Graphs

SFWRENG 2CO3: Data Structures and Algorithms

Jelle Hellings

Department of Computing and Software
McMaster University

Winter 2024

Undirected trees

Definition

Let $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ be an undirected graph.
The graph $\mathcal{G}$ is an *undirected tree* if:

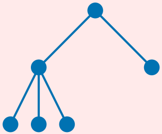
- the graph is connected
  (every pair of nodes is connected by a path);
- the graph has $|\mathcal{E}| = |\mathcal{N}| - 1$:
  (removing any one edge will make the graph unconnected).

Undirected trees

Definition

Let $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ be an undirected graph.
The graph $\mathcal{G}$ is an *undirected tree* if:

- the graph is connected
  (every pair of nodes is connected by a path);
- the graph has $|\mathcal{E}| = |\mathcal{N}| - 1$:
  (removing any one edge will make the graph unconnected).



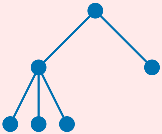
Tree

Undirected trees

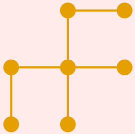
Definition

Let $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ be an undirected graph.
The graph $\mathcal{G}$ is an *undirected tree* if:

- the graph is connected
  (every pair of nodes is connected by a path);
- the graph has $|\mathcal{E}| = |\mathcal{N}| - 1$:
  (removing any one edge will make the graph unconnected).



Tree



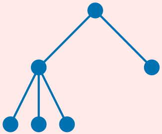
Tree

Undirected trees

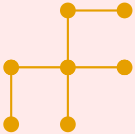
Definition

Let $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ be an undirected graph.
The graph $\mathcal{G}$ is an *undirected tree* if:

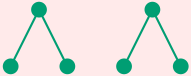
- the graph is connected
  (every pair of nodes is connected by a path);
- the graph has $|\mathcal{E}| = |\mathcal{N}| - 1$:
  (removing any one edge will make the graph unconnected).



Tree



Tree



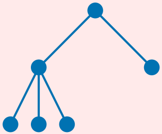
Not a tree
(not connected)

Undirected trees

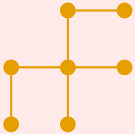
Definition

Let $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ be an undirected graph.
The graph $\mathcal{G}$ is an *undirected tree* if:

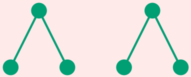
- the graph is connected
  (every pair of nodes is connected by a path);
- the graph has $|\mathcal{E}| = |\mathcal{N}| - 1$:
  (removing any one edge will make the graph unconnected).



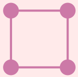
Tree



Tree



Not a tree
(not connected)



Not a tree
(cyclic, too many edges)

Spanning trees

Definition

Let $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ be a *connected* undirected graph.
A *spanning tree* of $\mathcal{G}$ is a subgraph $\mathcal{T} = (\mathcal{N}, \mathcal{E}')$ such that

- $\mathcal{T}$ is a tree; and
- $\mathcal{E}' \subseteq \mathcal{E}$.

Spanning trees

Definition

Let $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ be a *connected* undirected graph.
A *spanning tree* of $\mathcal{G}$ is a subgraph $\mathcal{T} = (\mathcal{N}, \mathcal{E}')$ such that

- $\mathcal{T}$ is a tree; and
- $\mathcal{E}' \subseteq \mathcal{E}$.

Informal: keep sufficient edges from $\mathcal{G}$ to keep $\mathcal{G}$ connected.

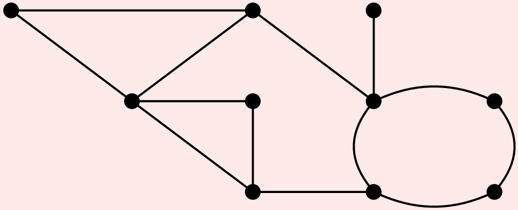
Spanning trees

Definition

Let $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ be a *connected* undirected graph.
A *spanning tree* of $\mathcal{G}$ is a subgraph $\mathcal{T} = (\mathcal{N}, \mathcal{E}')$ such that

- $\mathcal{T}$ is a tree; and
- $\mathcal{E}' \subseteq \mathcal{E}$.

Informal: keep sufficient edges from $\mathcal{G}$ to keep $\mathcal{G}$ connected.



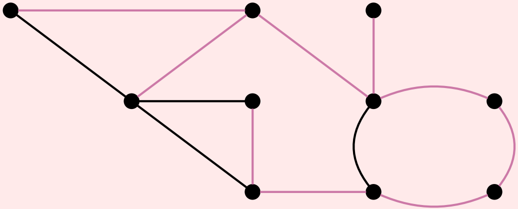
Spanning trees

Definition

Let $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ be a *connected* undirected graph.
A *spanning tree* of $\mathcal{G}$ is a subgraph $\mathcal{T} = (\mathcal{N}, \mathcal{E}')$ such that

- $\mathcal{T}$ is a tree; and
- $\mathcal{E}' \subseteq \mathcal{E}$.

Informal: keep sufficient edges from $\mathcal{G}$ to keep $\mathcal{G}$ connected.



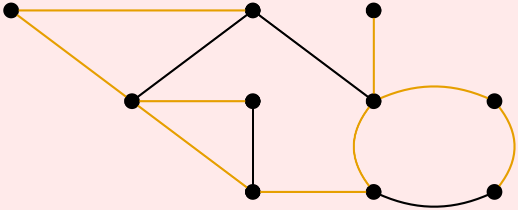
Spanning trees

Definition

Let $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ be a *connected* undirected graph.
A *spanning tree* of $\mathcal{G}$ is a subgraph $\mathcal{T} = (\mathcal{N}, \mathcal{E}')$ such that

- $\mathcal{T}$ is a tree; and
- $\mathcal{E}' \subseteq \mathcal{E}$.

Informal: keep sufficient edges from $\mathcal{G}$ to keep $\mathcal{G}$ connected.



Minimum spanning trees

Definition

Let $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ be a *connected* weighted undirected graph with weight function *weight*.
A *minimum spanning tree* of $\mathcal{G}$ is a subgraph $\mathcal{T} = (\mathcal{N}, \mathcal{E}')$ such that

- $\mathcal{T}$ is a spanning tree;
- the sum of edge weights $\sum_{e \in \mathcal{E}'} \mathit{weight}(e)$ is *minimal*
  (not larger than the sum of edge weights of any other spanning tree of $\mathcal{G}$).

Minimum spanning trees

Definition

Let $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ be a *connected* weighted undirected graph with weight function *weight*.
A *minimum spanning tree* of $\mathcal{G}$ is a subgraph $\mathcal{T} = (\mathcal{N}, \mathcal{E}')$ such that

- $\mathcal{T}$ is a spanning tree;
- the sum of edge weights $\sum_{e \in \mathcal{E}'} \mathit{weight}(e)$ is *minimal*
  (not larger than the sum of edge weights of any other spanning tree of $\mathcal{G}$).

Informal: lowest-cost tree that connects all of $\mathcal{G}$.

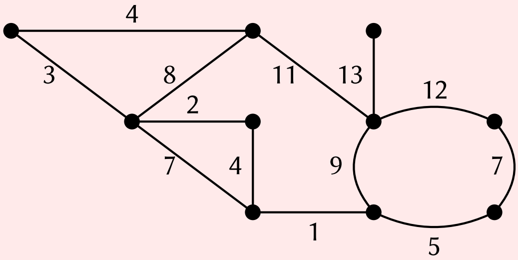
Minimum spanning trees

Definition

Let $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ be a *connected* weighted undirected graph with weight function *weight*.
A *minimum spanning tree* of $\mathcal{G}$ is a subgraph $\mathcal{T} = (\mathcal{N}, \mathcal{E}')$ such that

- $\mathcal{T}$ is a spanning tree;
- the sum of edge weights $\sum_{e \in \mathcal{E}'} weight(e)$ is *minimal*
  (not larger than the sum of edge weights of any other spanning tree of $\mathcal{G}$).

Informal: lowest-cost tree that connects all of $\mathcal{G}$.



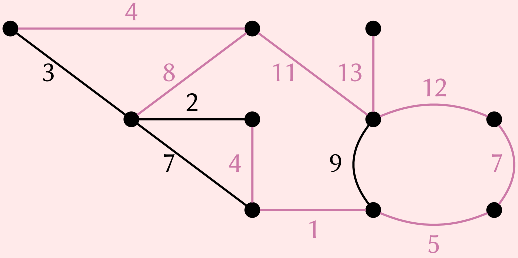
Minimum spanning trees

Definition

Let $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ be a *connected* weighted undirected graph with weight function *weight*.
A *minimum spanning tree* of $\mathcal{G}$ is a subgraph $\mathcal{T} = (\mathcal{N}, \mathcal{E}')$ such that

- $\mathcal{T}$ is a spanning tree;
- the sum of edge weights $\sum_{e \in \mathcal{E}'} weight(e)$ is *minimal* (not larger than the sum of edge weights of any other spanning tree of $\mathcal{G}$).

