Design and Analysis of Algorithms
Greedy Algorithms

Si Wu

School of CSE, SCUT

cswusi@scut.edu.cn

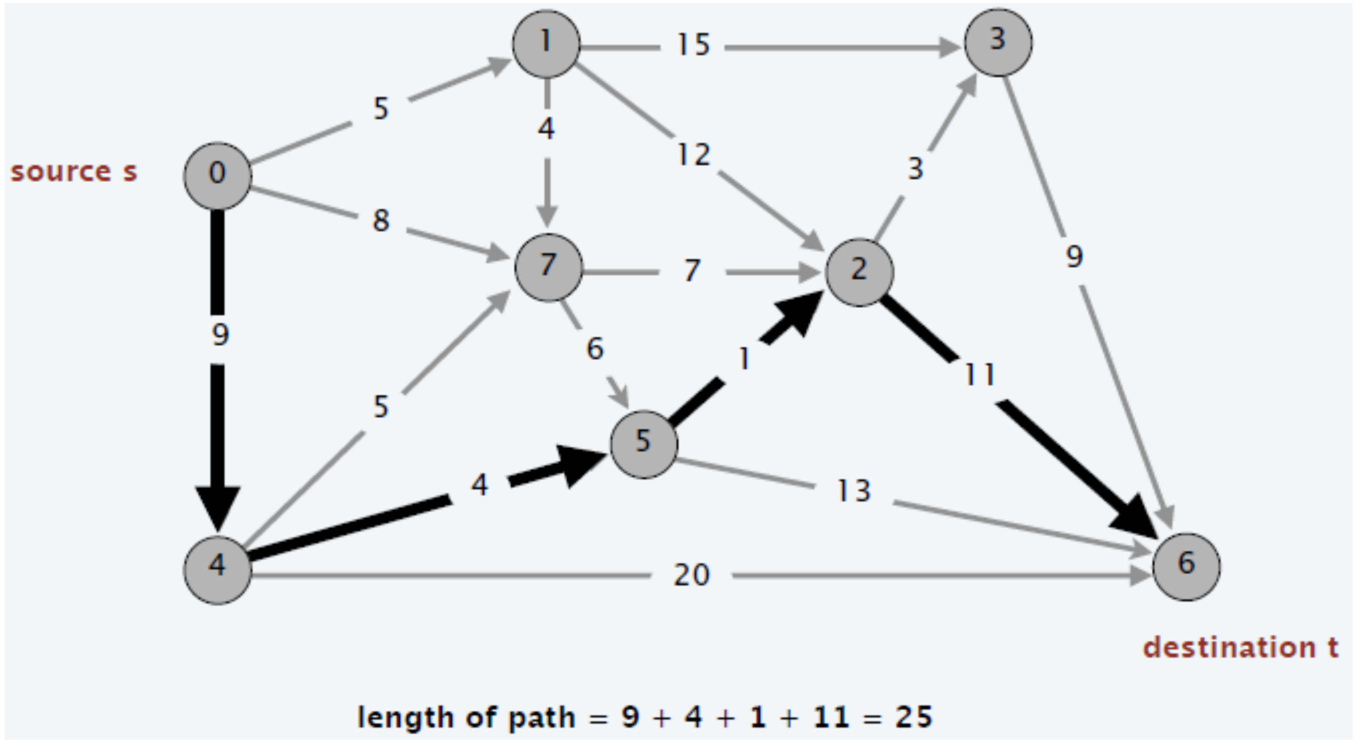
TA: 1684350406@qq.com

Topics

- Dijkstra’s Algorithm
- Minimum Spanning Trees
- Prim’s Algorithm
- Kruskal’s Algorithms

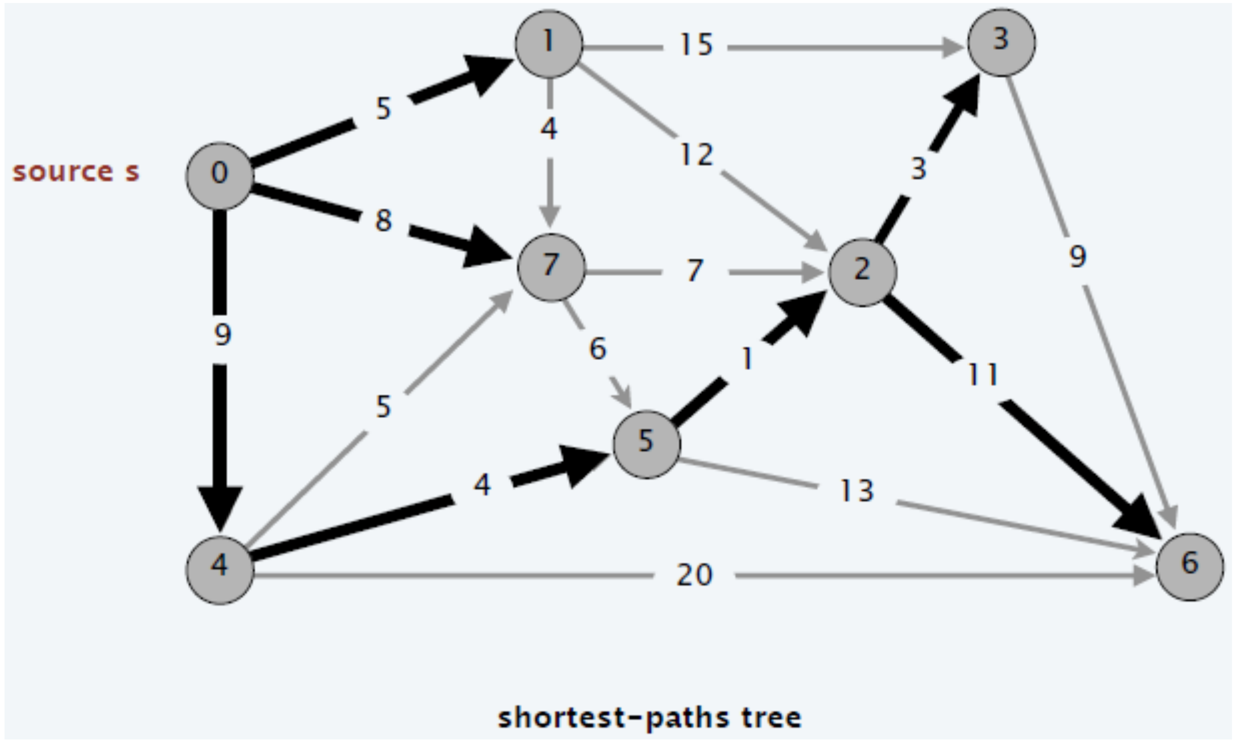
Single-Pair Shortest Path Problem

Problem. Given a digraph $G = (V, E)$, edge lengths $l_e \geq 0$, source $s \in V$, and destination $t \in V$, find a shortest directed path from s to t.



Single-Source Shortest Path Problem

Problem. Given a digraph $G = (V, E)$, edge lengths $l_e \geq 0$, source $s \in V$, find a shortest directed path from s to every node.



Car Navigation

Single-destination shortest paths problem.

Dijkstra's Algorithm for Single-Source Shortest Path Problem

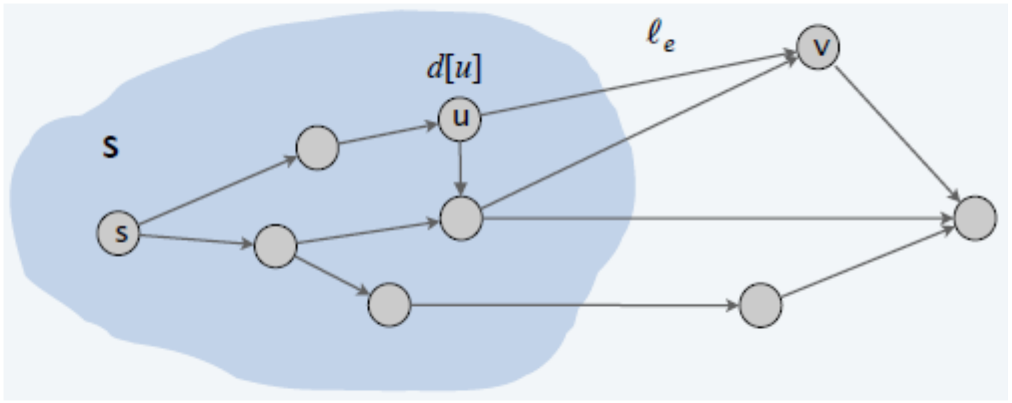
Greedy approach. Maintain a set of explored nodes S for which algorithm has determined $d[u]$ = length of a shortest $s \to u$ path.

- Initialize $S \leftarrow \{s\}, d[s] = 0$.
- Repeatedly choose unexplored node $v \notin S$ which minimizes

$$\pi(v) = \min_{e=(u,v):u\in S} d[u] + l_e$$

add v to S, set $d[v] = \pi(v)$.

The length of a shortest path from s to some node u in explored part S, followed by a single edge $e = (u, v)$.



Dijkstra's Algorithm for Single-Source Shortest Path Problem

Greedy approach. Maintain a set of explored nodes S for which algorithm has determined $d[u]$ = length of a shortest $s \to u$ path.

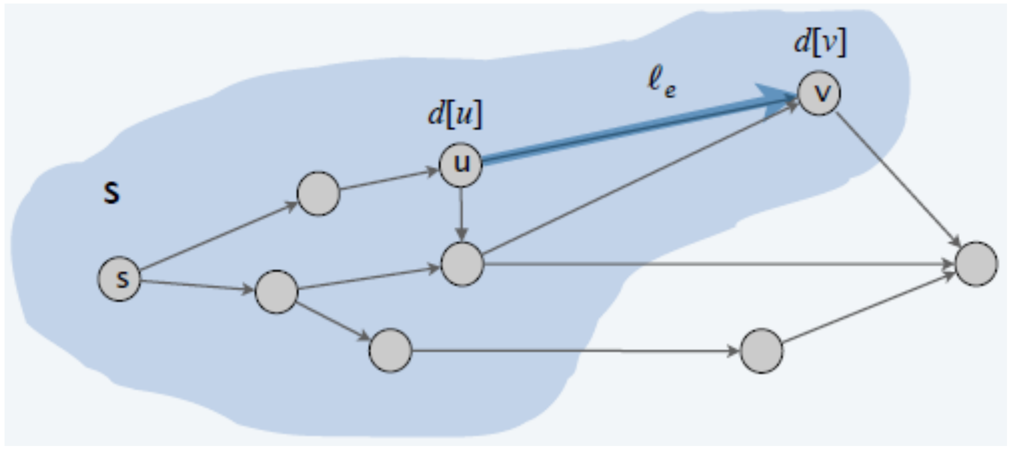
- Initialize $S \leftarrow \{s\}, d[s] = 0$.
- Repeatedly choose unexplored node $v \notin S$ which minimizes

$$\pi(v) = \min_{e=(u,v):u\in S} d[u] + l_e$$

add v to S, set $d[v] = \pi(v)$.

The length of a shortest path from s to some node u in explored part S, followed by a single edge $e = (u, v)$.

- To recover path, set $pred[v] \leftarrow e$ that achieves min.



Dijkstra's Algorithm: Proof of Correctness

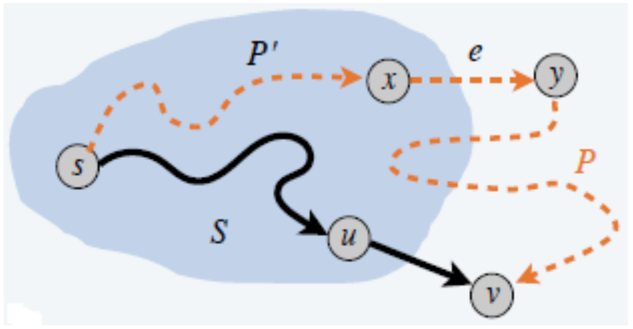
For each node $u \in S$: $d[u]$ = length of a shortest $s \rightarrow u$ path.

Pf. By induction on $|S|$

Base case: $|S| = 1$ is easy since $S = \{s\}$ and $d[s] = 0$.

Inductive hypothesis: Assume true for $|S| \geq 1$.

- Let v be next node added to S, and let (u, v) be the final edge.
- A shortest $s \rightarrow u$ path plus (u, v) is an $s \rightarrow v$ path of length $\pi(v)$.
- Consider any other $s \rightarrow v$ path P. We show that it is no shorter than $\pi(v)$.
- Let $e = (x, y)$ be the first edge in P that leaves S, and let P' be the sub-path to x.
- The length of P is already $\geq \pi(v)$ as soon as it reaches y:



$$l(P) \geq l(P') + l_e \geq d[x] + l_e \geq \pi(y) \geq \pi(v)$$

Non-negative lengths

Inductive hypothesis

Definition of $\pi(y)$

Dijkstra chose v instead of y

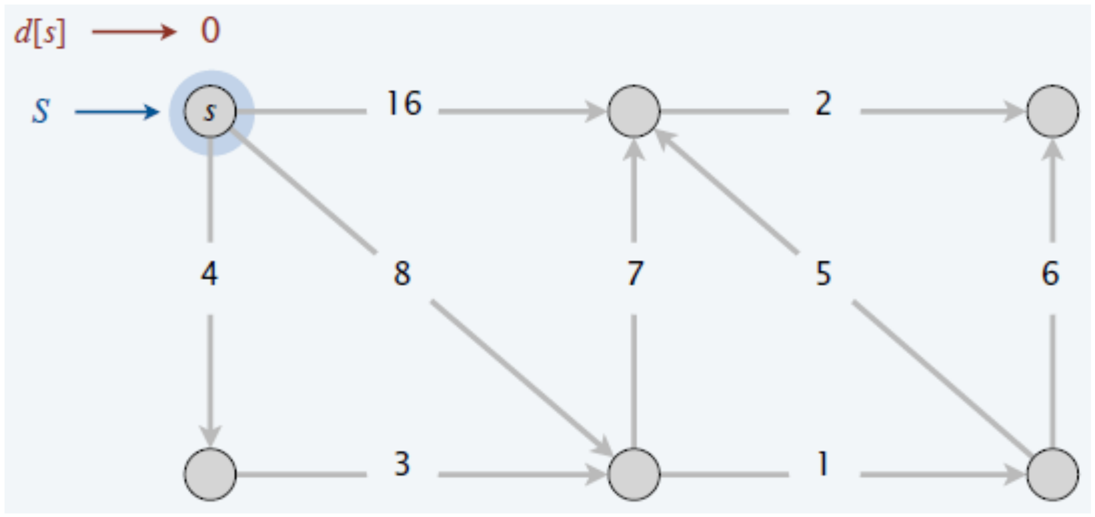
Dijkstra's Algorithm Demo

- Initialize $S \leftarrow \{s\}$ and $d[s] \leftarrow 0$.
- Repeatedly choose unexplored node $v \notin S$ which minimizes

$$\pi(v) = \min_{e=(u,v):u\in S} d[u] + l_e$$

Add v to S; set $d[v] \leftarrow \pi(v)$ and $pred(v) \leftarrow$ *argmin.*

The length of a shortest path from s to some node u in explored part S, followed by a single edge $e = (u, v)$.



