cut

$$w(s, \ldots, u) \geq \delta(s, x) + w(x, y) + w(y, \ldots, u)$$

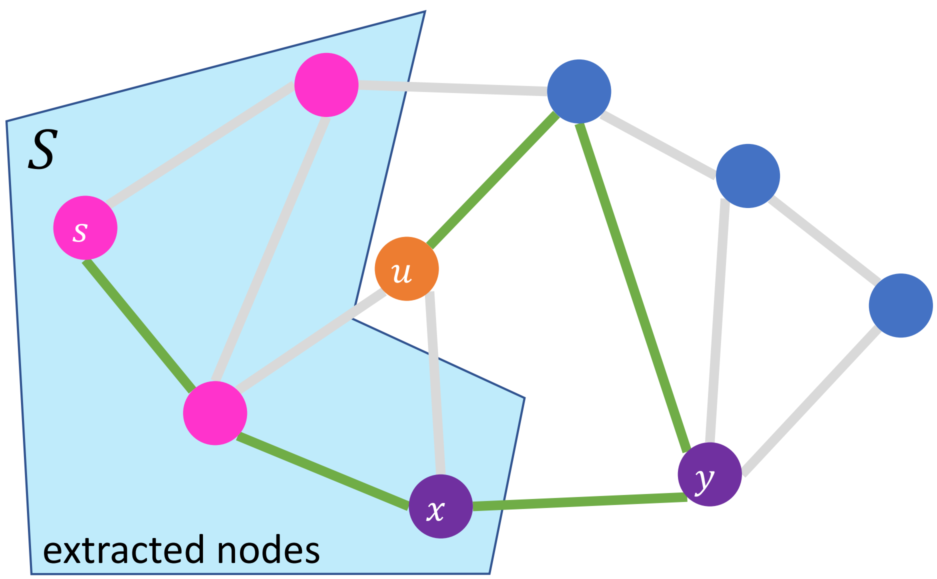
$w(s, \ldots, u) = w(s, \ldots, x) + w(x, y) + w(y, \ldots, u)$

$w(s, \ldots, x) \geq \delta(s, x)$ since $\delta(s, x)$ is weight of shortest path from s to x

Correctness of Dijkstra's Algorithm: Claim 2

Let u be the $(i+1)^{\text{st}}$ node extracted

Claim 2: For every path $(s, \dots, u)$, $w(s, \dots, u) \geq d_u$



Extracted nodes define a cut $(S, V - S)$ of G

Take any path $(s, \dots, u)$

Since $u \notin S$, $(s, \dots, u)$ crosses the cut somewhere

- Let (x, y) be last edge in the path that crosses the cut

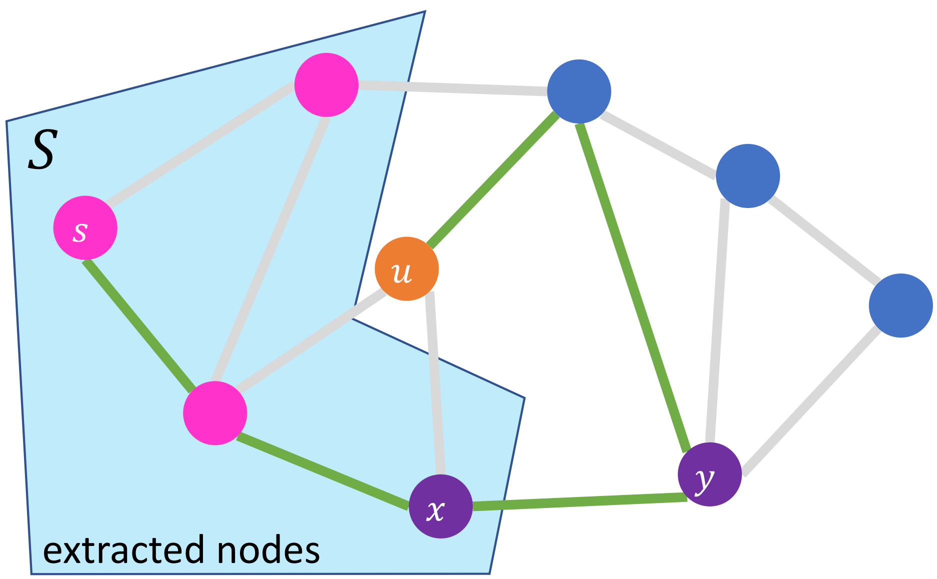
$$\begin{aligned} w(s, \dots, u) &\geq \delta(s, x) + w(x, y) + w(y, \dots, u) \\ &= d_x + w(x, y) + w(y, \dots, u) \end{aligned}$$