Informal: lowest-cost tree that connects all of $\mathcal{G}$.



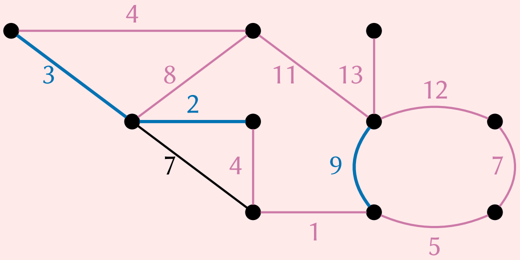
Minimum spanning trees

Definition

Let $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ be a *connected* weighted undirected graph with weight function *weight*.
A *minimum spanning tree* of $\mathcal{G}$ is a subgraph $\mathcal{T} = (\mathcal{N}, \mathcal{E}')$ such that

- $\mathcal{T}$ is a spanning tree;
- the sum of edge weights $\sum_{e \in \mathcal{E}'} weight(e)$ is *minimal* (not larger than the sum of edge weights of any other spanning tree of $\mathcal{G}$).

Informal: lowest-cost tree that connects all of $\mathcal{G}$.



Not minimal!

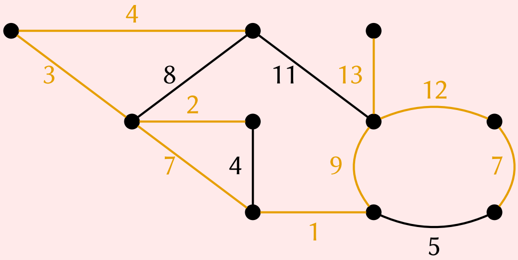
Minimum spanning trees

Definition

Let $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ be a *connected* weighted undirected graph with weight function *weight*.
A *minimum spanning tree* of $\mathcal{G}$ is a subgraph $\mathcal{T} = (\mathcal{N}, \mathcal{E}')$ such that

- $\mathcal{T}$ is a spanning tree;
- the sum of edge weights $\sum_{e \in \mathcal{E}'} \mathit{weight}(e)$ is *minimal*
  (not larger than the sum of edge weights of any other spanning tree of $\mathcal{G}$).

Informal: lowest-cost tree that connects all of $\mathcal{G}$.



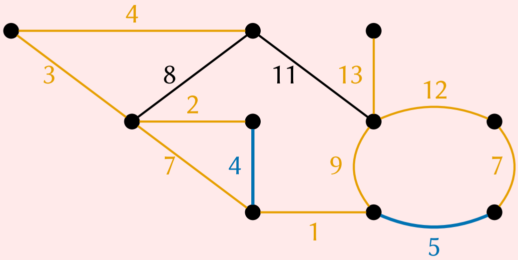
Minimum spanning trees

Definition

Let $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ be a *connected* weighted undirected graph with weight function *weight*.
A *minimum spanning tree* of $\mathcal{G}$ is a subgraph $\mathcal{T} = (\mathcal{N}, \mathcal{E}')$ such that

- $\mathcal{T}$ is a spanning tree;
- the sum of edge weights $\sum_{e \in \mathcal{E}'} \mathit{weight}(e)$ is *minimal*
  (not larger than the sum of edge weights of any other spanning tree of $\mathcal{G}$).

Informal: lowest-cost tree that connects all of $\mathcal{G}$.



Not minimal!

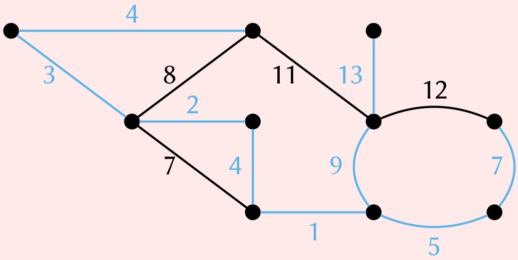
Minimum spanning trees

Definition

Let $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ be a *connected* weighted undirected graph with weight function *weight*.
A *minimum spanning tree* of $\mathcal{G}$ is a subgraph $\mathcal{T} = (\mathcal{N}, \mathcal{E}')$ such that

- $\mathcal{T}$ is a spanning tree;
- the sum of edge weights $\sum_{e \in \mathcal{E}'} \mathit{weight}(e)$ is *minimal* (not larger than the sum of edge weights of any other spanning tree of $\mathcal{G}$).

Informal: lowest-cost tree that connects all of $\mathcal{G}$.



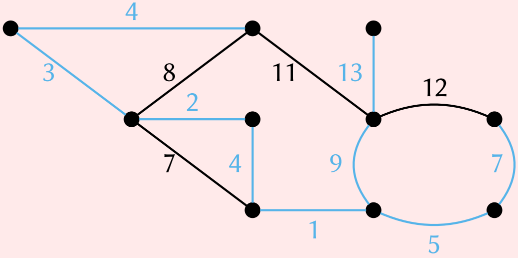
Minimum spanning trees

Definition

Let $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ be a *connected* weighted undirected graph with weight function *weight*.
A *minimum spanning tree* of $\mathcal{G}$ is a subgraph $\mathcal{T} = (\mathcal{N}, \mathcal{E}')$ such that

- $\mathcal{T}$ is a spanning tree;
- the sum of edge weights $\sum_{e \in \mathcal{E}'} \mathit{weight}(e)$ is *minimal*
  (not larger than the sum of edge weights of any other spanning tree of $\mathcal{G}$).

Informal: lowest-cost tree that connects all of $\mathcal{G}$.



Minimal!

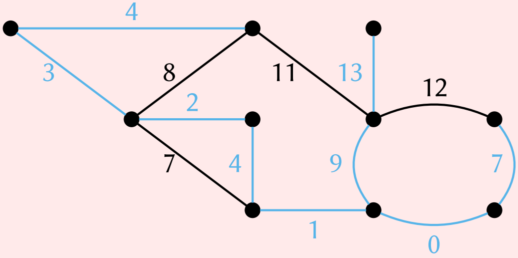
Minimum spanning trees

Definition

Let $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ be a *connected* weighted undirected graph with weight function *weight*.
A *minimum spanning tree* of $\mathcal{G}$ is a subgraph $\mathcal{T} = (\mathcal{N}, \mathcal{E}')$ such that

- $\mathcal{T}$ is a spanning tree;
- the sum of edge weights $\sum_{e \in \mathcal{E}'} \mathit{weight}(e)$ is *minimal*
  (not larger than the sum of edge weights of any other spanning tree of $\mathcal{G}$).

Informal: lowest-cost tree that connects all of $\mathcal{G}$.



Minimal!

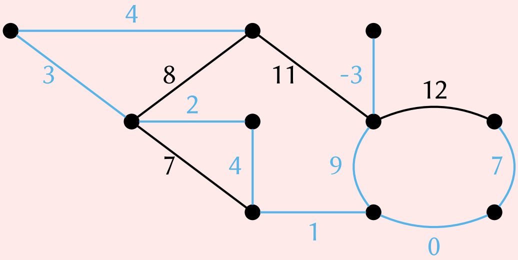
Minimum spanning trees

Definition

Let $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ be a *connected* weighted undirected graph with weight function *weight*.
A *minimum spanning tree* of $\mathcal{G}$ is a subgraph $\mathcal{T} = (\mathcal{N}, \mathcal{E}')$ such that

- $\mathcal{T}$ is a spanning tree;
- the sum of edge weights $\sum_{e \in \mathcal{E}'} \mathit{weight}(e)$ is *minimal*
  (not larger than the sum of edge weights of any other spanning tree of $\mathcal{G}$).

Informal: lowest-cost tree that connects all of $\mathcal{G}$.



Minimal!

A minimum spanning tree algorithm

Algorithm MST-HighLevel($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*):

$E := \emptyset$.
while $(\mathcal{N}, E)$ is not a spanning tree **do**
    Find an edge $(m, n) \in \mathcal{E}$ such that $E \cup \{(m, n)\}$ is
    a subset of the edges of a minimum spanning tree of $\mathcal{G}$.
    $E := E \cup \{(m, n)\}$.
return E.