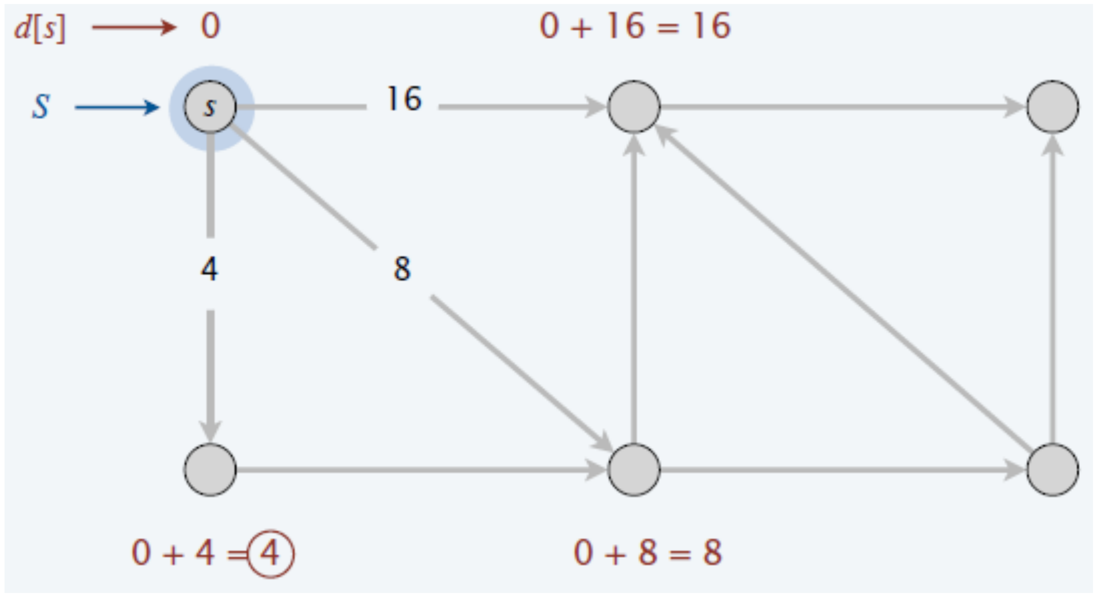
Dijkstra's Algorithm Demo

- Initialize $S \leftarrow \{s\}$ and $d[s] \leftarrow 0$.
- Repeatedly choose unexplored node $v \notin S$ which minimizes

$$\pi(v) = \min_{e=(u,v):u\in S} d[u] + l_e$$

Add v to S; set $d[v] \leftarrow \pi(v)$ and $pred(v) \leftarrow argmin.$

The length of a shortest path from s to some node u in explored part S, followed by a single edge $e = (u, v)$.

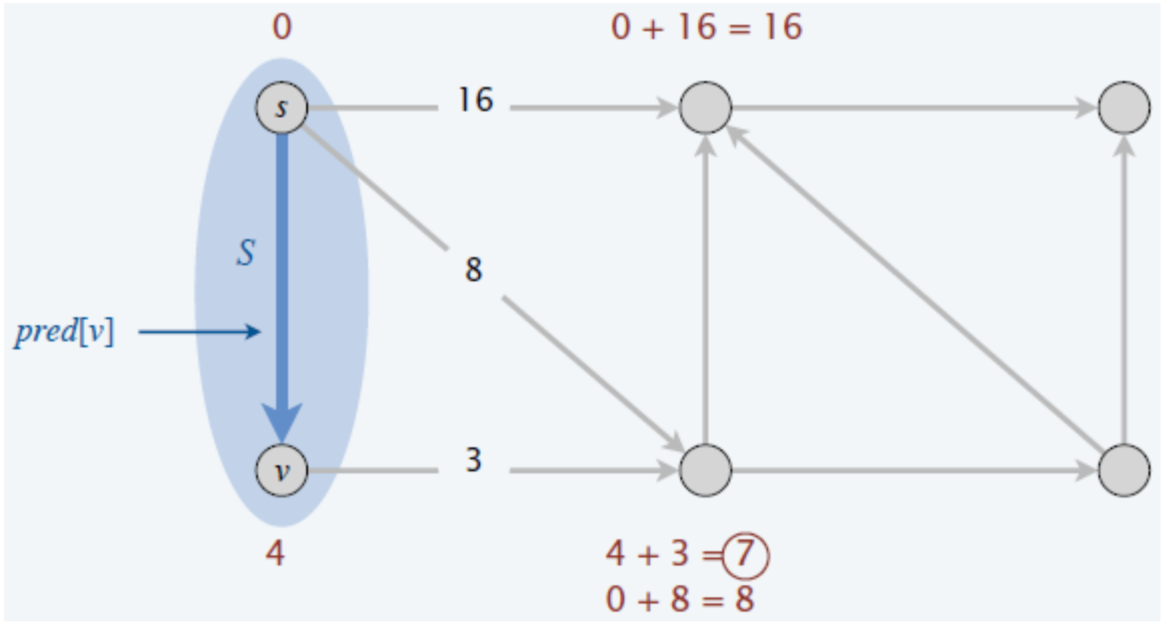


Dijkstra's Algorithm Demo

- Initialize $S \leftarrow \{s\}$ and $d[s] \leftarrow 0$.
- Repeatedly choose unexplored node $v \notin S$ which minimizes

$$\pi(v) = \min_{e=(u,v):u\in S} d[u] + l_e$$

Add v to S; set $d[v] \leftarrow \pi(v)$ and $pred(v) \leftarrow$ *argmin.*



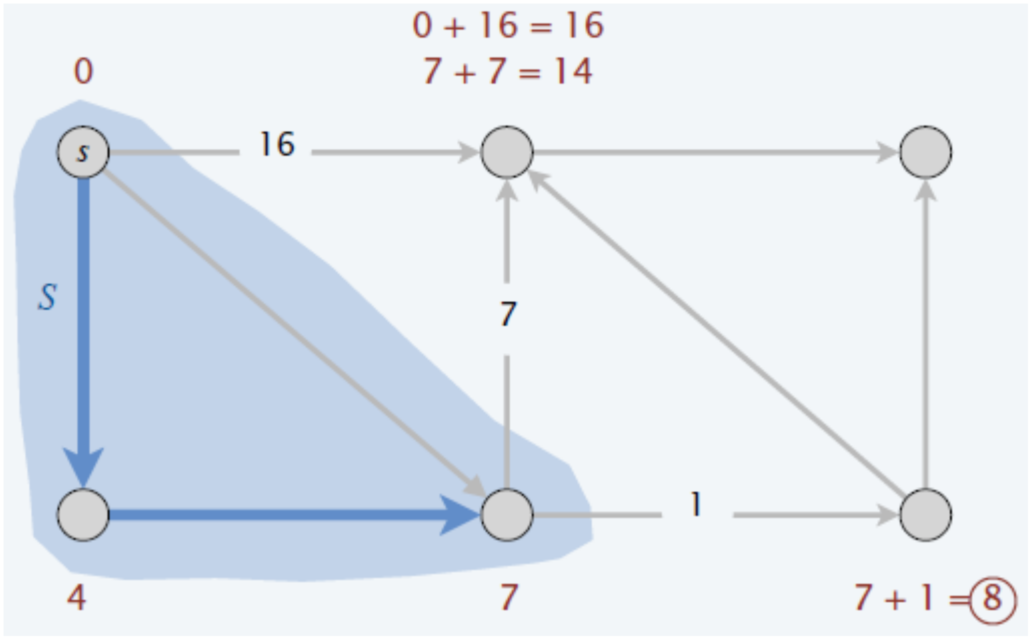
The length of a shortest path from s to some node u in explored part S, followed by a single edge $e = (u, v)$.

Dijkstra’s Algorithm Demo

- Initialize $S \leftarrow \{s\}$ and $d[s] \leftarrow 0$.
- Repeatedly choose unexplored node $v \notin S$ which minimizes

$$\pi(v) = \min_{e=(u,v):u\in S} d[u] + l_e$$

Add v to S; set $d[v] \leftarrow \pi(v)$ and $pred(v) \leftarrow argmin.$



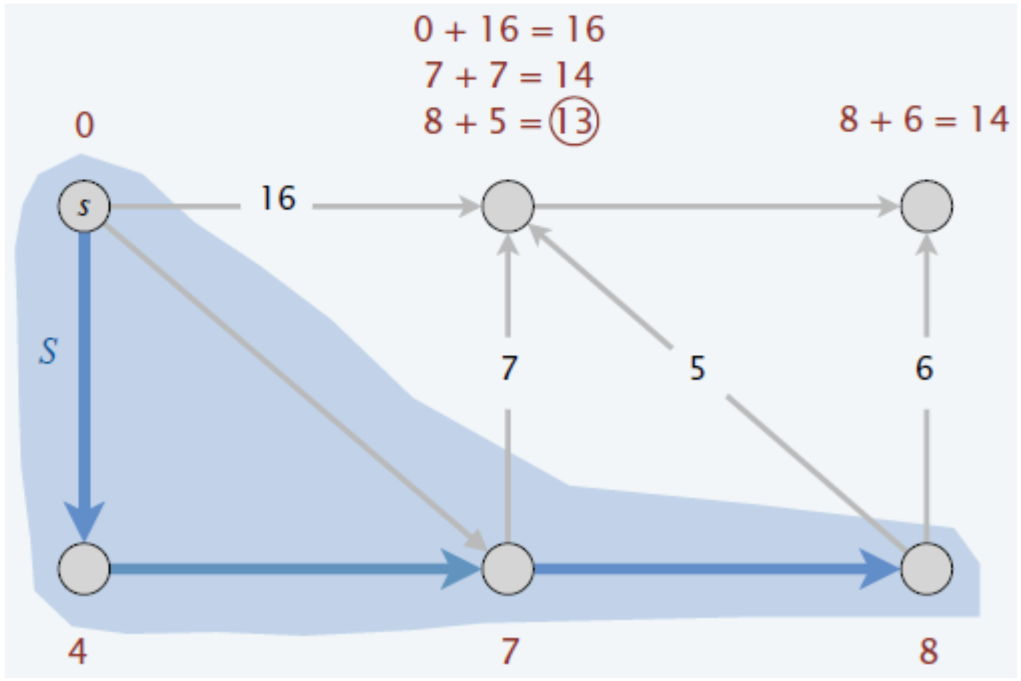
The length of a shortest path from s to some node u in explored part S, followed by a single edge $e = (u, v)$.

Dijkstra's Algorithm Demo

- Initialize $S \leftarrow \{s\}$ and $d[s] \leftarrow 0$.
- Repeatedly choose unexplored node $v \notin S$ which minimizes

$$\pi(v) = \min_{e=(u,v):u\in S} d[u] + l_e$$

Add v to S; set $d[v] \leftarrow \pi(v)$ and $pred(v) \leftarrow$ *argmin.*



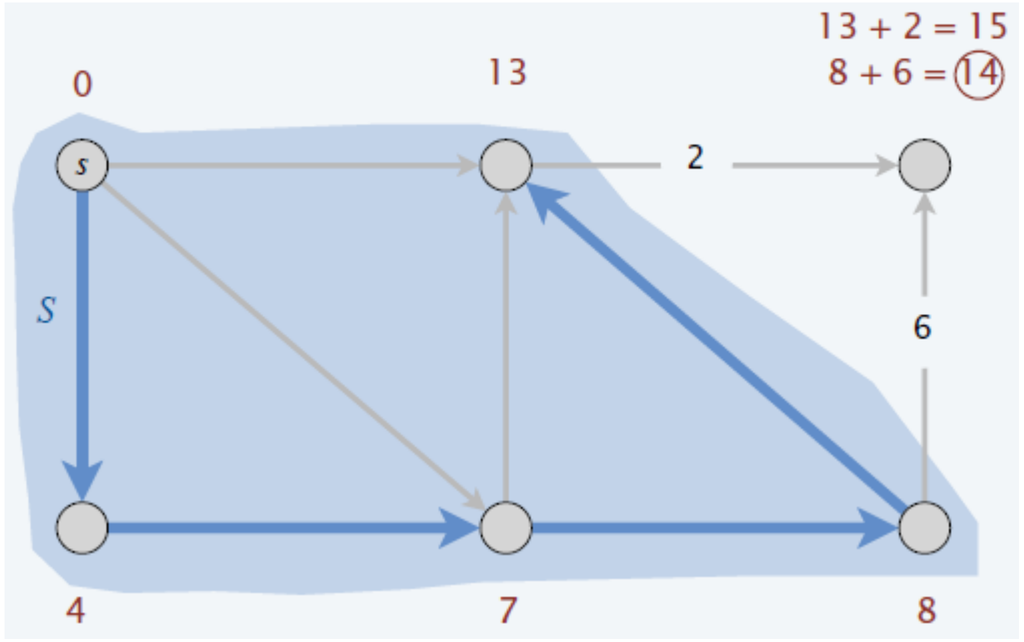
The length of a shortest path from s to some node u in explored part S, followed by a single edge $e = (u, v)$.

Dijkstra's Algorithm Demo

- Initialize $S \leftarrow \{s\}$ and $d[s] \leftarrow 0$.
- Repeatedly choose unexplored node $v \notin S$ which minimizes

$$\pi(v) = \min_{e=(u,v):u\in S} d[u] + l_e$$

Add v to S; set $d[v] \leftarrow \pi(v)$ and $pred(v) \leftarrow argmin$.



The length of a shortest path from s to some node u in explored part S, followed by a single edge $e = (u, v)$.

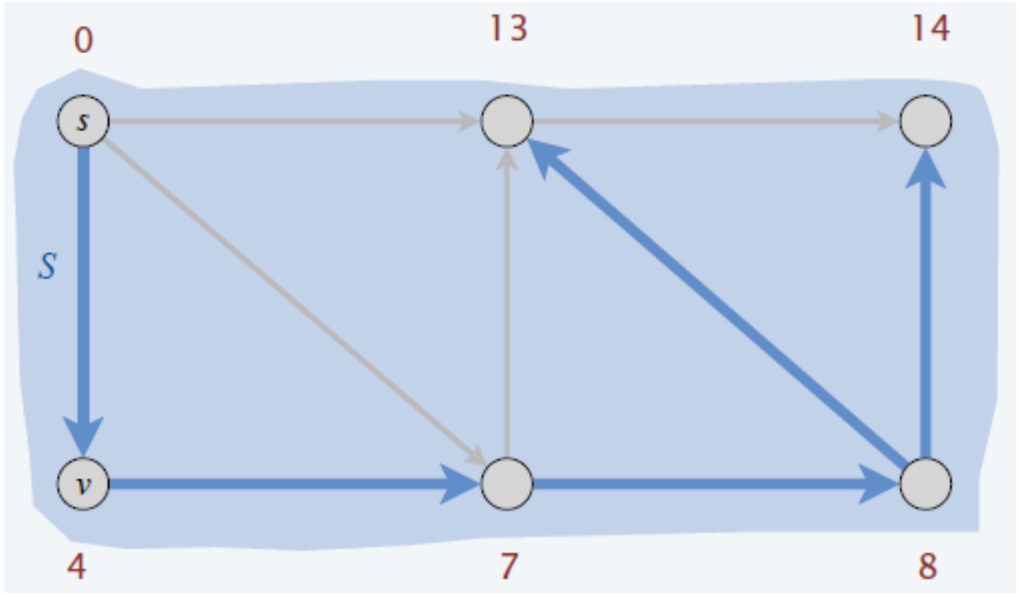
Dijkstra's Algorithm Demo

- Initialize $S \leftarrow \{s\}$ and $d[s] \leftarrow 0$.
- Repeatedly choose unexplored node $v \notin S$ which minimizes

$$\pi(v) = \min_{e=(u,v):u\in S} d[u] + l_e$$

Add v to S; set $d[v] \leftarrow \pi(v)$ and $pred(v) \leftarrow$ *argmin.*

The length of a shortest path from s to some node u in explored part S, followed by a single edge $e = (u, v)$.