Inductive hypothesis: since x was extracted before, $d_x = \delta(s, x)$

Correctness of Dijkstra's Algorithm: Claim 2

Let u be the $(i+1)^{\text{st}}$ node extracted

Claim 2: For every path $(s, \dots, u)$, $w(s, \dots, u) \geq d_u$



Extracted nodes define a cut $(S, V - S)$ of G

Take any path $(s, \dots, u)$

Since $u \notin S$, $(s, \dots, u)$ crosses the cut somewhere

- Let (x, y) be last edge in the path that crosses the cut

$$
\begin{aligned}
w(s, \dots, u) &\geq \delta(s, x) + w(x, y) + w(y, \dots, u) \\
&= d_x + w(x, y) + w(y, \dots, u) \\
&\geq d_y + w(y, \dots, u)
\end{aligned}
$$

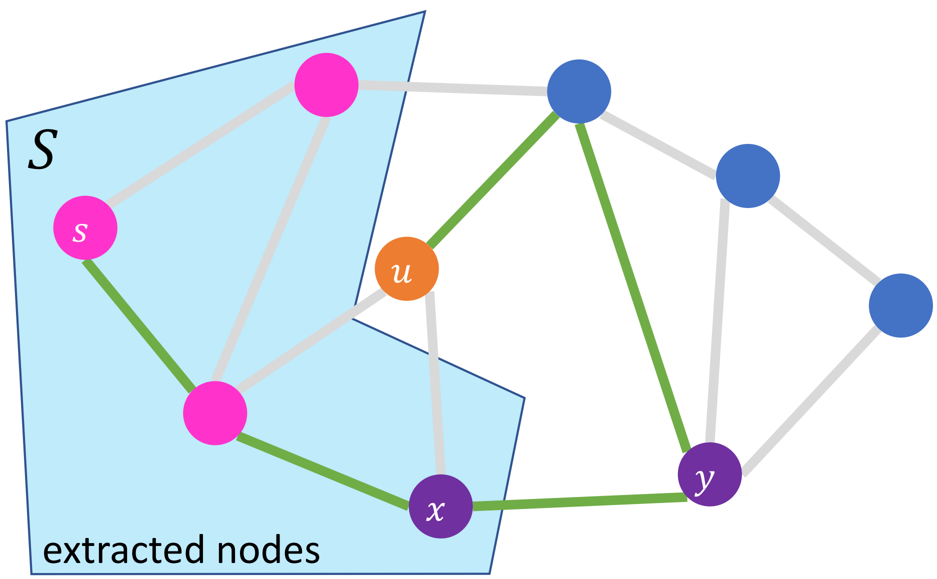
By construction of Dijkstra's algorithm, when x is extracted, d_y is updated to satisfy