A minimum spanning tree algorithm

Algorithm MST-HighLevel($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*):

$E := \emptyset$.
while $(\mathcal{N}, E)$ is not a spanning tree **do**
    Find an edge $(m, n) \in \mathcal{E}$ such that $E \cup \{(m, n)\}$ is
    a subset of the edges of a minimum spanning tree of $\mathcal{G}$.
    $E := E \cup \{(m, n)\}$.
return E.

How to find such edges (m, n)?

A minimum spanning tree algorithm

Algorithm MST-HighLevel($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*):

$E := \emptyset$.
while $(\mathcal{N}, E)$ is not a spanning tree **do**
    Find an edge $(m, n) \in \mathcal{E}$ such that $E \cup \{(m, n)\}$ is a subset of the edges of a minimum spanning tree of $\mathcal{G}$.
    $E := E \cup \{(m, n)\}$.
return E.

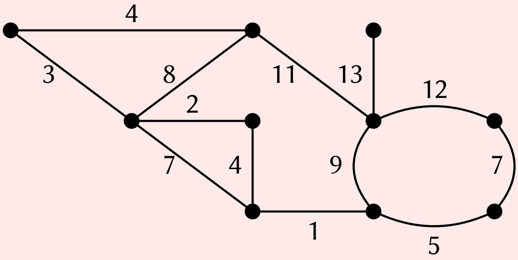
How to find such edges (m, n)?

We need some properties of minimum spanning trees.

A minimum spanning tree algorithm

Definition

Consider an undirected graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$.

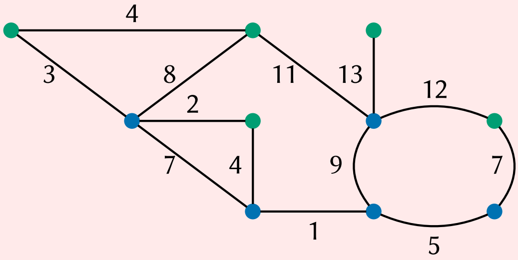


A minimum spanning tree algorithm

Definition

Consider an undirected graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$.

- A *cut* is a partition of $\mathcal{N}$ into two sets S and $\mathcal{N} \setminus S$.

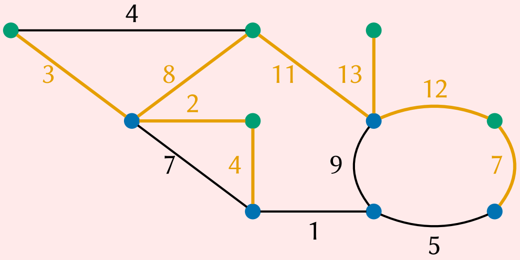


A minimum spanning tree algorithm

Definition

Consider an undirected graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$.

- A *cut* is a partition of $\mathcal{N}$ into two sets S and $\mathcal{N} \setminus S$.
- A *crossing edge* for a cut is an edge that connects a node in S with a node in $\mathcal{N} \setminus S$.

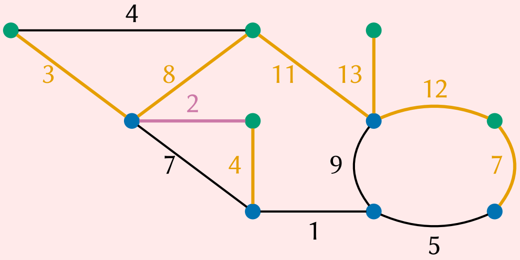


A minimum spanning tree algorithm

Definition

Consider an undirected graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$.

- A *cut* is a partition of $\mathcal{N}$ into two sets S and $\mathcal{N} \setminus S$.
- A *crossing edge* for a cut is an edge that connects a node in S with a node in $\mathcal{N} \setminus S$.
- A *light edge* for a cut is a crossing edge with minimal weight.



A minimum spanning tree algorithm

Definition

Consider an undirected graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$.

- A *cut* is a partition of $\mathcal{N}$ into two sets S and $\mathcal{N} \setminus S$.
- A *crossing edge* for a cut is an edge that connects a node in S with a node in $\mathcal{N} \setminus S$.
- A *light edge* for a cut is a crossing edge with minimal weight.

Theorem

Any minimum spanning tree of $\mathcal{G}$ holds a light edge for any cut $(S, \mathcal{N} \setminus S)$.

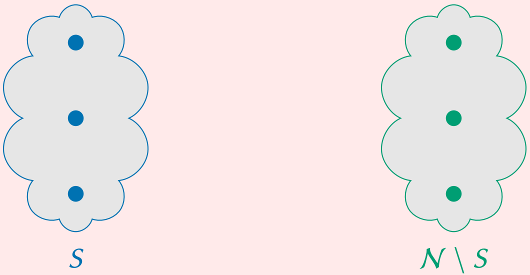
A minimum spanning tree algorithm

Theorem

Any minimum spanning tree of $\mathcal{G}$ holds a light edge for any cut $(S, \mathcal{N} \setminus S)$.

Proof

Assume a minimum spanning tree $\mathcal{T}$ of $\mathcal{G}$ that does *not* hold a light edge for $(S, \mathcal{N} \setminus S)$.



A minimum spanning tree algorithm

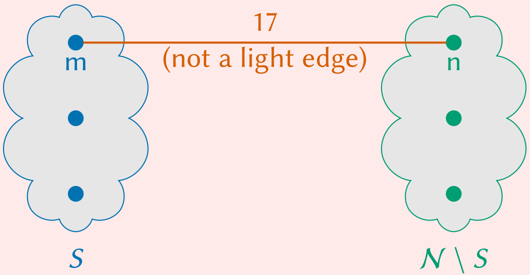
Theorem

Any minimum spanning tree of $\mathcal{G}$ holds a light edge for any cut $(S, \mathcal{N} \setminus S)$.

Proof

Assume a minimum spanning tree $\mathcal{T}$ of $\mathcal{G}$ that does *not* hold a light edge for $(S, \mathcal{N} \setminus S)$.

- $\mathcal{T}$ must have a *non-light edge* connecting a node $m \in S$ with a node $n \in (\mathcal{N} \setminus S)$.



A minimum spanning tree algorithm

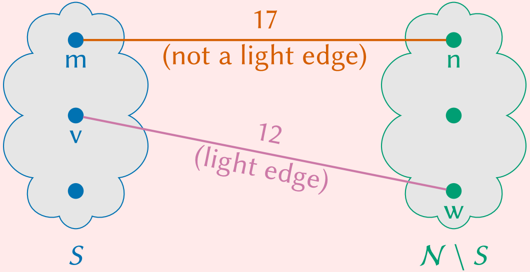
Theorem

Any minimum spanning tree of $\mathcal{G}$ holds a light edge for any cut $(S, \mathcal{N} \setminus S)$.

Proof

Assume a minimum spanning tree $\mathcal{T}$ of $\mathcal{G}$ that does *not* hold a light edge for $(S, \mathcal{N} \setminus S)$.

- $\mathcal{T}$ must have a *non-light edge* connecting a node $m \in S$ with a node $n \in (\mathcal{N} \setminus S)$.
- There must exist a *light edge* (v, w) for cut $(S, \mathcal{N} \setminus S)$ (*not* an edge of $\mathcal{T}$).



A minimum spanning tree algorithm

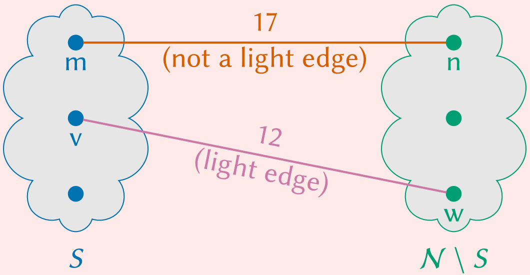
Theorem

Any minimum spanning tree of $\mathcal{G}$ holds a light edge for any cut $(S, \mathcal{N} \setminus S)$.

Proof

Assume a minimum spanning tree $\mathcal{T}$ of $\mathcal{G}$ that does *not* hold a light edge for $(S, \mathcal{N} \setminus S)$.

- Now consider the graph $\mathcal{T}'$ obtained from $\mathcal{T}$ by *removing* (m, n) and *adding* (v, w).



A minimum spanning tree algorithm

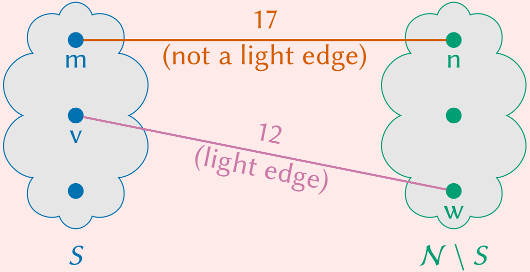
Theorem

Any minimum spanning tree of $\mathcal{G}$ holds a light edge for any cut $(S, \mathcal{N} \setminus S)$.