Dijkstra's Algorithm: Efficient Implementation

Critical optimization 1. For each unexplored node $v \notin S$: explicitly maintain $\pi[v]$ instead of computing directly from definition

$$\pi(v) = \min_{e=(u,v):u\in S} d[u] + l_e$$

- For each $v \notin S$: $\pi(v)$ can only decrease (because S only increases).
- More specifically, suppose u is added to S and there is an edge $e = (u, v)$ leaving u. Then, it suffices to update:

$$\pi[v] \leftarrow \min\{\pi[v], \pi[u] + l_e\}$$

Recall: for each $u \in S$, $\pi[u] = d[u]$ =length of shortest $s \rightarrow u$ path.

Critical optimization 2. Use a min-oriented priority queue (PQ) to choose an unexplored node that minimizes $\pi[v]$.

Dijkstra's Algorithm: Efficient Implementation

Implementation.

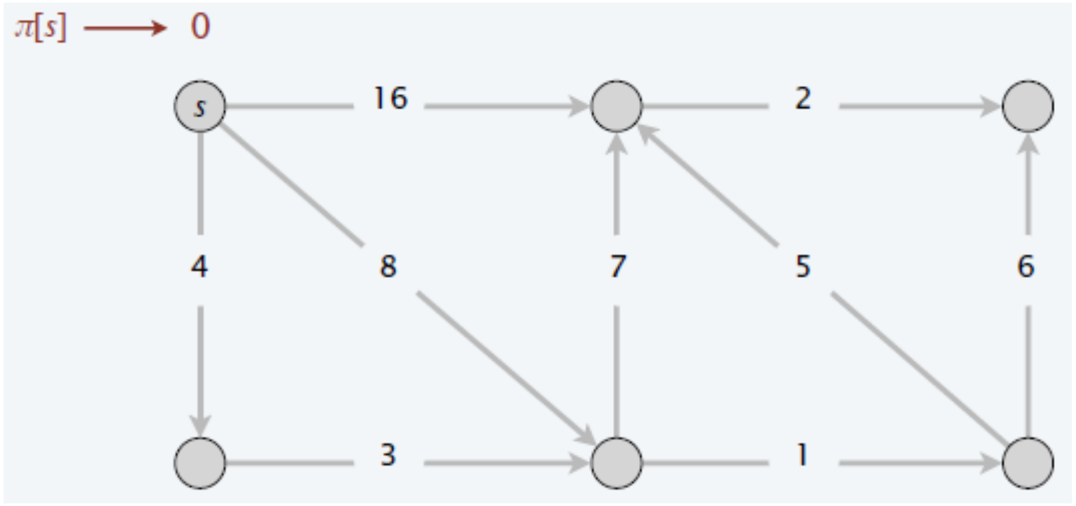
- Algorithm stores $\pi[v]$ for each node v.
- Priority Queue (PQ) stores unexplored nodes, using $\pi[.]$ as priorities.
- Once u is deleted from the PQ, $\pi[u]$ = length of a shortest $s \rightarrow u$ path.

```
Dijkstra $(V, E, l, s)$
Create an empty priority queue PQ.
for each $v \neq s$: $\pi[v] \leftarrow \infty, pred[v] \leftarrow null; \pi[s] \leftarrow 0.$
for each $v \in V$: Insert (PQ, $v, \pi[v]$).
while  Is-Not-Empty (PQ)
    $u \leftarrow$ Del-Min (PQ).
    for each edge $e = (u, v) \in E$ leaving u:
        if $\pi[v] > \pi[u] + l_e$
            Decrease-Key (PQ, $v, \pi[u] + l_e$).
            $\pi[v] \leftarrow \pi[u] + l_e; pred[v] \leftarrow e.$
```

Dijkstra's Algorithm Demo (Efficient Implementation)

Initialization.

- For all $v \neq s$: $\pi[v] \leftarrow \infty$.
- For all $v \neq s$: $pred[v] \leftarrow null$.
- $S \leftarrow \emptyset$ and $\pi[s] \leftarrow 0$.

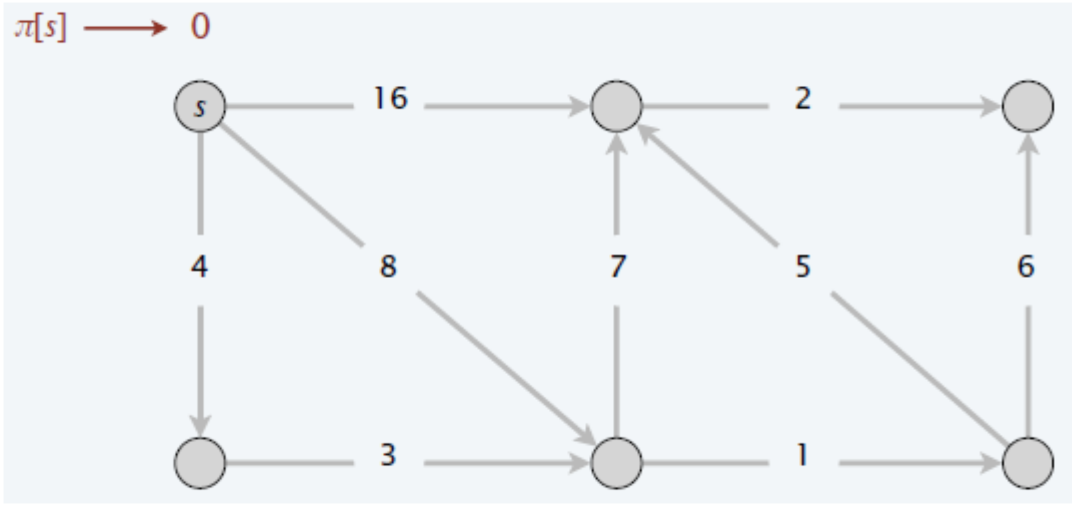


Dijkstra's Algorithm Demo (Efficient Implementation)

Basic step. Choose unexplored node $u \neq s$ with minimum $\pi[u]$.

- Add u to S.
- For each edge $e = (u, v)$ leaving u, if $\pi[v] > \pi[u] + l_e$ then:

- $\pi[v] \leftarrow \pi[u] + l_e$.
- $pred[v] \leftarrow e$.

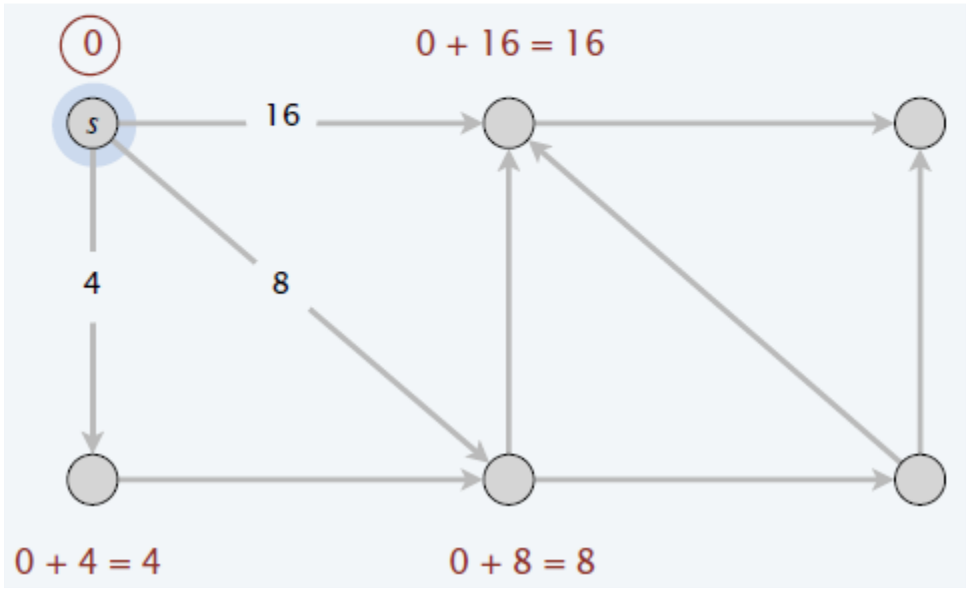


Dijkstra’s Algorithm Demo (Efficient Implementation)

Basic step. Choose unexplored node $u \neq s$ with minimum $\pi[u]$.

- Add u to S.
- For each edge e $= (u, v)$ leaving u, if $\pi[v] > \pi[u] + l_e$ then:

- $\pi[v] \leftarrow \pi[u] + l_e$.
- $pred[v] \leftarrow e$.

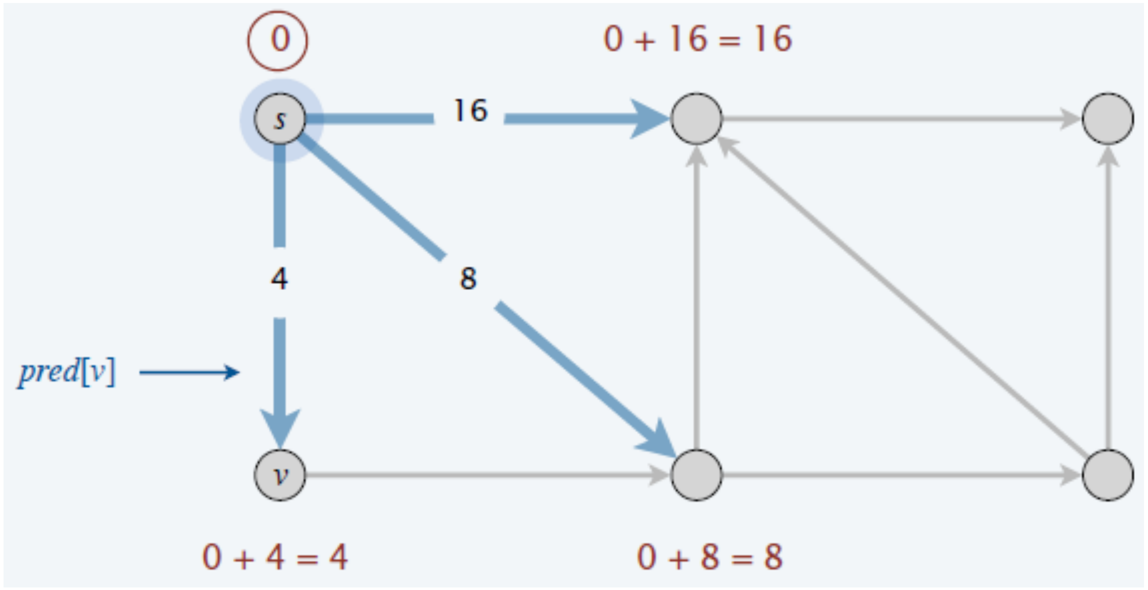


Dijkstra's Algorithm Demo (Efficient Implementation)

Basic step. Choose unexplored node $u \neq s$ with minimum $\pi[u]$.

- Add u to S.
- For each edge $\text{e} = (u, v)$ leaving u, if $\pi[v] > \pi[u] + l_e$ then:

- $\pi[v] \leftarrow \pi[u] + l_e$.
- $pred[v] \leftarrow e$.

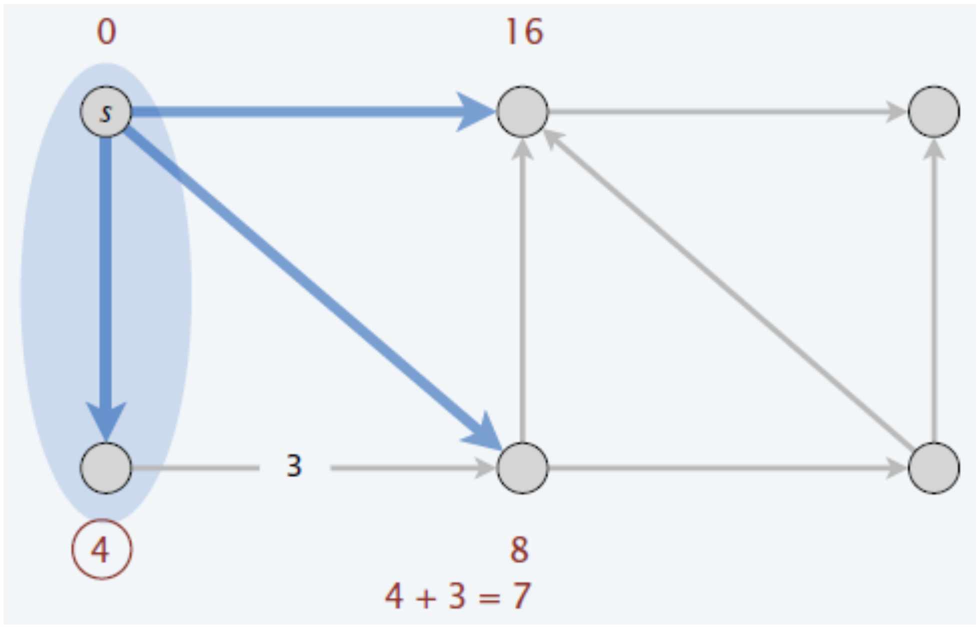


Dijkstra's Algorithm Demo (Efficient Implementation)

Basic step. Choose unexplored node $u \neq s$ with minimum $\pi[u]$.

- Add u to S.
- For each edge e $= (u, v)$ leaving u, if $\pi[v] > \pi[u] + l_e$ then:

- $\pi[v] \leftarrow \pi[u] + l_e$.
- $pred[v] \leftarrow e$.

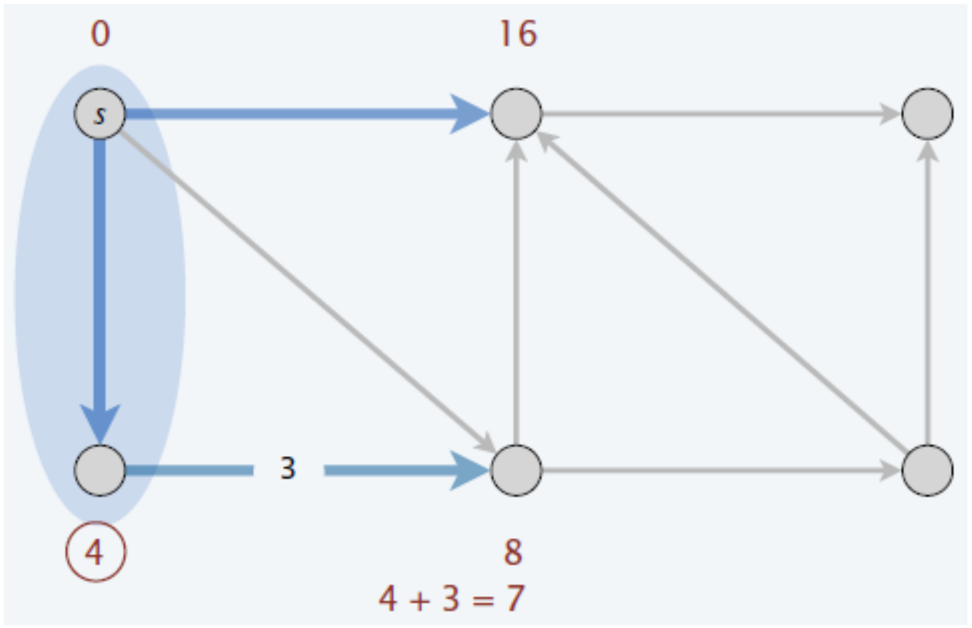


Dijkstra’s Algorithm Demo (Efficient Implementation)

Basic step. Choose unexplored node $u \neq s$ with minimum $\pi[u]$.

- Add u to S.
- For each edge e $= (u, v)$ leaving u, if $\pi[v] > \pi[u] + l_e$ then:

- $\pi[v] \leftarrow \pi[u] + l_e$.
- $pred[v] \leftarrow e$.