$$d_y \leq d_x + w(x, y)$$

Correctness of Dijkstra's Algorithm: Claim 2

Let u be the $(i+1)^{\text{st}}$ node extracted

Claim 2: For every path $(s, \dots, u)$, $w(s, \dots, u) \geq d_u$



Extracted nodes define a cut $(S, V - S)$ of G

Take any path $(s, \dots, u)$

Since $u \notin S$, $(s, \dots, u)$ crosses the cut somewhere

- Let (x, y) be last edge in the path that crosses the cut

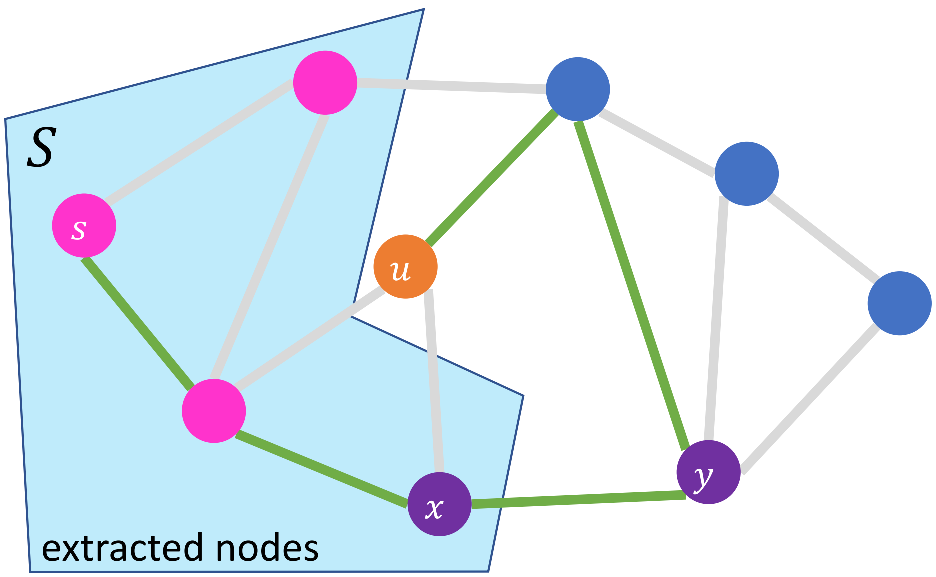
$$\begin{aligned} w(s, \dots, u) &\geq \delta(s, x) + w(x, y) + w(y, \dots, u) \\ &= d_x + w(x, y) + w(y, \dots, u) \\ &\geq d_y + w(y, \dots, u) \\ &\geq d_u + w(y, \dots, u) \end{aligned}$$

Greedy choice property: we always extract the node of minimal distance so $d_u \leq d_y$

Correctness of Dijkstra's Algorithm: Claim 2

Let u be the $(i+1)^{\text{st}}$ node extracted

Claim 2: For every path $(s, \dots, u)$, $w(s, \dots, u) \geq d_u$



Extracted nodes define a cut $(S, V - S)$ of G

Take any path $(s, \dots, u)$

Since $u \notin S$, $(s, \dots, u)$ crosses the cut somewhere

- Let (x, y) be last edge in the path that crosses the cut

$$
\begin{aligned}
w(s, \dots, u) &\geq \delta(s, x) + w(x, y) + w(y, \dots, u) \\
&= d_x + w(x, y) + w(y, \dots, u) \\
&\geq d_y + w(y, \dots, u) \\
&\geq d_u + w(y, \dots, u) \\
&\geq d_u
\end{aligned}
$$

All edge weights assumed to be positive

Correctness of Dijkstra's Algorithm

Proof by induction

Proof Idea: we will show that when node u is removed from the priority queue, $d_u = \delta(s, u)$

- **Claim 1:** There is a path of length d_u (as long as $d_u < \infty$) from s to u in G
- **Claim 2:** For every path $(s, \dots, u)$, $w(s, \dots, u) \geq d_u$

Breadth-First Search

Input: a graph G (weighted or unweighted) and a node s

Behavior: Start with node s, visit all neighbors of s, then all neighbors of neighbors of s, until all nodes have been visited

Output: BFS can be used to do many useful things, so lots of choices!

- Is the graph connected?
- Is there a path from s to u?
- Smallest number of "hops" from s to u

Sounds like a "shortest path" property!

Notes: BFS doesn't use edge weights at all!