Proof

Assume a minimum spanning tree $\mathcal{T}$ of $\mathcal{G}$ that does *not* hold a light edge for $(S, \mathcal{N} \setminus S)$.

- Now consider the graph $\mathcal{T}'$ obtained from $\mathcal{T}$ by *removing* (m, n) and *adding* (v, w).
- *Claim*: the sum of edge weights of $\mathcal{T}'$ is *lower* than the sum of edge weights of $\mathcal{T}$.



A minimum spanning tree algorithm

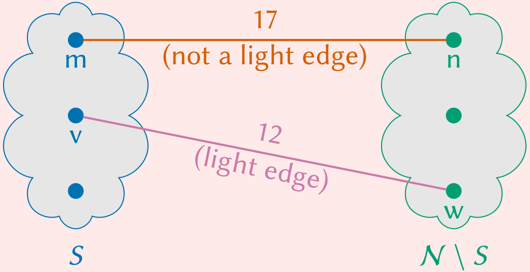
Theorem

Any minimum spanning tree of $\mathcal{G}$ holds a light edge for any cut $(S, \mathcal{N} \setminus S)$.

Proof

Assume a minimum spanning tree $\mathcal{T}$ of $\mathcal{G}$ that does *not* hold a light edge for $(S, \mathcal{N} \setminus S)$.

- Now consider the graph $\mathcal{T}'$ obtained from $\mathcal{T}$ by *removing* (m, n) and *adding* (v, w).
- *Claim*: $\mathcal{T}'$ is *connected* and, hence, a tree.



A minimum spanning tree algorithm

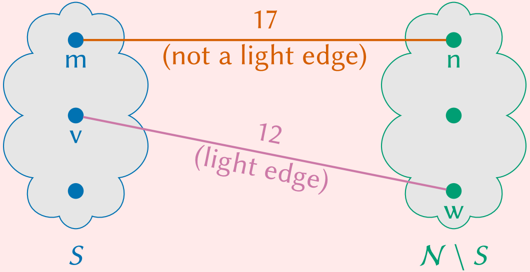
Theorem

Any minimum spanning tree of $\mathcal{G}$ holds a light edge for any cut $(S, \mathcal{N} \setminus S)$.

Proof

Assume a minimum spanning tree $\mathcal{T}$ of $\mathcal{G}$ that does *not* hold a light edge for $(S, \mathcal{N} \setminus S)$.

- Now consider the graph $\mathcal{T}'$ obtained from $\mathcal{T}$ by *removing* (m, n) and *adding* (v, w).
- *Contradiction* $\mathcal{T}$ cannot be a minimum spanning tree!



A minimum spanning tree algorithm

Algorithm MST-HighLevel($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*):

```
E := ∅.
while (N, E) is not a spanning tree do
    Find an edge (m, n) ∈ ℰ such that E ∪ {(m, n)} is
    a subset of the edges of a minimum spanning tree of 𝒢.
    E := E ∪ {(m, n)}.
return E.
```

How to find such edges (m, n)?

A minimum spanning tree algorithm

Algorithm MST-HighLevel($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*):

$E := \emptyset$.
while $(\mathcal{N}, E)$ is not a spanning tree **do**
    Find an edge $(m, n) \in \mathcal{E}$ such that $E \cup \{(m, n)\}$ is
    a subset of the edges of a minimum spanning tree of $\mathcal{G}$.
    $E := E \cup \{(m, n)\}$.
return E.

How to find such edges (m, n)?

Consider a cut $(S, S \setminus \mathcal{N})$ such that no edge in E is a crossing edge.

We can pick any light edge for cut $(S, S \setminus \mathcal{N})$.

Kruskal's Algorithm

Algorithm MST-HighLevel($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*):

$E := \emptyset$.
while $(\mathcal{N}, E)$ is not a spanning tree **do**
    Find an edge $(m, n) \in \mathcal{E}$ such that $E \cup \{(m, n)\}$ is a subset of the edges of a minimum spanning tree of $\mathcal{G}$.
    $E := E \cup \{(m, n)\}$.
return E.

Algorithm MST-Kruskal($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*):

$E := \emptyset$.
while $|E| \neq |\mathcal{N}| - 1$ **do**
    Find an edge $(m, n) \in \mathcal{E}$ with minimum edge weight such that m and n are not yet connected in $(\mathcal{N}, E)$.
    $E := E \cup \{(m, n)\}$.
return E.

Kruskal's Algorithm

Algorithm MST-Kruskal($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*):

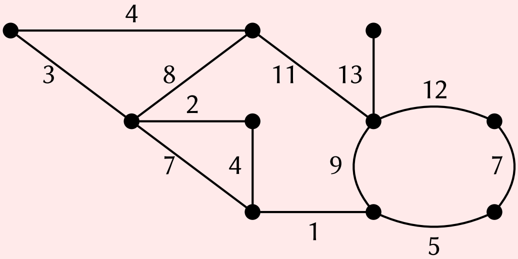
$E := \emptyset$.

while $|E| \neq |\mathcal{N}| - 1$ **do**

  Find an edge $(m, n) \in \mathcal{E}$ with minimum edge weight such that m and n are not yet connected in $(\mathcal{N}, E)$.

  $E := E \cup \{(m, n)\}$.

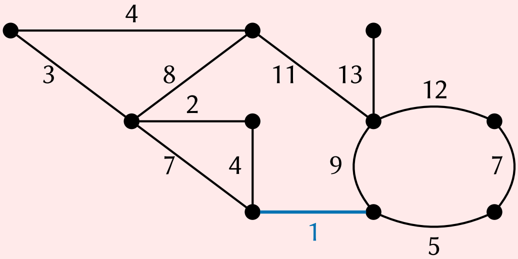
return E.



Kruskal's Algorithm

Algorithm MST-KRUSKAL($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*):

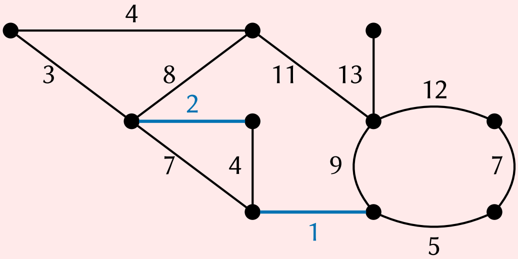
1: $E := \emptyset$.
2: **while** $|E| \neq |\mathcal{N}| - 1$ **do**
3: Find an edge $(m, n) \in \mathcal{E}$ with minimum edge weight such that m and n are not yet connected in $(\mathcal{N}, E)$.
4: $E := E \cup \{(m, n)\}$.
5: **return** E.



Kruskal's Algorithm

Algorithm MST-KRUSKAL($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*):

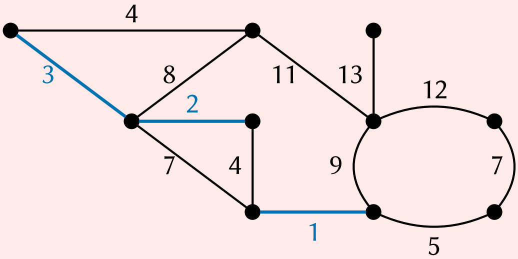
$E := \emptyset$.
while $|E| \neq |\mathcal{N}| - 1$ **do**
    Find an edge $(m, n) \in \mathcal{E}$ with minimum edge weight such that m and n are not yet connected in $(\mathcal{N}, E)$.
    $E := E \cup \{(m, n)\}$.
return E.



Kruskal's Algorithm

Algorithm MST-KRUSKAL($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*):

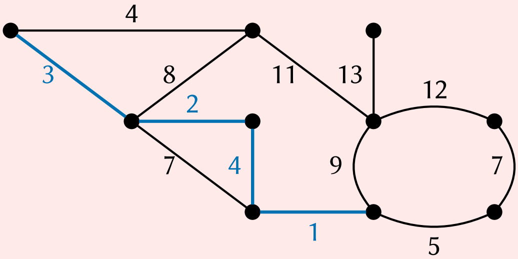
$E := \emptyset$.
while $|E| \neq |\mathcal{N}| - 1$ **do**
    Find an edge $(m, n) \in \mathcal{E}$ with minimum edge weight such that m and n are not yet connected in $(\mathcal{N}, E)$.
    $E := E \cup \{(m, n)\}$.
return E.