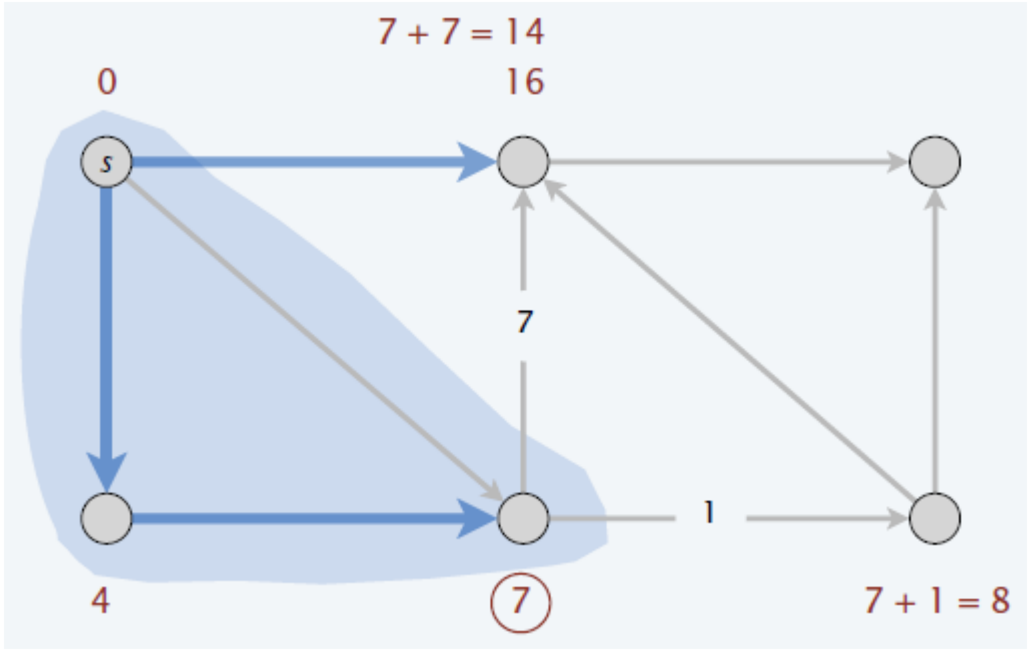
Dijkstra's Algorithm Demo (Efficient Implementation)

Basic step. Choose unexplored node $u \neq s$ with minimum $\pi[u]$.

- Add u to S.
- For each edge $\mathrm{e} = (u, v)$ leaving u, if $\pi[v] > \pi[u] + l_e$ then:

- $\pi[v] \leftarrow \pi[u] + l_e$.

- $pred[v] \leftarrow e$.

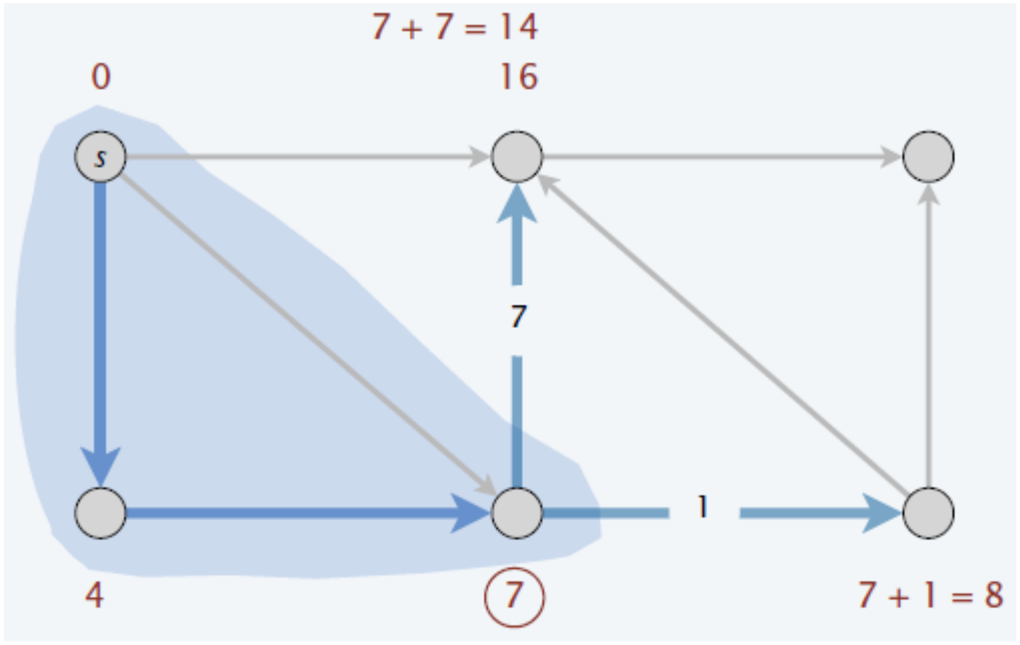


Dijkstra's Algorithm Demo (Efficient Implementation)

Basic step. Choose unexplored node $u \neq s$ with minimum $\pi[u]$.

- Add u to S.
- For each edge $\mathrm{e} = (u, v)$ leaving u, if $\pi[v] > \pi[u] + l_e$ then:

- $\pi[v] \leftarrow \pi[u] + l_e$.
- $pred[v] \leftarrow e$.

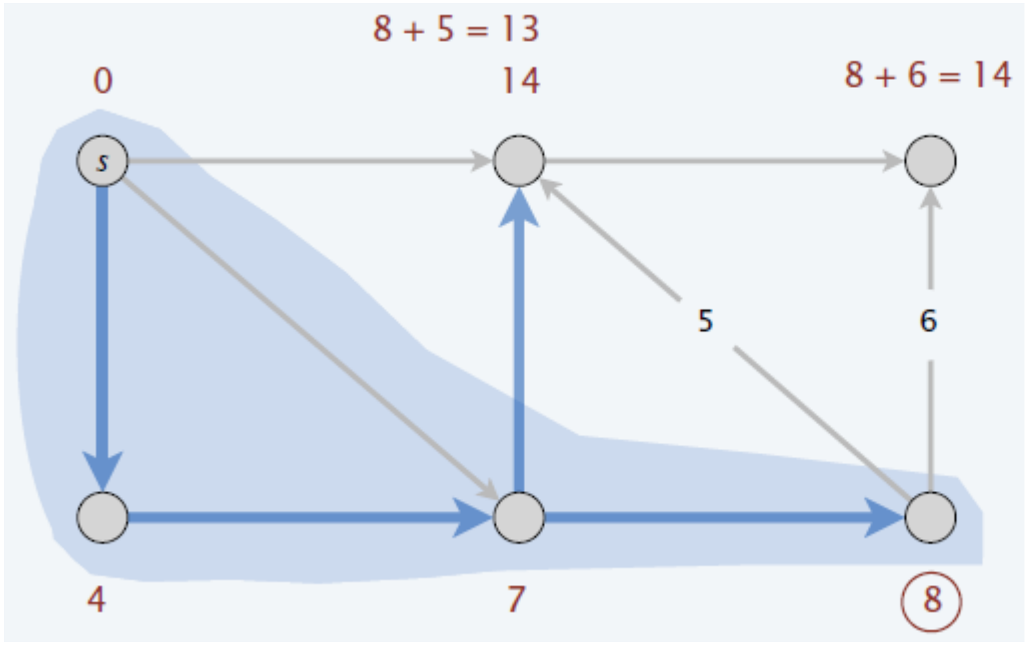


Dijkstra's Algorithm Demo (Efficient Implementation)

Basic step. Choose unexplored node $u \neq s$ with minimum $\pi[u]$.

- Add u to S.
- For each edge $\mathrm{e} = (u, v)$ leaving u, if $\pi[v] > \pi[u] + l_e$ then:

- $\pi[v] \leftarrow \pi[u] + l_e$.
- $pred[v] \leftarrow e$.

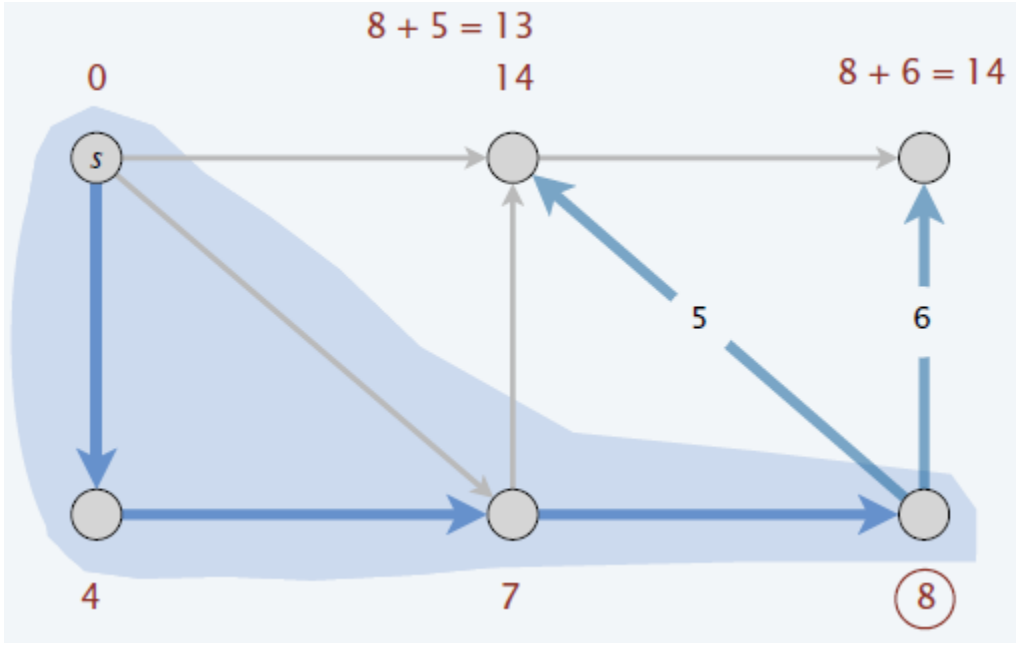


Dijkstra’s Algorithm Demo (Efficient Implementation)

Basic step. Choose unexplored node $u \neq s$ with minimum $\pi[u]$.

- Add u to S.
- For each edge $\mathrm{e} = (u, v)$ leaving u, if $\pi[v] > \pi[u] + l_e$ then:

- $\pi[v] \leftarrow \pi[u] + l_e$.
- $pred[v] \leftarrow e$.

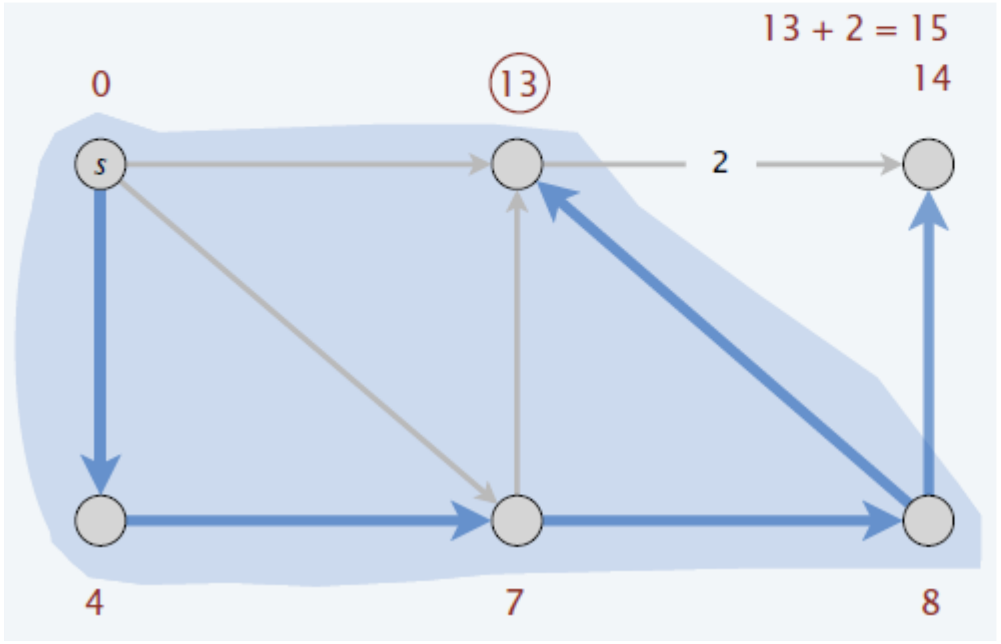


Dijkstra's Algorithm Demo (Efficient Implementation)

Basic step. Choose unexplored node $u \neq s$ with minimum $\pi[u]$.

- Add u to S.
- For each edge $e = (u, v)$ leaving u, if $\pi[v] > \pi[u] + l_e$ then:

- $\pi[v] \leftarrow \pi[u] + l_e$.
- $pred[v] \leftarrow e$.

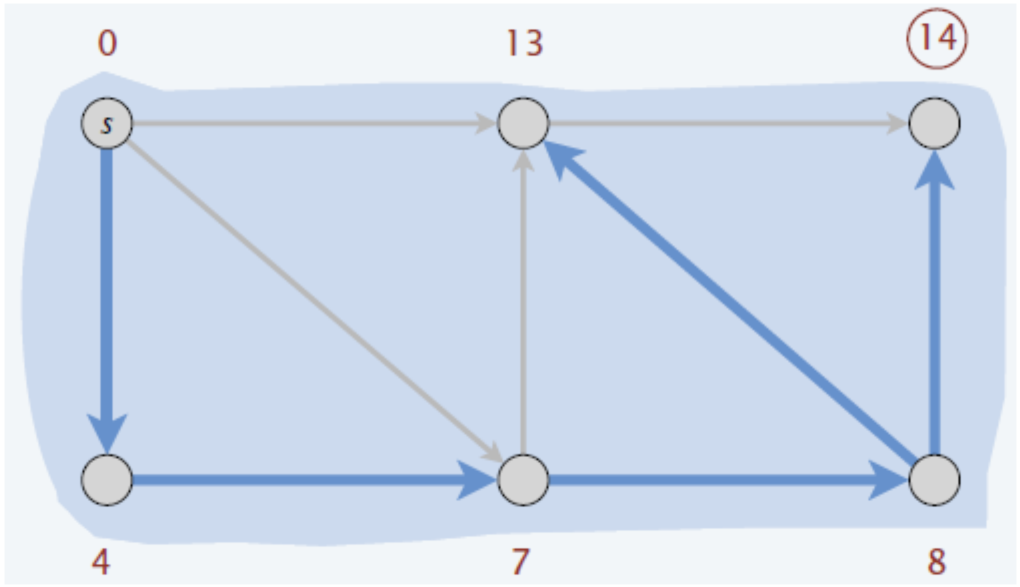


Dijkstra's Algorithm Demo (Efficient Implementation)

Basic step. Choose unexplored node $u \neq s$ with minimum $\pi[u]$.