Also, depth-first search (DFS) also similarly useful

Dijkstra's SP Algorithm

initialize $d_v = \infty$ for each node v
add all nodes $v \in V$ to the priority queue PQ, using d_v as the key
set $d_s = 0$
while PQ is not empty:
    $v = \mathrm{PQ}.\mathrm{extractMin}()$
    for each $u \in V$ such that $(v, u) \in E$:
        if $u \in \mathrm{PQ}$ and $d_v + w(v, u) < d_u$:
            $\mathrm{PQ}.\mathrm{decreaseKey}\left(u, d_v + w(v, u)\right)$
            $u.\mathrm{parent} = v$

Breadth-First Search

initialize a flag $d_v = 0$ for each node v
pick a start node s
$\mathrm{Q.push}(s)$
while Q is not empty:
    $v = \mathrm{Q.pop}()$ and set $d_v = 1$
    for each $u \in V$ such that $(v, u) \in E$:
        if $d_u = 0$:
            $\mathrm{Q.push}(u)$

flag to denote whether a node has been visited or not

Key observation: replace the priority queue with a queue

Breadth-First Search: Time Complexity

initialize a flag $d_v = 0$ for each node v — Initialization: $O(|V|)$
pick a start node s
$\mathrm{Q.push}(s)$
while Q is not empty: — $|V|$ iterations
    $v = \mathrm{Q.pop}()$ and set $d_v = 1$
    for each $u \in V$ such that $(v, u) \in E$: — $2|E|$ iterations total
        if $d_u = 0$:
            $\mathrm{Q.push}(u)$

Overall running time: $O(|E| + |V|)$
The larger of $|E|$ and $|V|$. (For graphs, we call this "linear".)

BFS to Count Number of Hops

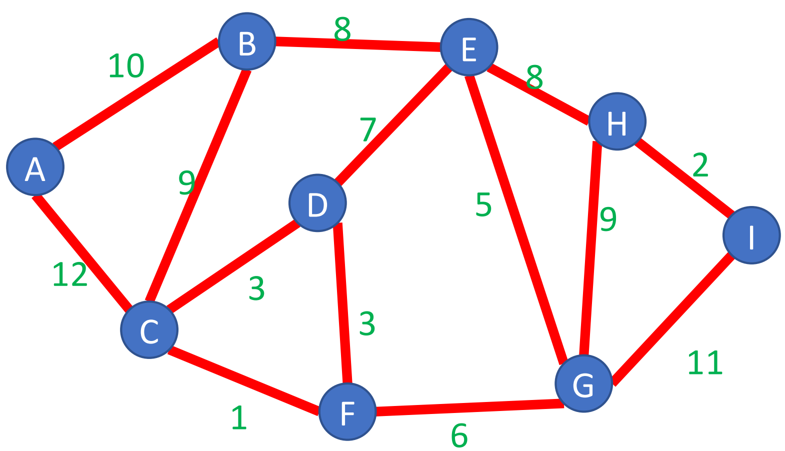
initialize a counter $d_v = \infty$ for each node v
pick a start node s and set $d_s = 0$
$\text{Q.push}(s)$
while Q is not empty:
    $v = \text{Q.pop}()$
    for each $u \in V$ such that $(v, u) \in E$:
        if $d_u = \infty$:
            $\text{Q.push}(u)$
            $d_u = d_v + 1$