Kruskal's Algorithm

Algorithm MST-KRUSKAL($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*):

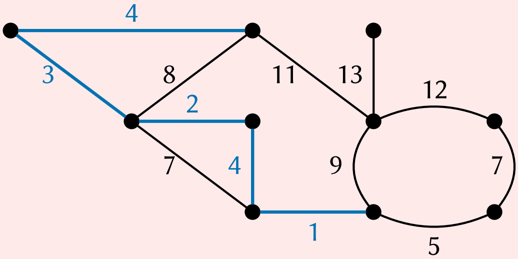
$E := \emptyset$.
while $|E| \neq |\mathcal{N}| - 1$ **do**
  Find an edge $(m, n) \in \mathcal{E}$ with minimum edge weight such that m and n are not yet connected in $(\mathcal{N}, E)$.
  $E := E \cup \{(m, n)\}$.
return E.



Kruskal's Algorithm

Algorithm MST-KRUSKAL($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*):

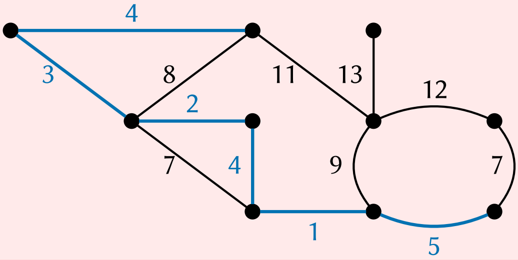
$E := \emptyset$.
while $|E| \neq |\mathcal{N}| - 1$ **do**
    Find an edge $(m, n) \in \mathcal{E}$ with minimum edge weight such that m and n are not yet connected in $(\mathcal{N}, E)$.
    $E := E \cup \{(m, n)\}$.
return E.



Kruskal's Algorithm

Algorithm $\textsc{MST-Kruskal}(\mathcal{G} = (\mathcal{N}, \mathcal{E}), \mathit{weight})$**:**

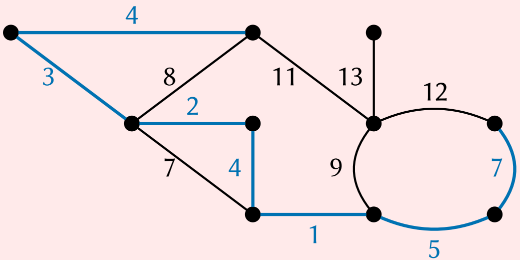
```
E := ∅.
while |E| ≠ |N| − 1 do
    Find an edge (m, n) ∈ ℰ with minimum edge weight such that
    m and n are not yet connected in (N, E).
    E := E ∪ {(m, n)}.
return E.
```



Kruskal's Algorithm

Algorithm MST-KRUSKAL($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*):

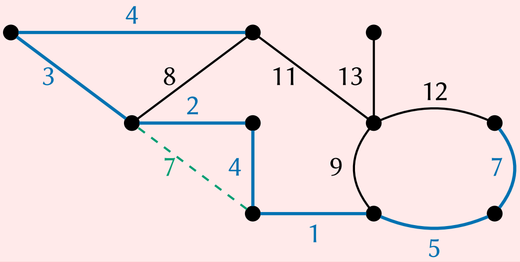
$E := \emptyset$.
while $|E| \neq |\mathcal{N}| - 1$ **do**
    Find an edge $(m, n) \in \mathcal{E}$ with minimum edge weight such that m and n are not yet connected in $(\mathcal{N}, E)$.
    $E := E \cup \{(m, n)\}$.
return E.



Kruskal's Algorithm

Algorithm MST-Kruskal($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*):

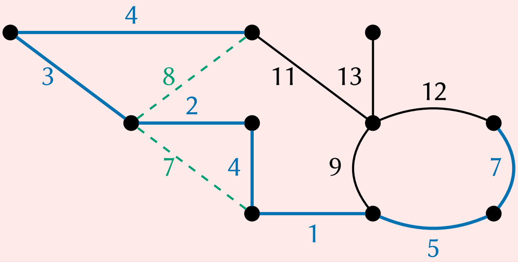
$E := \emptyset$.
while $|E| \neq |\mathcal{N}| - 1$ **do**
  Find an edge $(m, n) \in \mathcal{E}$ with minimum edge weight such that m and n are not yet connected in $(\mathcal{N}, E)$.
  $E := E \cup \{(m, n)\}$.
return E.



Kruskal's Algorithm

Algorithm MST-KRUSKAL($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*):

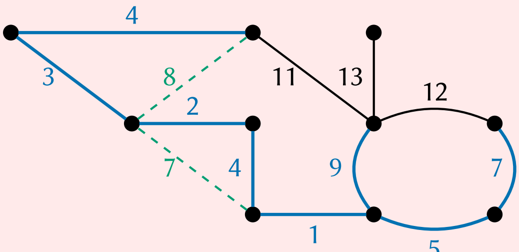
$E := \emptyset$.
while $|E| \neq |\mathcal{N}| - 1$ **do**
    Find an edge $(m, n) \in \mathcal{E}$ with minimum edge weight such that m and n are not yet connected in $(\mathcal{N}, E)$.
    $E := E \cup \{(m, n)\}$.
return E.



Kruskal's Algorithm

Algorithm MST-KRUSKAL($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*)**:**

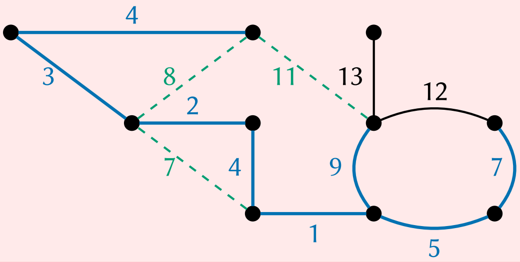
$E := \emptyset$.
while $|E| \neq |\mathcal{N}| - 1$ **do**
    Find an edge $(m, n) \in \mathcal{E}$ with minimum edge weight such that m and n are not yet connected in $(\mathcal{N}, E)$.
    $E := E \cup \{(m, n)\}$.
return E.



Kruskal's Algorithm

Algorithm MST-KRUSKAL($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*):

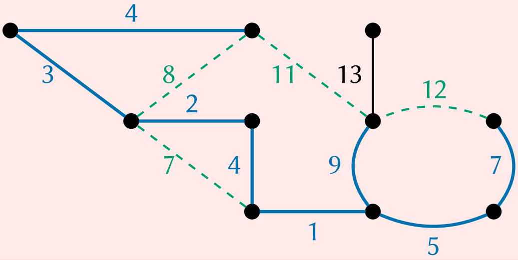
1: $E := \emptyset$.
2: **while** $|E| \neq |\mathcal{N}| - 1$ **do**
3: Find an edge $(m, n) \in \mathcal{E}$ with minimum edge weight such that m and n are not yet connected in $(\mathcal{N}, E)$.
4: $E := E \cup \{(m, n)\}$.
5: **return** E.



Kruskal's Algorithm

Algorithm MST-KRUSKAL($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*):

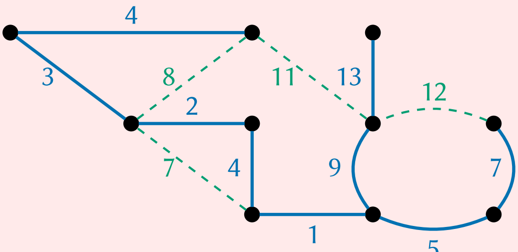
1: $E := \emptyset$.
2: **while** $|E| \neq |\mathcal{N}| - 1$ **do**
3: Find an edge $(m, n) \in \mathcal{E}$ with minimum edge weight such that m and n are not yet connected in $(\mathcal{N}, E)$.
4: $E := E \cup \{(m, n)\}$.
5: **return** E.



Kruskal's Algorithm

Algorithm MST-KRUSKAL($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*):

$E := \emptyset$.
while $|E| \neq |\mathcal{N}| - 1$ **do**
    Find an edge $(m, n) \in \mathcal{E}$ with minimum edge weight such that m and n are not yet connected in $(\mathcal{N}, E)$.
    $E := E \cup \{(m, n)\}$.
return E.



Kruskal's Algorithm

Algorithm MST-KRUSKAL($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*):

```
E := ∅.
while |E| ≠ |N| − 1 do
    Find an edge (m, n) ∈ ℰ with minimum edge weight such that
    m and n are not yet connected in (N, E).
    E := E ∪ {(m, n)}.
return E.
```

How to find edges $(m, n) \in \mathcal{E}$?

Kruskal's Algorithm

Algorithm MST-KRUSKAL($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*):

$E := \emptyset$.
while $|E| \neq |\mathcal{N}| - 1$ **do**
    Find an edge $(m, n) \in \mathcal{E}$ with minimum edge weight such that m and n are not yet connected in $(\mathcal{N}, E)$.
    $E := E \cup \{(m, n)\}$.
return E.