- Add u to S.
- For each edge e $= (u, v)$ leaving u, if $\pi[v] > \pi[u] + l_e$ then:

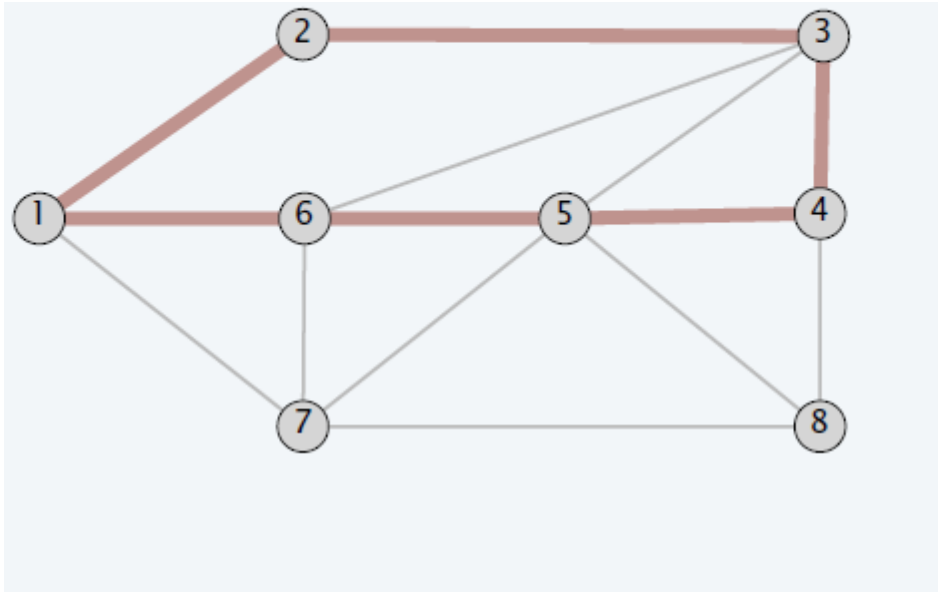
- $\pi[v] \leftarrow \pi[u] + l_e$.
- $pred[v] \leftarrow e$.



Cycles and Cuts

Def. A path is a sequence of edges which connects a sequence of nodes.

Def. A cycle is a path with no repeated nodes or edges other than the starting and ending nodes.

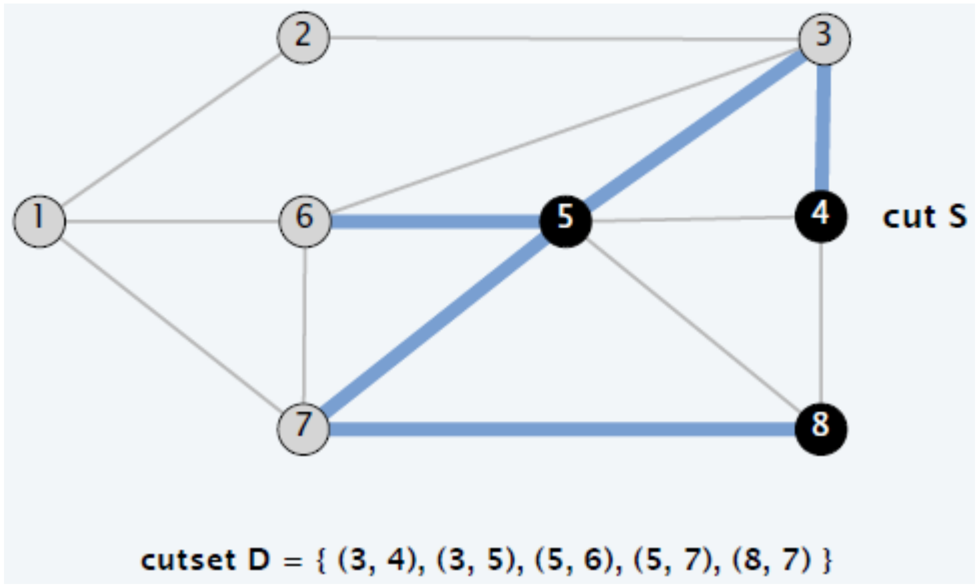


cycle C = { (1, 2), (2, 3), (3, 4), (4, 5), (5, 6), (6, 1) }

Cycles and Cuts

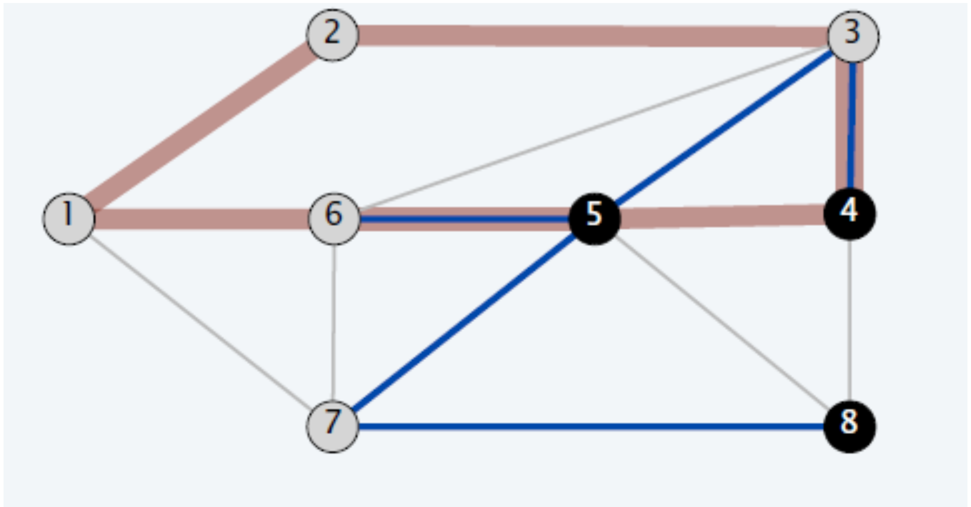
Def. A cut is a partition of the nodes into two nonempty subset S and $V - S$.

Def. The cutset determined by a cut is the set of edges that have one endpoint in each subset of the partition.



Cycle-Cut Intersection

Proposition. A cycle and a cutset intersect in an even number of edges.



cutset D = { (3, 4), (3, 5), (5, 6), (5, 7), (8, 7) }

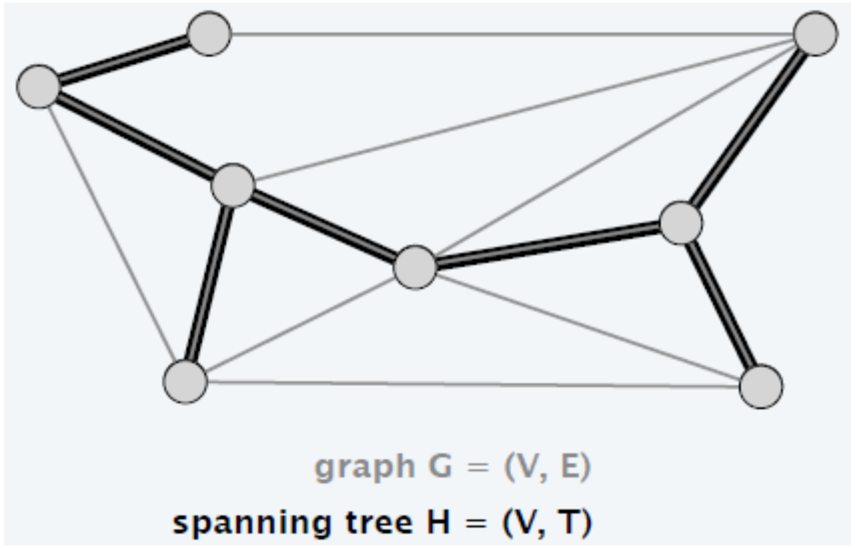
cycle C = { (1, 2), (2, 3), (3, 4), (4, 5), (5, 6), (6, 1) }

intersection $C \cap D$ = { (3, 4), (5, 6) }

Spanning Tree Definition

连通无环

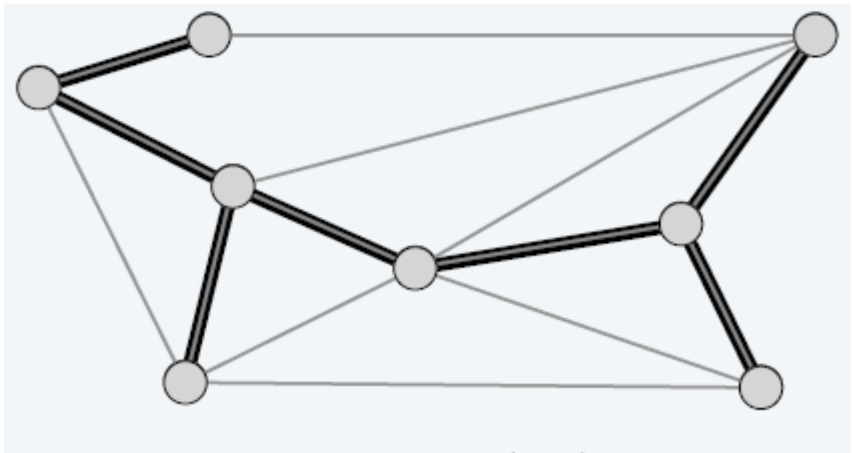
Def. Let $H = (V, T)$ be a subgraph of an undirected graph $G = (V, E)$. H is a spanning tree of G if H is both acyclic and connected.



Spanning Tree Properties

Proposition. Let $H = (V, T)$ be a subgraph of an undirected graph $G = (V, E)$. Then, the following are equivalent:

- H is a spanning tree of G.
- H is acyclic and connected.
- H is connected and has $n - 1$ edges.
- H is acyclic and has $n - 1$ edges.
- H is minimally connected: removal of any edge disconnects it.
- H is maximally acyclic: addition of any edge creates a cycle.

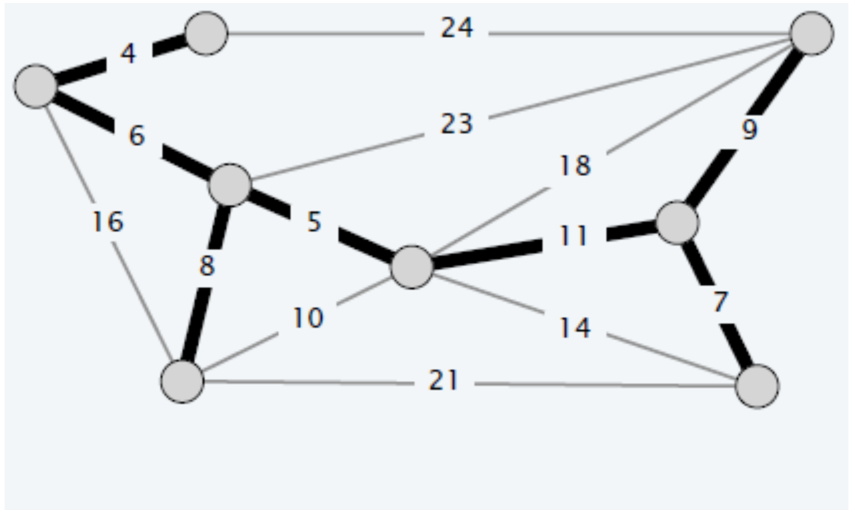


graph $G = (V, E)$

spanning tree $H = (V, T)$

Minimum Spanning Tree (MST)

Def. Given a connected, undirected graph $G = (V, E)$ with edge costs c_e, a minimum spanning tree (V, T) is a spanning tree of G such that the sum of the edge costs in T is minimized.



MST cost = 50 = 4 + 6 + 8 + 5 + 11 + 9 + 7

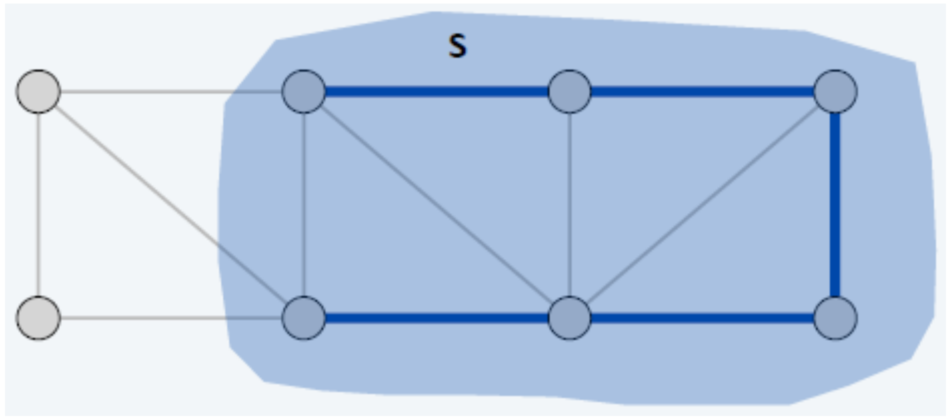
Prim's Algorithm 保证连通再选

Initialize $S =$ any node, $T = \emptyset$.

Repeat $n - 1$ times:

- Add to T a min-weight edge with one endpoint in S.
- Add new node to S.

Theorem. Prim's algorithm computes an MST.



Prim's Algorithm: Implementation

Implementation almost identical to Dijkstra's algorithm.

$\pi[v]$ = weight of cheapest known edge between v and S.

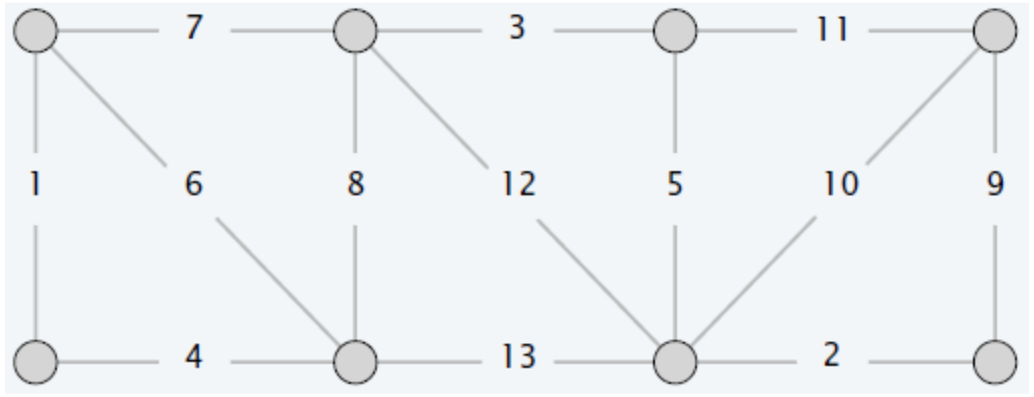
```
Prim $(V, E, c)$
Create an empty priority queue $PQ$.
$S \leftarrow \emptyset, T \leftarrow \emptyset$.
$s \leftarrow$ any node in $V$.
for each $v \neq s$: $\pi[v] \leftarrow \infty, pred[v] \leftarrow null$; $\pi[s] \leftarrow 0$.
for each $v \in V$: Insert $(PQ, v, \pi[v])$,
while Is-Not-Empty $(PQ)$
   $u \leftarrow$ Del-Min $(PQ)$.
   $S \leftarrow S \cup \{u\}, T \leftarrow T \cup \{pred[u]\}$.
   for each edge $e = (u, v) \in E$ with $v \notin S$:
      if $c_e < \pi[v]$
         Decrease-Key $(PQ, v, c_e)$.
         $\pi[v] \leftarrow c_e$; $pred[v] \leftarrow e$.
```

Prim's Algorithm Demo

Initialize S=any node, $T = \emptyset$

Repeat n-1 times:

- Add to T a min-weight edge with one endpoint in S.
- Add new node to S.