counter to denote number of hops from the source

BFS Trees

Let's draw a *BFS tree*, a trace of its execution

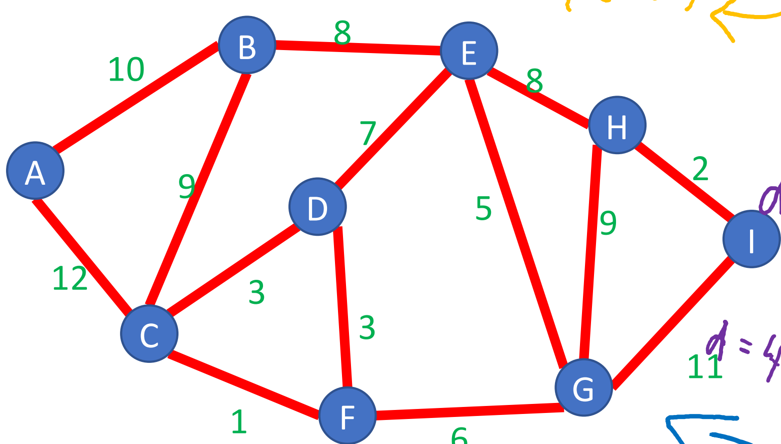
- Number each node as visited
- Distance from start
- Tree edges vs non-tree edges

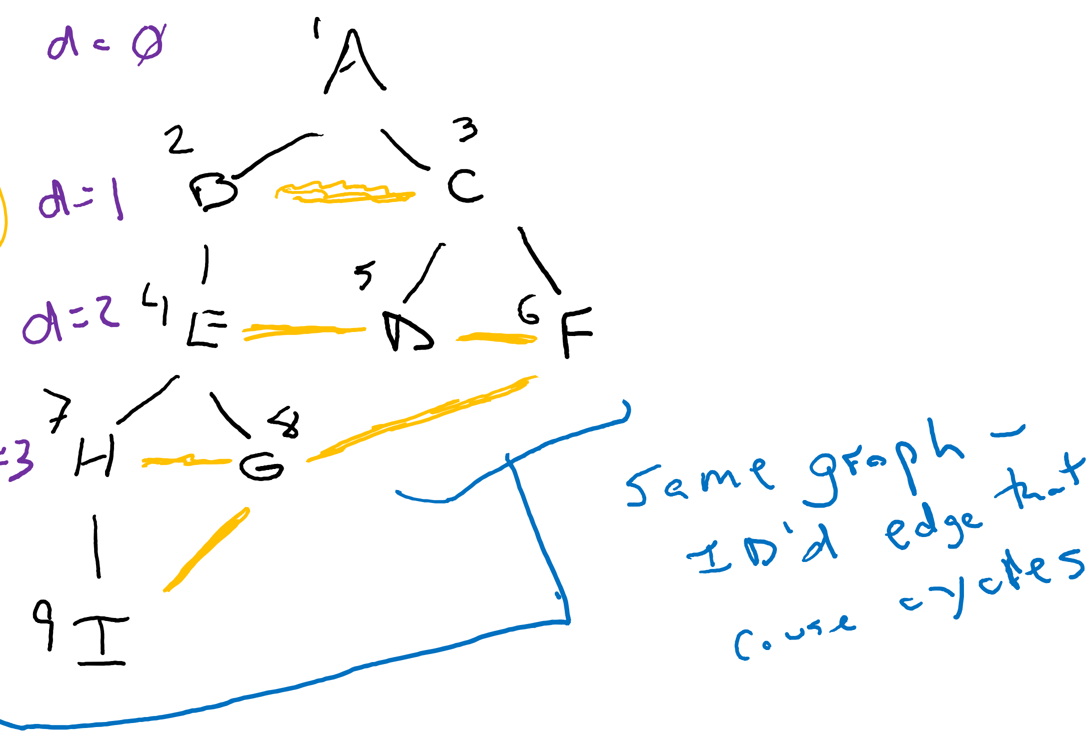


(Duplicated slide) BFS Trees

Let's draw a *BFS tree*, a trace of its execution

- Number each node as visited
- Distance from start
- Tree edges vs non-tree edges





Summary

Shortest path in weighted-graphs (single-source)

- Dijkstra’s SP Algorithm
  - Greedy algorithm
  - Similar in structure to Prim’s MST algorithm
  - Priority queue ordered by distance from start (not connecting edge weight)

Unweighted graphs, number of “hops”

- Distance is number of edges (not sum of edge weights)
- Breadth-first Search (BFS)
  - Not greedy. Doesn’t used edge weights

BFS (and DFS) useful to solve many other graph problems

- Connectivity, find cycles, ….