How to find edges $(m, n) \in \mathcal{E}$?

- Sort all edges on increasing edge weight.

Kruskal's Algorithm

Algorithm MST-Kruskal($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*):

```
E := ∅.
while |E| ≠ |N| − 1 do
    Find an edge (m, n) ∈ ℰ with minimum edge weight such that
    m and n are not yet connected in (N, E).
    E := E ∪ {(m, n)}.
return E.
```

How to find edges $(m, n) \in \mathcal{E}$?

- Sort all edges on increasing edge weight.
- Maintain a *dynamic connectivity* data structure D that represents the connected components in $(\mathcal{N}, E)$.

Kruskal's Algorithm

Algorithm MST-KRUSKAL($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*):

```
E := ∅.
while |E| ≠ |N| − 1 do
    Find an edge (m, n) ∈ ℰ with minimum edge weight such that
    m and n are not yet connected in (N, E).
    E := E ∪ {(m, n)}.
return E.
```

How to find edges $(m, n) \in \mathcal{E}$?

- Sort all edges on increasing edge weight.
- Maintain a *dynamic connectivity* data structure D that represents the connected components in $(\mathcal{N}, E)$.
- For each edge (m, n) in sorted order: check whether they are connected via D.

Kruskal's Algorithm

Algorithm MST-KRUSKAL($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*):

```
E := ∅.
while |E| ≠ |N| − 1 do
    Find an edge (m, n) ∈ ℰ with minimum edge weight such that
    m and n are not yet connected in (N, E).
    E := E ∪ {(m, n)}.
return E.
```

How to find edges $(m, n) \in \mathcal{E}$?

- Sort all edges on increasing edge weight.
- Maintain a *dynamic connectivity* data structure D that represents the connected components in $(\mathcal{N}, E)$.
- For each edge (m, n) in sorted order: check whether they are connected via D.

Complexity.

Kruskal's Algorithm

Algorithm MST-KRUSKAL($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*):

$E := \emptyset$.
while $|E| \neq |\mathcal{N}| - 1$ **do**
    Find an edge $(m, n) \in \mathcal{E}$ with minimum edge weight such that m and n are not yet connected in $(\mathcal{N}, E)$.
    $E := E \cup \{(m, n)\}$.
return E.

How to find edges $(m, n) \in \mathcal{E}$?

- Sort all edges on increasing edge weight.
- Maintain a *dynamic connectivity* data structure D that represents the connected components in $(\mathcal{N}, E)$.
- For each edge (m, n) in sorted order: check whether they are connected via D.

Complexity. We need a *dynamic connectivity* data structure!

Kruskal's Algorithm

Algorithm MST-Kruskal($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*):

```
E := ∅.
while |E| ≠ |N| − 1 do
    Find an edge (m, n) ∈ ℰ with minimum edge weight such that
    m and n are not yet connected in (N, E).
    E := E ∪ {(m, n)}.
return E.
```

How to find edges $(m, n) \in \mathcal{E}$?

- Sort all edges on increasing edge weight. $\rightarrow \Theta\left(|\mathcal{E}| \log(|\mathcal{E}|)\right)$.
- Maintain a *dynamic connectivity* data structure D that represents the connected components in $(\mathcal{N}, E)$. $\rightarrow$?
- For each edge (m, n) in sorted order: check whether they are connected via D. $\rightarrow$?

Complexity. We need a *dynamic connectivity* data structure!

Dynamic connectivity

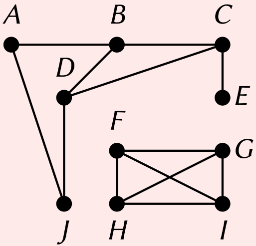
Definition

Given a list of pairs (p, q) that imply that p and q are *connected*, we can classify values based on whether they are connected with each other (possibly via other values).

Dynamic connectivity

Definition

Given a list of pairs (p, q) that imply that p and q are *connected*, we can classify values based on whether they are connected with each other (possibly via other values).



(A, B) (B, C) (C, D)

(C, E) (D, B) (F, G)

(F, H) (G, H) (G, I)

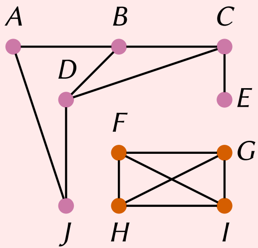
(H, I) (I, F) (J, A) (J, D)

Values can represents *computers* connected via *networks*.

Dynamic connectivity

Definition

Given a list of pairs (p, q) that imply that p and q are *connected*, we can classify values based on whether they are connected with each other (possibly via other values).



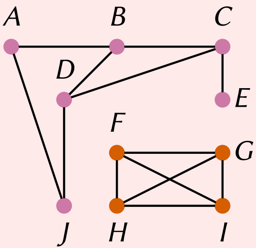
(A, B) (B, C) (C, D)
(C, E) (D, B) (F, G)
(F, H) (G, H) (G, I)
(H, I) (I, F) (J, A) (J, D)

Values can represents *computers* connected via *networks*.

Dynamic connectivity

Definition

Given a list of pairs (p, q) that imply that p and q are *connected*, we can classify values based on whether they are connected with each other (possibly via other values).



(A, B) (B, C) (C, D)
(C, E) (D, B) (F, G)
(F, H) (G, H) (G, I)
(H, I) (I, F) (J, A) (J, D)

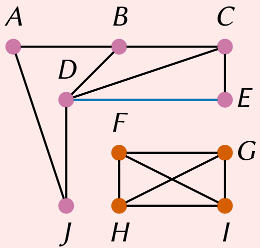
Dynamic connectivity problem

Given a list of connections L and a new connection (p, q),
determine whether adding (p, q) to L changes the classifications.

Dynamic connectivity

Definition

Given a list of pairs (p, q) that imply that p and q are *connected*, we can classify values based on whether they are connected with each other (possibly via other values).



(A, B)	(B, C)	(C, D)	
(C, E)	(D, B)	(F, G)	
(F, H)	(G, H)	(G, I)	
(H, I)	(I, F)	(J, A)	(J, D)

Adding (D, E).

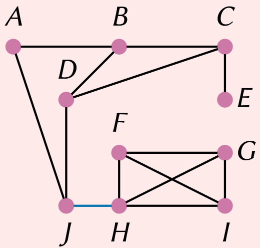
Dynamic connectivity problem

Given a list of connections L and a new connection (p, q),
determine whether adding (p, q) to L changes the classifications.

Dynamic connectivity

Definition

Given a list of pairs (p, q) that imply that p and q are *connected*, we can classify values based on whether they are connected with each other (possibly via other values).



(A, B) (B, C) (C, D)
(C, E) (D, B) (F, G)
(F, H) (G, H) (G, I)
(H, I) (I, F) (J, A) (J, D)

Adding (J, H).

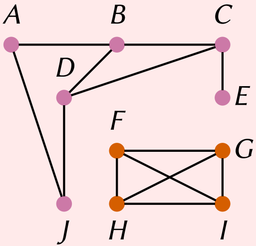
Dynamic connectivity problem

Given a list of connections L and a new connection (p, q),
determine whether adding (p, q) to L changes the classifications.

Dynamic connectivity

Definition

Given a list of pairs (p, q) that imply that p and q are *connected*, we can classify values based on whether they are connected with each other (possibly via other values).



(A, B)	(B, C)	(C, D)	
(C, E)	(D, B)	(F, G)	
(F, H)	(G, H)	(G, I)	
(H, I)	(I, F)	(J, A)	(J, D)

Process a list of connections L

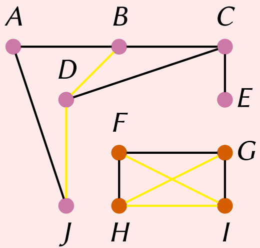
Start with an empty result R.

Add pair $(p, q) \in L$ to R if that *changes the classifications* in R.

Dynamic connectivity

Definition

Given a list of pairs (p, q) that imply that p and q are *connected*, we can classify values based on whether they are connected with each other (possibly via other values).



(A, B)	(B, C)	(C, D)	
(C, E)	(D, B)	(F, G)	
(F, H)	(G, H)	(G, I)	
(H, I)	(I, F)	(J, A)	(J, D)

Process a list of connections L

Start with an empty result R.

Add pair $(p, q) \in L$ to R if that *changes the classifications* in R.

Dynamic connectivity: A first attempt

We need both data structures and algorithms:

data structures to represent the classification we have;

algorithms to *check* whether adding a connection changes the classification;

algorithms to *update* the classification by adding a connection.