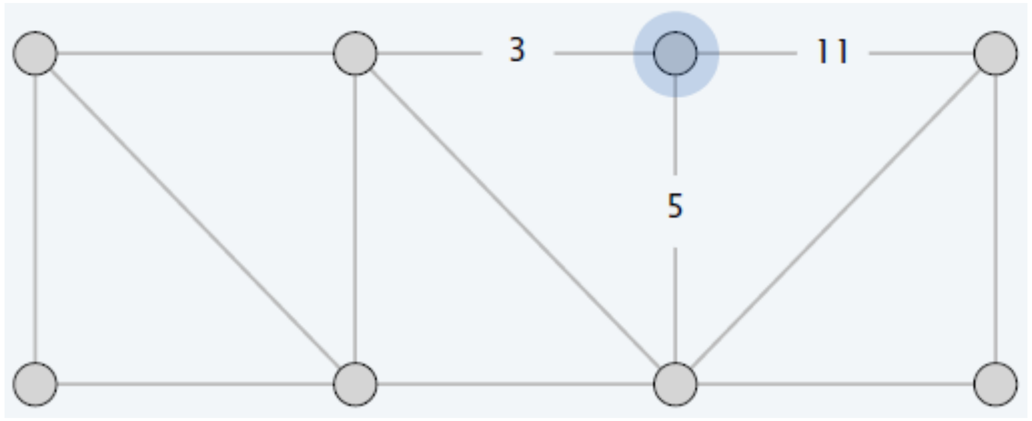


Prim's Algorithm Demo

Initialize S=any node, $T = \emptyset$

Repeat n-1 times:

- Add to T a min-weight edge with one endpoint in S.
- Add new node to S.

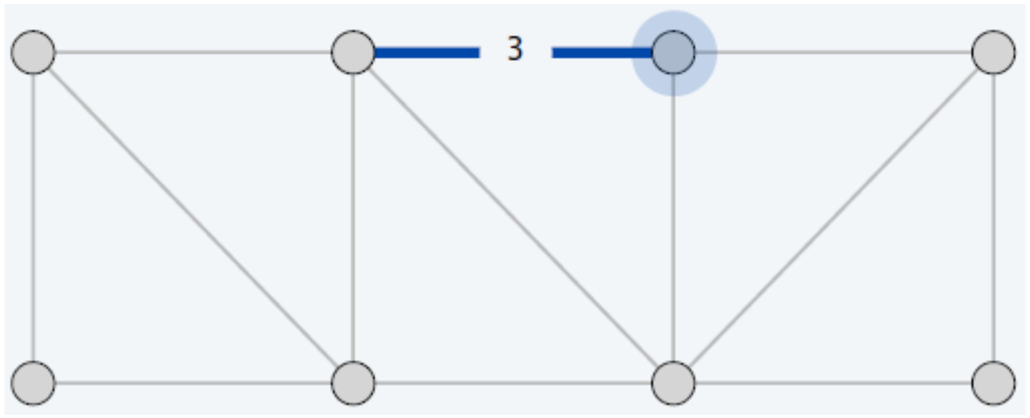


Prim’s Algorithm Demo

Initialize S=any node, $T = \emptyset$

Repeat n-1 times:

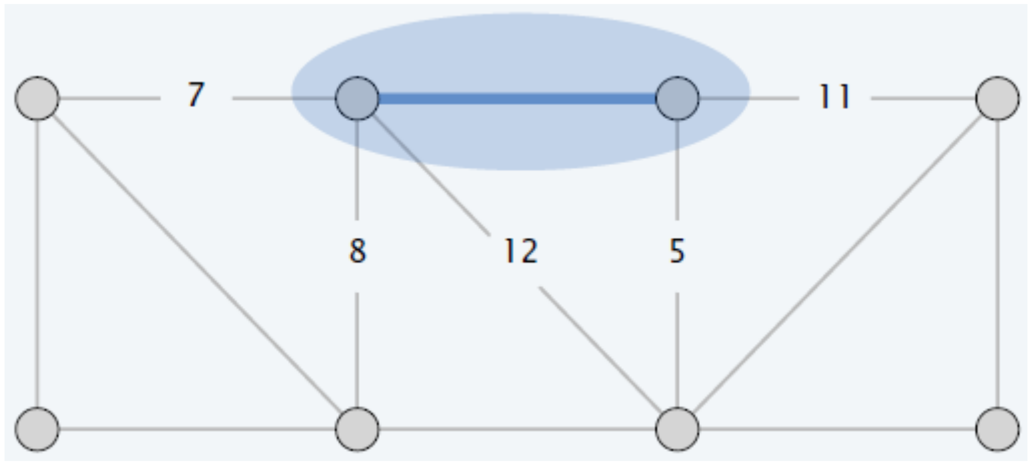
- Add to T a min-weight edge with one endpoint in S.
- Add new node to S.



Prim's Algorithm Demo

Initialize S=any node, $T = \emptyset$
Repeat n-1 times:

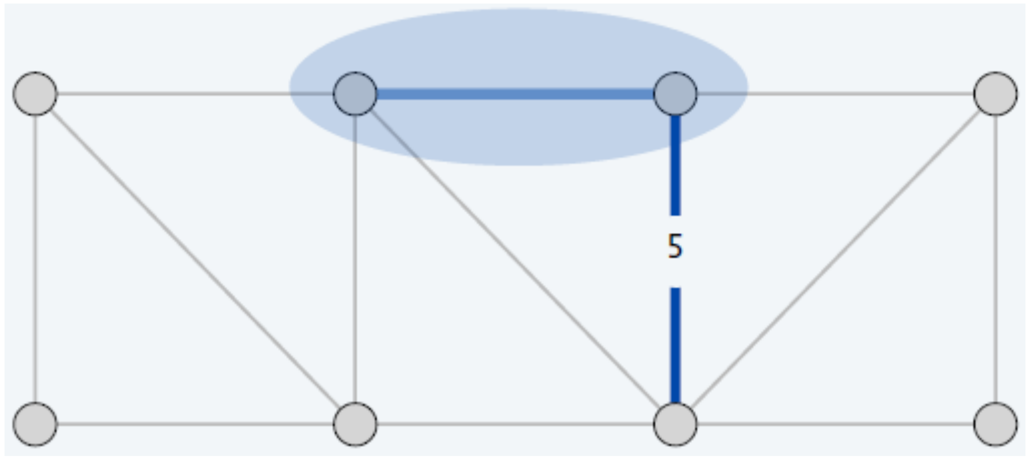
- Add to T a min-weight edge with one endpoint in S.
- Add new node to S.



Prim's Algorithm Demo

Initialize S=any node, $T = \emptyset$
Repeat n-1 times:

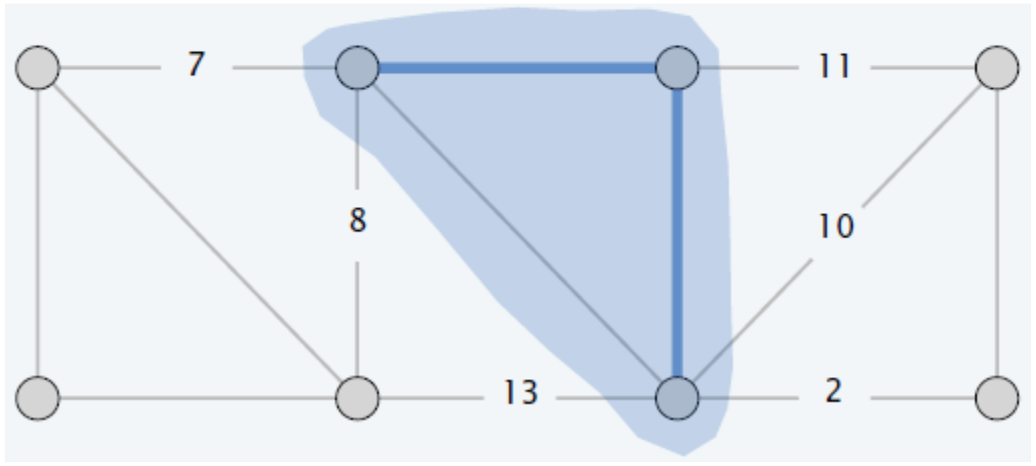
- Add to T a min-weight edge with one endpoint in S.
- Add new node to S.



Prim's Algorithm Demo

Initialize S=any node, $T = \emptyset$
Repeat n-1 times:

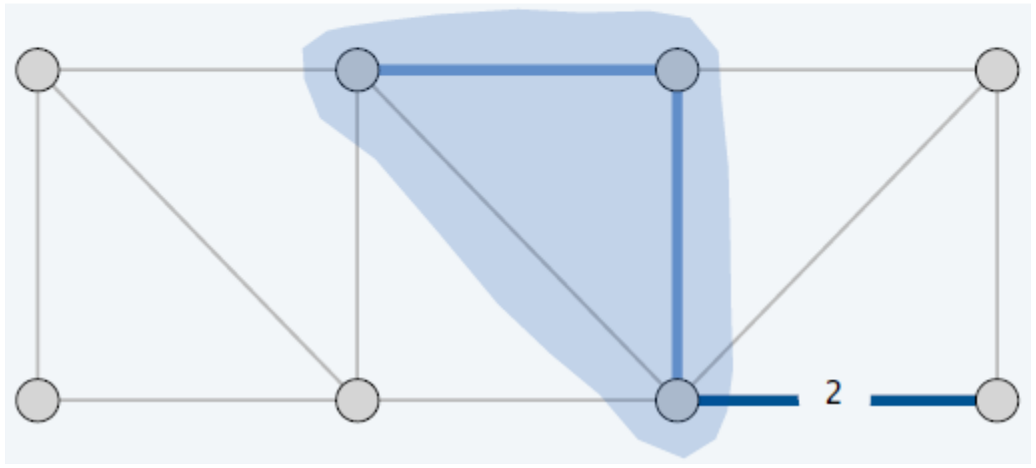
- Add to T a min-weight edge with one endpoint in S.
- Add new node to S.



Prim's Algorithm Demo

Initialize S=any node, $T = \emptyset$
Repeat n-1 times:

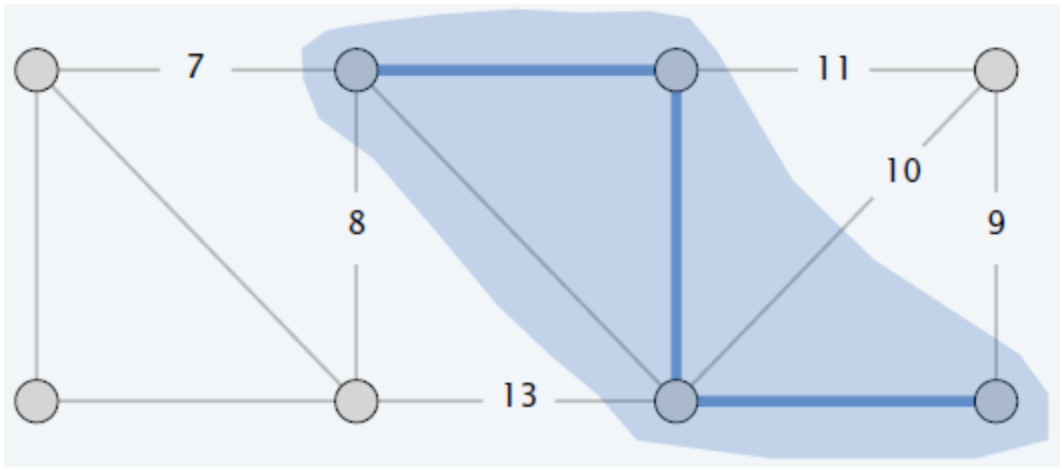
- Add to T a min-weight edge with one endpoint in S.
- Add new node to S.



Prim's Algorithm Demo

Initialize S=any node, $T = \emptyset$
Repeat n-1 times:

- Add to T a min-weight edge with one endpoint in S.
- Add new node to S.

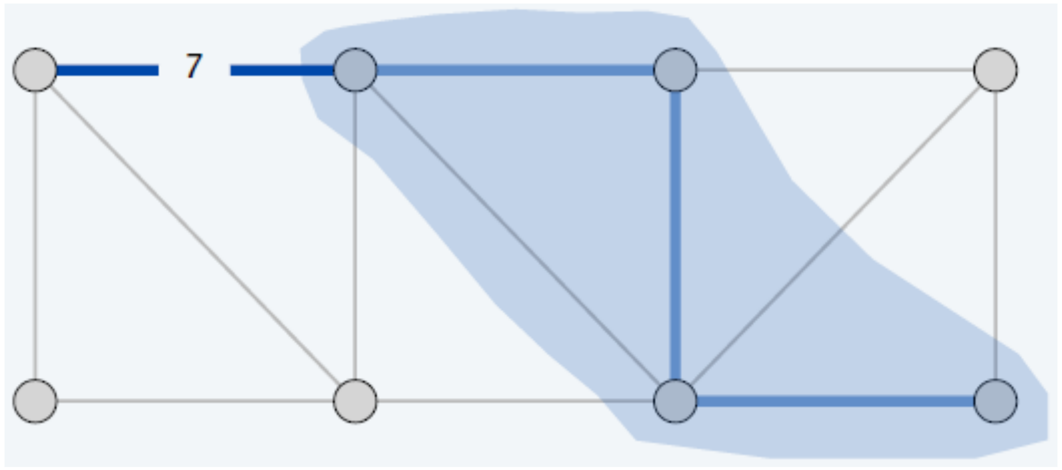


Prim's Algorithm Demo

Initialize S=any node, $T = \emptyset$

Repeat n-1 times:

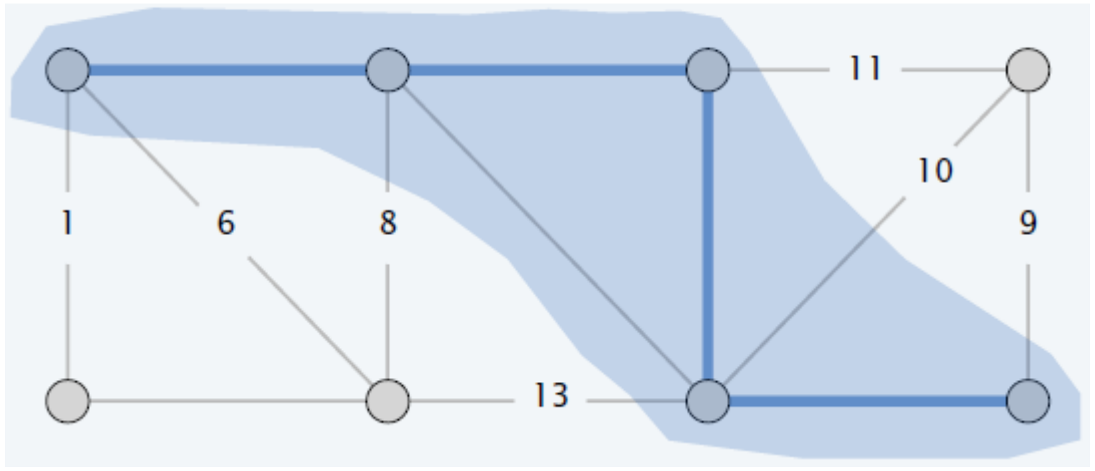
- Add to T a min-weight edge with one endpoint in S.
- Add new node to S.



Prim's Algorithm Demo

Initialize S=any node, $T = \emptyset$
Repeat n-1 times:

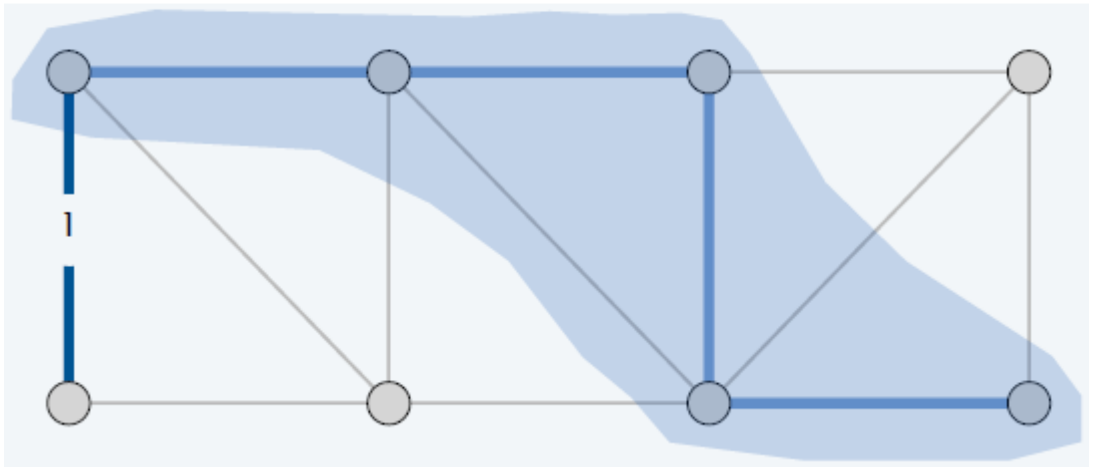
- Add to T a min-weight edge with one endpoint in S.
- Add new node to S.