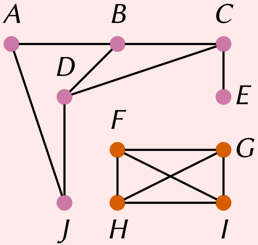
Dynamic connectivity: A first attempt

We need both data structures and algorithms:

data structures to represent the classification we have;

algorithms to *check* whether adding a connection changes the classification;

algorithms to *update* the classification by adding a connection.



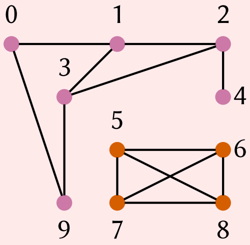
Dynamic connectivity: A first attempt

We need both data structures and algorithms:

data structures to represent the classification we have;

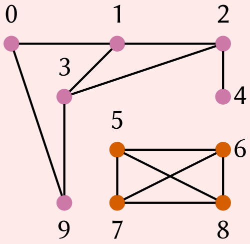
algorithms to *check* whether adding a connection changes the classification;

algorithms to *update* the classification by adding a connection.



Simplification: assume N values, each a unique integer in the range $0, \ldots, N-1$.

Dynamic connectivity: A first attempt



A:

V	V	V	V	V	W	W	W	W	V

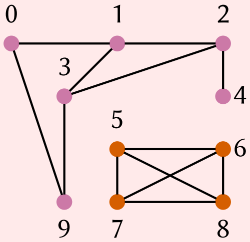
Simplification: assume N values, each a unique integer in the range $0, \dots, N-1$.

The simple representation

Create an array $A[0 \dots N)$ such that $A[i]$ is the class identifier of value i.

We can efficiently check in $\Theta(1)$ whether pair (p, q) are already connected: $S[p] = S[q]$.

Dynamic connectivity: A first attempt



A:									
V	V	V	V	V	W	W	W	W	V

Simplification: assume N values, each a unique integer in the range $0, \ldots, N-1$.

The simple representation

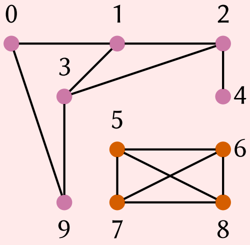
Create an array $A[0 \ldots N)$ such that $A[i]$ is the class identifier of value i.

We can efficiently check in $\Theta(1)$ whether pair (p, q) are already connected: $S[p] = S[q]$.

Updating the simple representation

Given a new (p, q), change all entries in A with value $A[p]$ to value $A[q]$ in $\Theta(N)$.

Dynamic connectivity: A first attempt



Adding (J, H).

A:									
V	V	V	V	V	W	W	W	W	V

Simplification: assume N values, each a unique integer in the range $0, \ldots, N-1$.

The simple representation

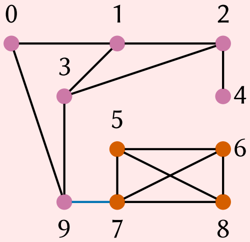
Create an array $A[0 \ldots N)$ such that $A[i]$ is the class identifier of value i.

We can efficiently check in $\Theta(1)$ whether pair (p, q) are already connected: $S[p] = S[q]$.

Updating the simple representation

Given a new (p, q), change all entries in A with value $A[p]$ to value $A[q]$ in $\Theta(N)$.

Dynamic connectivity: A first attempt



Adding $(9, 7)$.

A:									
V	V	V	V	V	W	W	W	W	V

Simplification: assume N values, each a unique integer in the range $0, \ldots, N-1$.

The simple representation

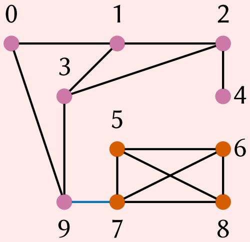
Create an array $A[0 \ldots N)$ such that $A[i]$ is the class identifier of value i.

We can efficiently check in $\Theta(1)$ whether pair (p, q) are already connected: $S[p] = S[q]$.

Updating the simple representation

Given a new (p, q), change all entries in A with value $A[p]$ to value $A[q]$ in $\Theta(N)$.

Dynamic connectivity: A first attempt



Adding (9, 7).

Simplification: assume N values, each a unique integer in the range $0, \dots, N-1$.

The simple representation

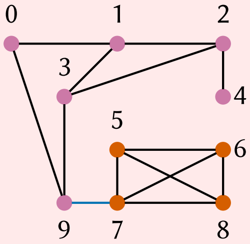
Create an array $A[0 \dots N)$ such that $A[i]$ is the class identifier of value i.

We can efficiently check in $\Theta(1)$ whether pair (p, q) are already connected: $S[p] = S[q]$.

Updating the simple representation

Given a new (p, q), change all entries in A with value $A[p]$ to value $A[q]$ in $\Theta(N)$.

Dynamic connectivity: A first attempt



Adding $(9, 7)$.

A:

W	W	W	W	W	W	W	W	W	W

Simplification: assume N values, each a unique integer in the range $0, \ldots, N - 1$.

The simple representation

Create an array $A[0 \ldots N)$ such that $A[i]$ is the class identifier of value i.

We can efficiently check in $\Theta(1)$ whether pair (p, q) are already connected: $S[p] = S[q]$.

Updating the simple representation

Given a new (p, q), change all entries in A with value $A[p]$ to value $A[q]$ in $\Theta(N)$.

Dynamic connectivity: A first attempt

We need both data structures and algorithms:

data structures to represent the classification we have;

algorithms to *check* whether adding a connection changes the classification;

algorithms to *update* the classification by adding a connection.

The simple representation

Create an array $A[0 \dots N)$ such that $A[i]$ is the class identifier of value i.

Check whether adding a connection changes the classification: $\Theta(1)$;

Update the classification by adding a connection: $\Theta(N)$.

Dynamic connectivity: A first attempt

We need both data structures and algorithms:

data structures to represent the classification we have;
algorithms to *check* whether adding a connection changes the classification;
algorithms to *update* the classification by adding a connection.

The simple representation

Create an array $A[0 \dots N)$ such that $A[i]$ is the class identifier of value i.

Check whether adding a connection changes the classification: $\Theta(1)$;

Update the classification by adding a connection: $\Theta(N)$.

Process a list of connections L: at-least N^2 if all values end up in the same class!

Dynamic connectivity: A first attempt

We need both data structures and algorithms:

data structures to represent the classification we have;
algorithms to *check* whether adding a connection changes the classification;
algorithms to *update* the classification by adding a connection.

The simple representation

Create an array $A[0\ldots N)$ such that $A[i]$ is the class identifier of value i.

Check whether adding a connection changes the classification: $\Theta(1)$;

Update the classification by adding a connection: $\Theta(N)$.

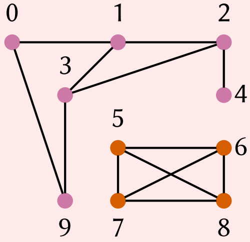
Process a list of connections L: at-least N^2 if all values end up in the same class!

Faster *minimization*?

The *simple representation* is optimized for checking classifications, not updating them.
We need another representation!

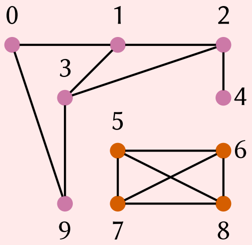
Dynamic connectivity: A second attempt

We want to optimize for *updating*: no updating of an entire array.



Dynamic connectivity: A second attempt

We want to optimize for *updating*: no updating of an entire array.



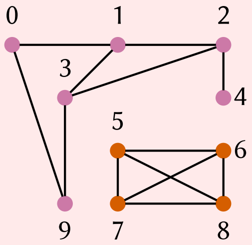
The forest representation

Create an array $S[0 \ldots N)$ such that $S[i]$ either

- holds the value i indicating that i is the *tree root* for the class containing i;
- holds the value $j \neq i$ indicating that j is a *tree parent* for the class containing i.

Dynamic connectivity: A second attempt

We want to optimize for *updating*: no updating of an entire array.



F:

3	2	9	4	4	6	7	7	6	4

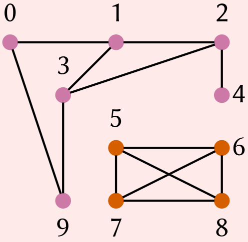
The forest representation

Create an array $S[0\ldots N)$ such that $S[i]$ either

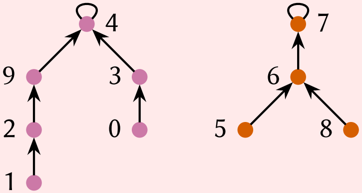
- holds the value i indicating that i is the *tree root* for the class containing i;
- holds the value $j \neq i$ indicating that j is a *tree parent* for the class containing i.