Prim's Algorithm Demo

Initialize S=any node, $T = \emptyset$
Repeat n-1 times:

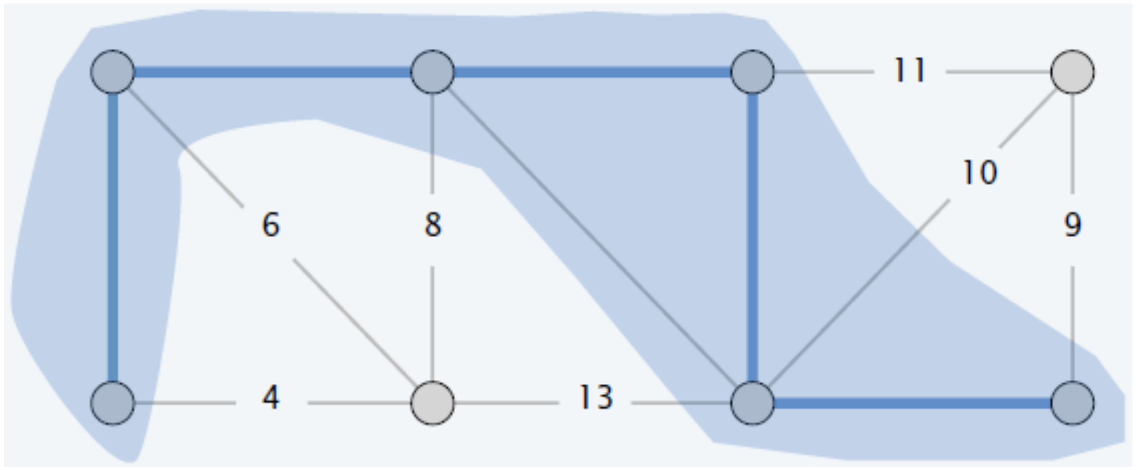
- Add to T a min-weight edge with one endpoint in S.
- Add new node to S.



Prim's Algorithm Demo

Initialize S=any node, $T = \emptyset$
Repeat n-1 times:

- Add to T a min-weight edge with one endpoint in S.
- Add new node to S.

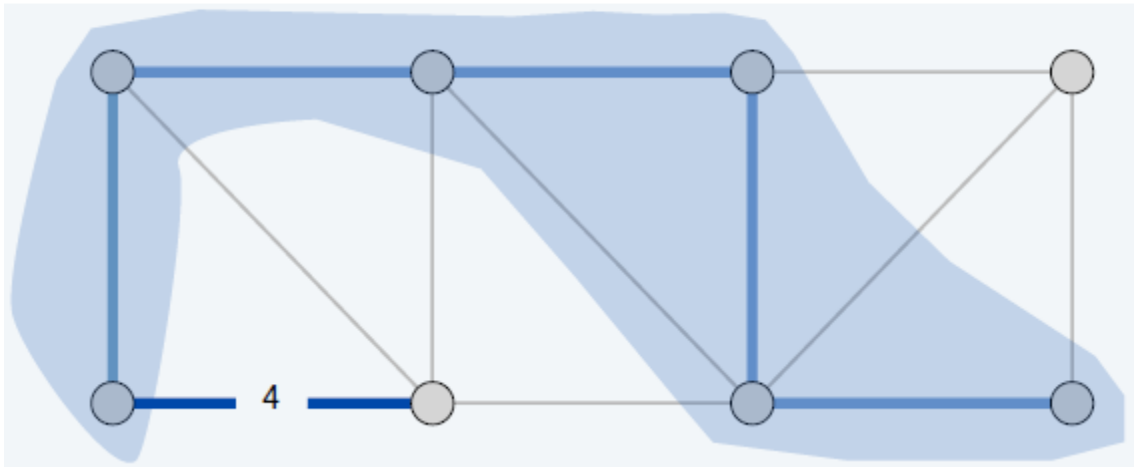


Prim's Algorithm Demo

Initialize S=any node, $T = \emptyset$

Repeat n-1 times:

- Add to T a min-weight edge with one endpoint in S.
- Add new node to S.

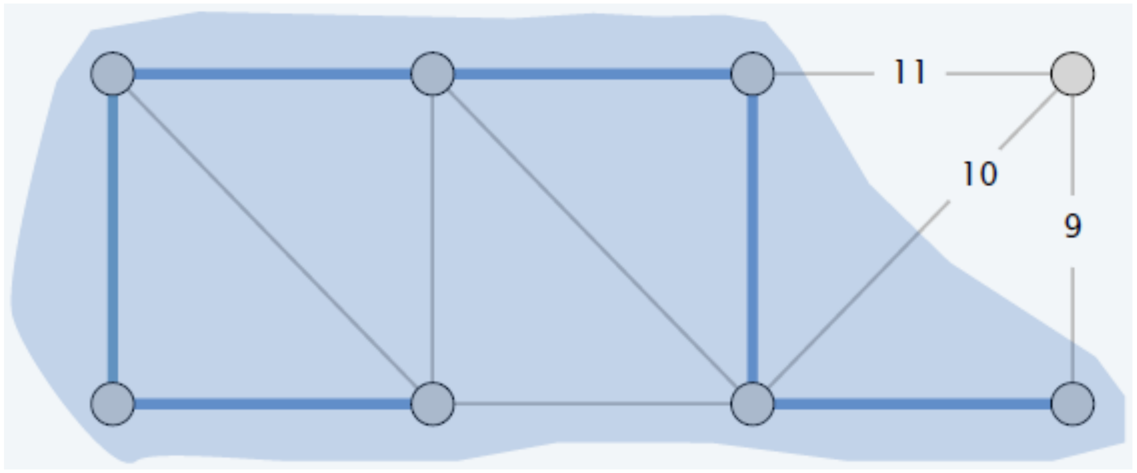


Prim's Algorithm Demo

Initialize S=any node, $T = \emptyset$

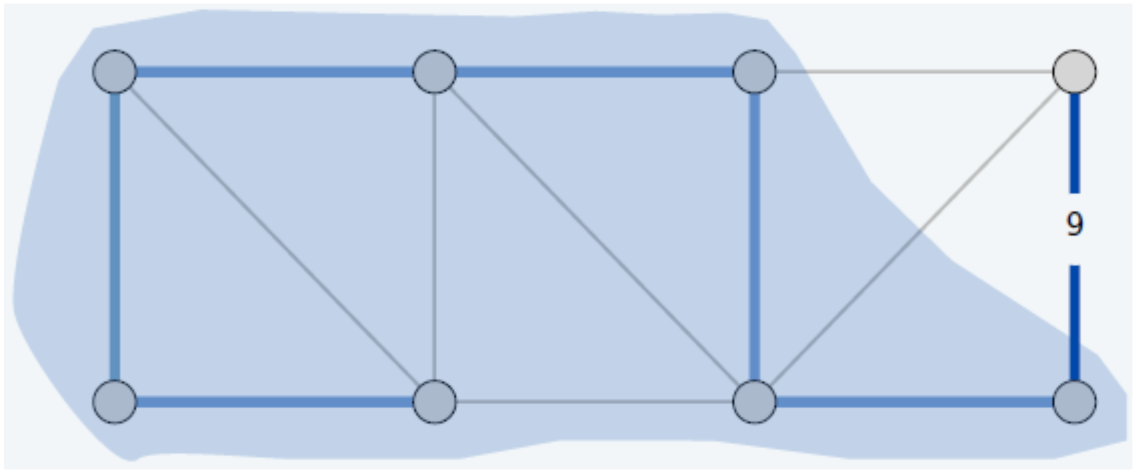
Repeat n-1 times:

- Add to T a min-weight edge with one endpoint in S.
- Add new node to S.



Prim's Algorithm Demo

Initialize S=any node, $T = \emptyset$
Repeat n-1 times:
- Add to T a min-weight edge with one endpoint in S.
- Add new node to S.

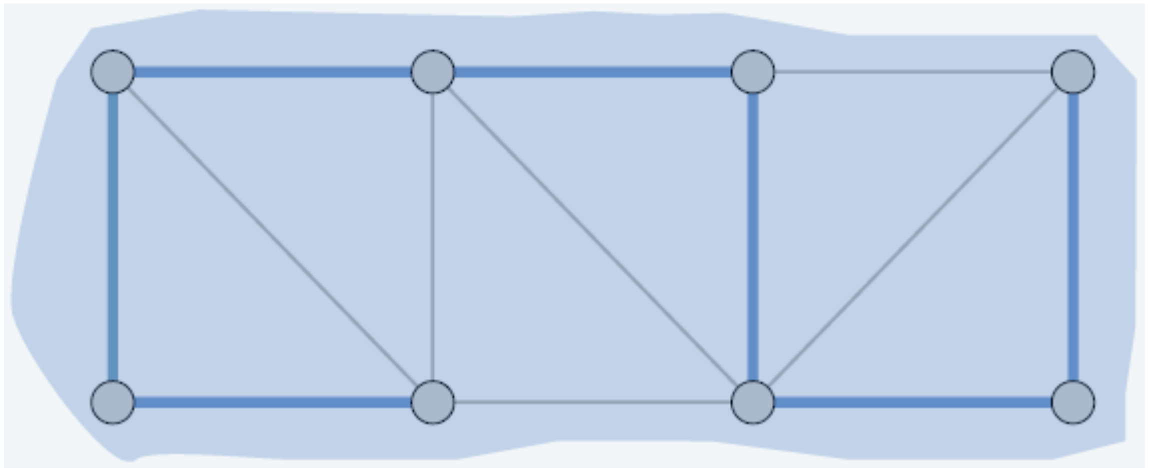


Prim’s Algorithm Demo

Initialize S=any node, $T = \emptyset$

Repeat n-1 times:

- Add to T a min-weight edge with one endpoint in S.
- Add new node to S.

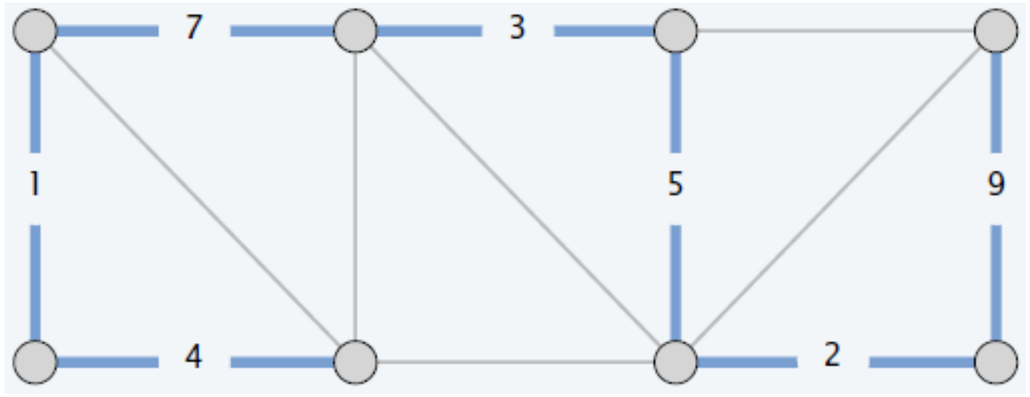


Prim's Algorithm Demo

Initialize S=any node, $T = \emptyset$

Repeat n-1 times:

- Add to T a min-weight edge with one endpoint in S.
- Add new node to S.

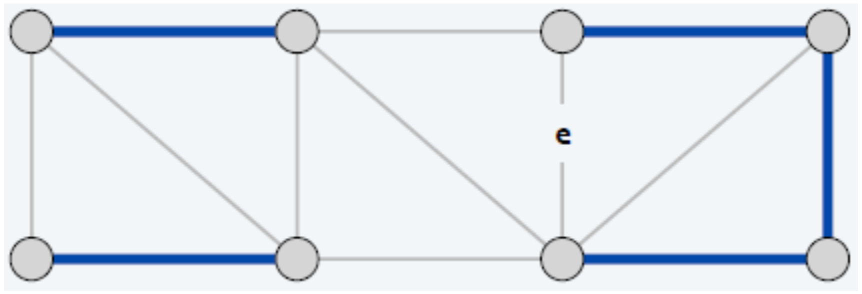


Kruskal’s Algorithm

Consider edges in ascending order of weight:

- Add to tree unless it would create a cycle.

Theorem. Kruskal’s algorithm computes an MST.



Kruskal's Algorithm: Implementation

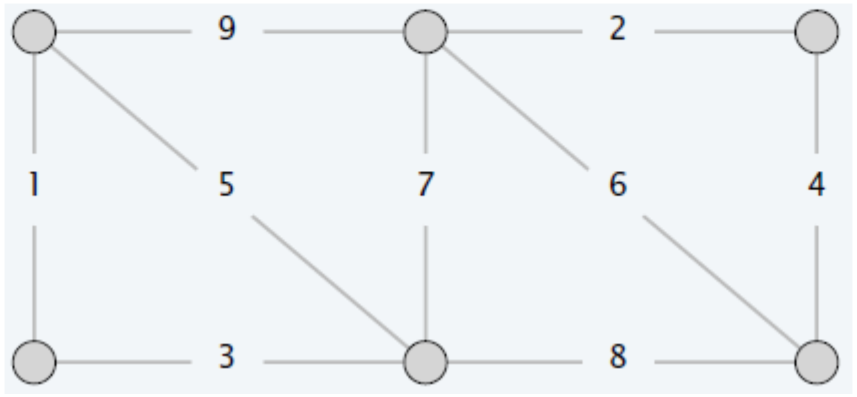
- Sort edges by weights.
- Use union-find data structure to dynamically maintain connected components.

```
Kruskal $(V, E, c)$
Sort $m$ edges by weight so that $c(e_1) \le c(e_1) \le \cdots \le c(e_m)$.
$T \leftarrow \emptyset$.
for each $v \in V$: Make-Set $(v)$.
for $i = 1$ to $m$
    $(u, v) \leftarrow e_i$.
    if Find-Set $(u) \neq$ Find-Set $(v)$      ← are $u$ and $v$ in same component?
        $T \leftarrow T \cup \{e_i\}$.
        Union $(u, v)$.      ← make $u$ and $v$ in same component
Return $T$.
```

Kruskal’s Algorithm Demo

Consider edges in ascending order of weight:

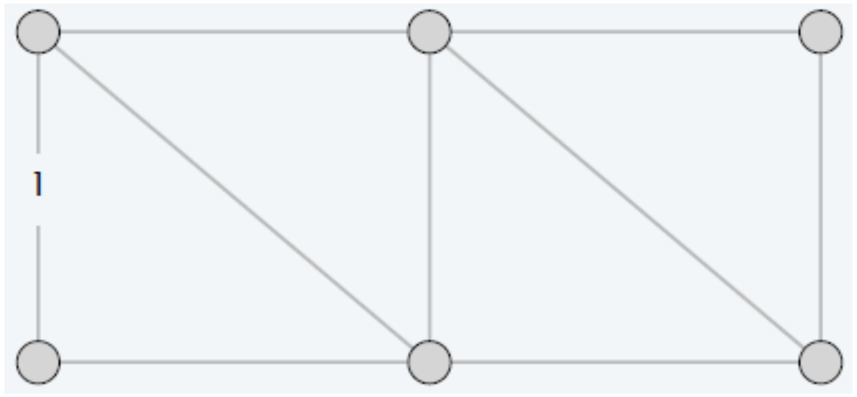
- Add to T unless it would create a cycle.



Kruskal’s Algorithm Demo

Consider edges in ascending order of weight:

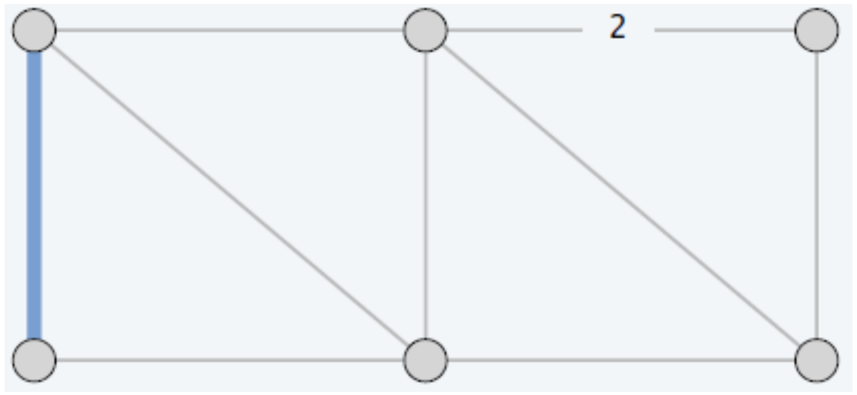
- Add to T unless it would create a cycle.



Kruskal’s Algorithm Demo

Consider edges in ascending order of weight:

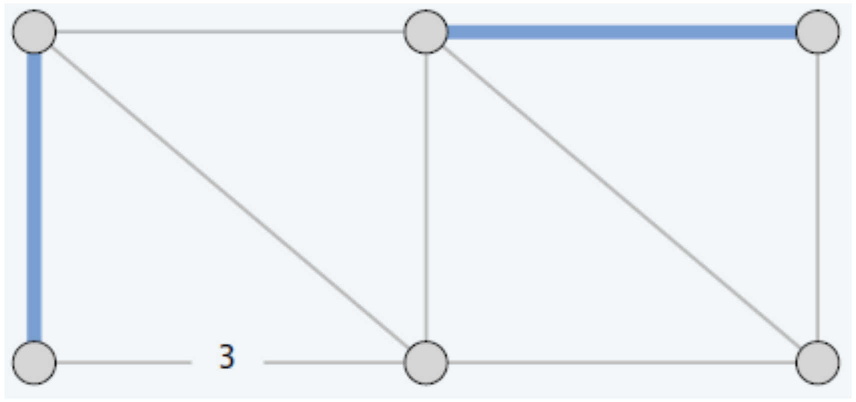
- Add to T unless it would create a cycle.



Kruskal’s Algorithm Demo

Consider edges in ascending order of weight:

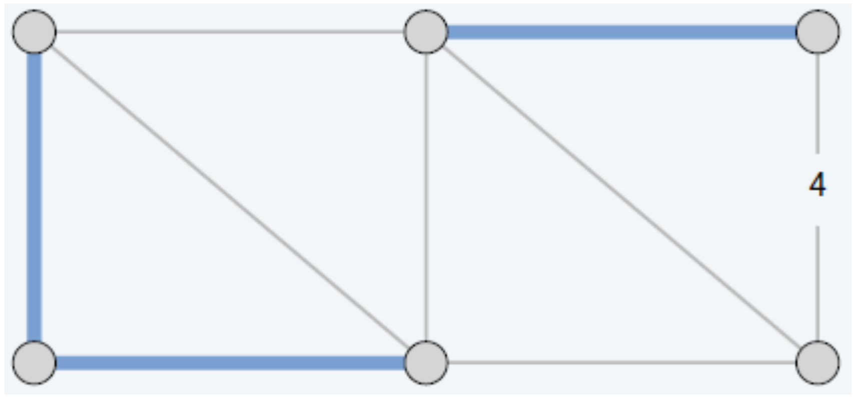
- Add to T unless it would create a cycle.



Kruskal’s Algorithm Demo

Consider edges in ascending order of weight:

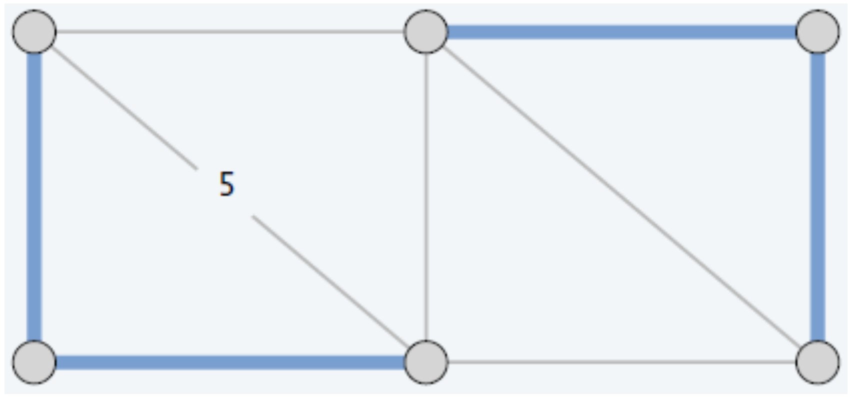
- Add to T unless it would create a cycle.



Kruskal’s Algorithm Demo

Consider edges in ascending order of weight:

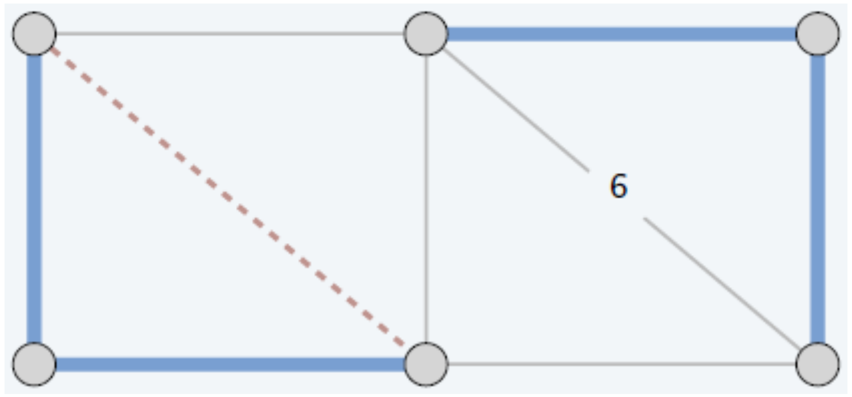
- Add to T unless it would create a cycle.



Kruskal's Algorithm Demo

Consider edges in ascending order of weight:

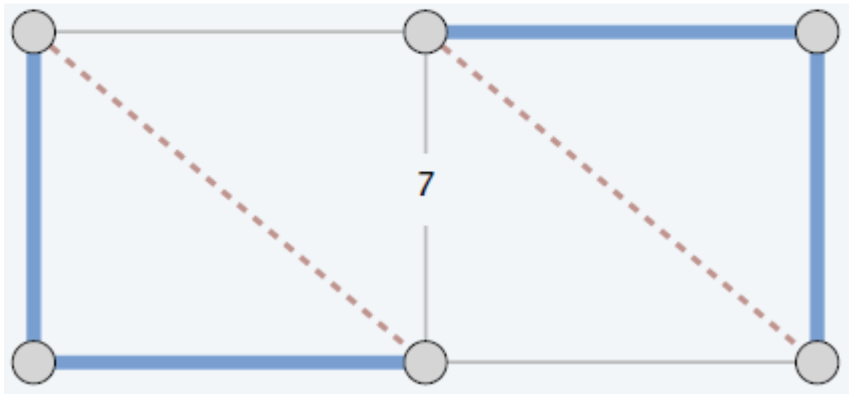
- Add to T unless it would create a cycle.



Kruskal’s Algorithm Demo

Consider edges in ascending order of weight:

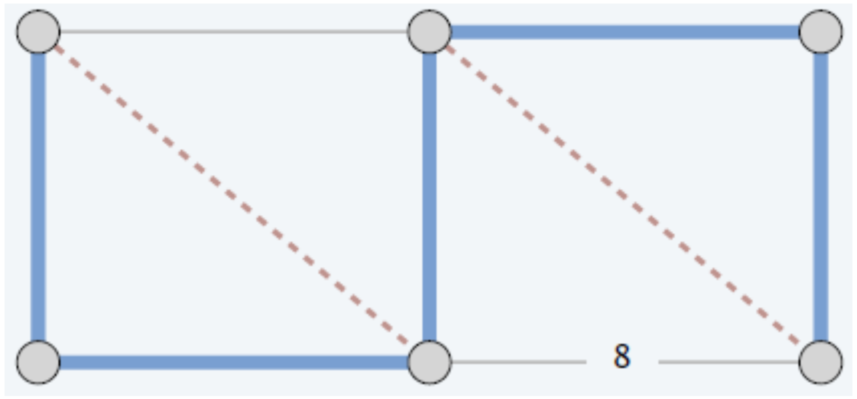
- Add to T unless it would create a cycle.



Kruskal’s Algorithm Demo

Consider edges in ascending order of weight:

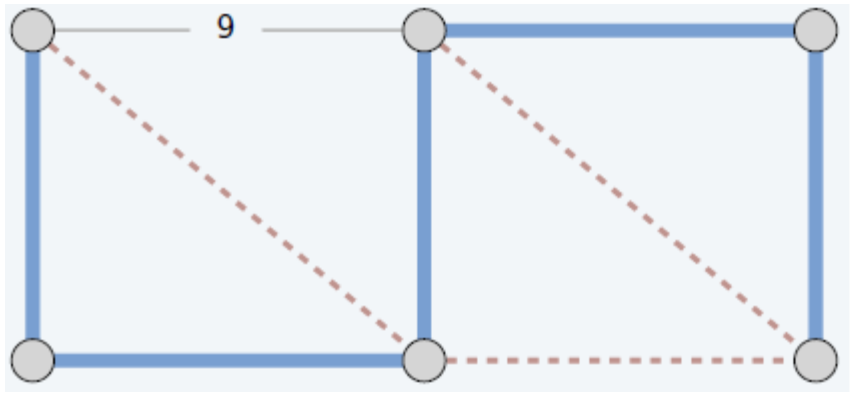
- Add to T unless it would create a cycle.



Kruskal's Algorithm Demo

Consider edges in ascending order of weight:

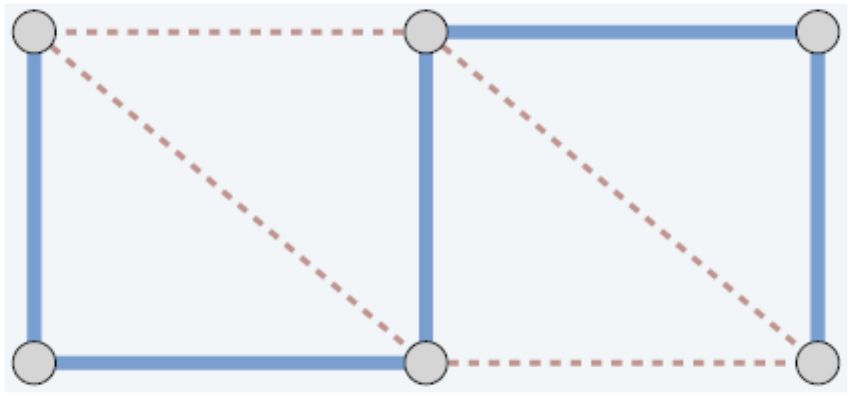
- Add to T unless it would create a cycle.



Kruskal's Algorithm Demo

Consider edges in ascending order of weight:

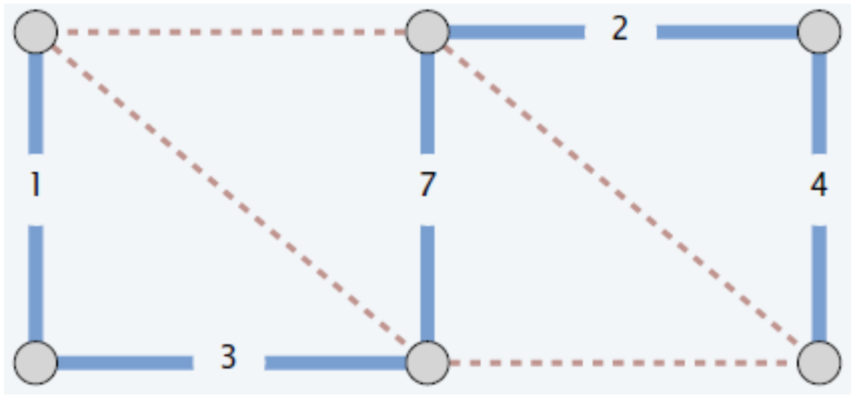
- Add to T unless it would create a cycle.



Kruskal's Algorithm Demo

Consider edges in ascending order of weight:

- Add to T unless it would create a cycle.



Proof of Kruskal’s Algorithm

Theorem. After running Kruskal’s algorithm on a connected weight graph G, its output T is a minimum weight spanning tree.

Proof of Kruskal's Algorithm

Theorem. After running Kruskal's algorithm on a connected weight graph G, its output T is a minimum weight spanning tree.

Proof. First, T is a spanning tree. This is because:

- T is a acyclic.
- T is spanning.
- T is connected.

Second, T is a spanning tree of minimum weight. We can prove this using induction:
Let T^* be a minimum-weight spanning tree. If $T = T^*$, then T is a minimum weight spanning tree. If $T \neq T^*$, then there exist an edge $e \in T^*$ of minimum weight that is not in T. Further, $T \cup \{e\}$ contains a cycle C such that:

a. Every edge in C has weight less than $weight(e)$. (This follows from how the algorithm constructed T.)

Proof of Kruskal's Algorithm

Theorem. After running Kruskal's algorithm on a connected weight graph G, its output T is a minimum weight spanning tree.

If $T = T^*$, then there exist an edge $e \in T^*$ of minimum weight that is not in T. Further, $T \cup \{e\}$ contains a cycle C such that:

a. Other edges in C have weights less than $weight(e)$. (This follows from how the algorithm constructed T.)
b. There is some edge f in C that is not in T^*. (Because T^* does not contain the cycle C.) Consider the tree $T_2 = T \cup \{e\} \backslash \{f\}$:
c. T_2 is a spanning tree.
d. T_2 has more edges in common with T^* than T did.
e. And $weight(T_2) \geq weight(T)$. (We exchanged an edge for one that is no more expensive.)

We can redo the same process with T_2 to find a spanning tree T_3 with more edge in common with T^*.

Proof of Kruskal's Algorithm

Theorem. After running Kruskal's algorithm on a connected weight graph G, its output T is a minimum weight spanning tree.

We can redo the same process with T_2 to find a spanning tree T_3 with more edge in common with T^*. By induction, we can continue this process until we reach T^*, from which we see

$$weight(T) \leq weight(T_2) \leq weight(T_3) \leq \cdots \leq weight(T^*)$$

Since T^* is a minimum weight spanning tree, then these inequalities must be equalities and we conclude that T is a minimum weight spanning tree.