Dynamic connectivity: A second attempt

We want to optimize for *updating*: no updating of an entire array.



Representation visualization



F:										
	3	2	9	4	4	6	7	7	6	4

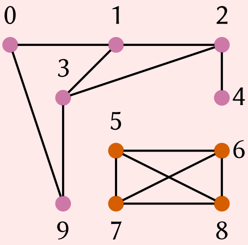
The forest representation

Create an array $S[0 \dots N)$ such that $S[i]$ either

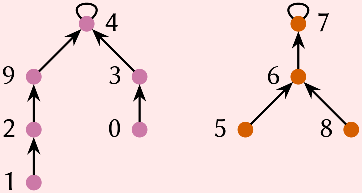
- holds the value i indicating that i is the *tree root* for the class containing i;
- holds the value $j \neq i$ indicating that j is a *tree parent* for the class containing i.

Dynamic connectivity: A second attempt

We want to optimize for *updating*: no updating of an entire array.



Representation visualization



F:

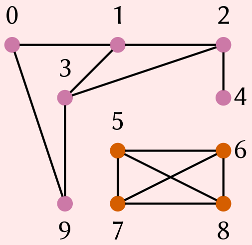
3	2	9	4	4	6	7	7	6	4

The forest representation

Check whether adding a connection (p, q) changes the classification: compare the *roots for the trees* holding p and q.

Dynamic connectivity: A second attempt

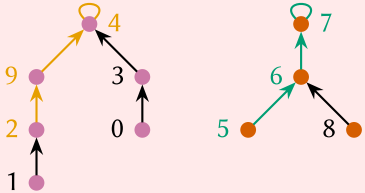
We want to optimize for *updating*: no updating of an entire array.



F:

3	2	9	4	4	6	7	7	6	4

Representation visualization



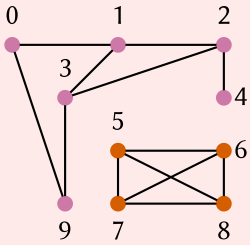
Check (5, 2).

The forest representation

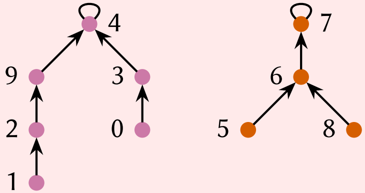
Check whether adding a connection (p, q) changes the classification: compare the *roots for the trees* holding p and q.

Dynamic connectivity: A second attempt

We want to optimize for *updating*: no updating of an entire array.



Representation visualization



F:

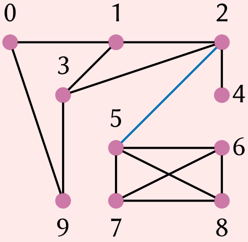
3	2	9	4	4	6	7	7	6	4

The forest representation

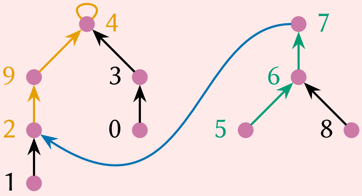
Update the classification by adding a connection (p, q):
change the *root of the tree* holding q so that it points to p.

Dynamic connectivity: A second attempt

We want to optimize for *updating*: no updating of an entire array.



Representation visualization



F:

3	2	9	4	4	6	7	2	6	4

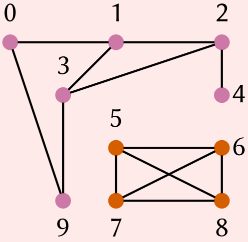
Adding $(5, 2)$.

The forest representation

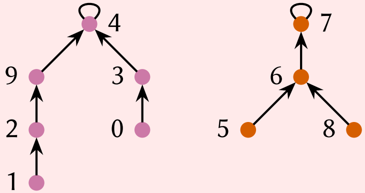
Update the classification by adding a connection (p, q):
change the *root of the tree* holding q so that it points to p.

Dynamic connectivity: A second attempt

We want to optimize for *updating*: no updating of an entire array.



Representation visualization



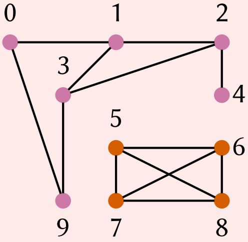
F:									
3	2	9	4	4	6	7	2	6	4

The forest representation

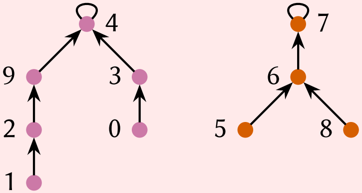
We need to find roots *fast*: costs of checking and updating depends on it!

Dynamic connectivity: A second attempt

We want to optimize for *updating*: no updating of an entire array.



Representation visualization



F:										
	3	2	9	4	4	6	7	2	6	4

The forest representation

We need to find roots *fast*: costs of checking and updating depends on it!

Finding the root of p: depends on the *distance toward the root* (the depth of p): worst-case $\Theta(N)$.

Dynamic connectivity: A second attempt

The forest representation

We need to find roots *fast*: costs of checking and updating depends on it!

Finding the root of p: depends on the *distance toward the root* (the depth of p): worst-case $\Theta(N)$.

Problem: Can we guarantee *low distances* to roots?

Idea: Keep the *tree height* low when adding a connection (p, q).

Dynamic connectivity: A second attempt

The forest representation

We need to find roots *fast*: costs of checking and updating depends on it!

Finding the root of p: depends on the *distance toward the root* (the depth of p): worst-case $\Theta(N)$.

Problem: Can we guarantee *low distances* to roots?

Idea: Keep the *tree height* low when adding a connection (p, q):

- find the roots of the trees holding p and q;
- make the tree root of the *smallest* tree point to the other root.

Dynamic connectivity: A second attempt

The forest representation

We need to find roots *fast*: costs of checking and updating depends on it!

Finding the root of p: depends on the *distance toward the root* (the depth of p): worst-case $\Theta(N)$.

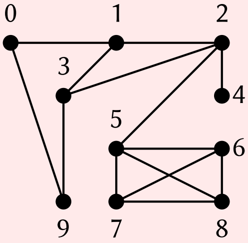
Problem: Can we guarantee *low distances* to roots?

Idea: Keep the *tree height* low when adding a connection (p, q):

- find the roots of the trees holding p and q;
- make the tree root of the *smallest* tree point to the other root.

We need to maintain tree size for roots: an extra array.

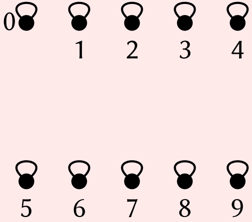
Dynamic connectivity: A second attempt



F:	0	1	2	3	4	5	6	7	8	9

S:	1	1	1	1	1	1	1	1	1	1

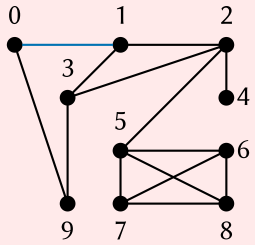
Representation visualization



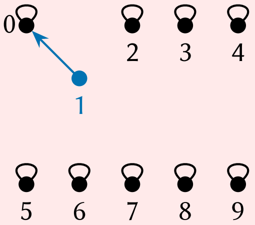
Problem: Can we guarantee *low distances* to roots?

Consider Adding (0, 1), (1, 2), (2, 3), (2, 4), (3, 1), (5, 6), (5, 7), (6, 7), (6, 8), (7, 8), (8, 5), (9, 0), (9, 3), (5, 2).

Dynamic connectivity: A second attempt



Representation visualization



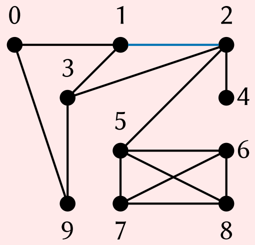
F:									
0	0	2	3	4	5	6	7	8	9

S:									
2	1	1	1	1	1	1	1	1	1

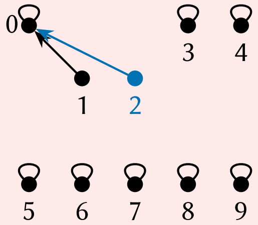
Problem: Can we guarantee *low distances* to roots?

Consider Adding $(0, 1)$, $(1, 2)$, $(2, 3)$, $(2, 4)$, $(3, 1)$, $(5, 6)$, $(5, 7)$, $(6, 7)$, $(6, 8)$, $(7, 8)$, $(8, 5)$, $(9, 0)$, $(9, 3)$, $(5, 2)$.

Dynamic connectivity: A second attempt



Representation visualization

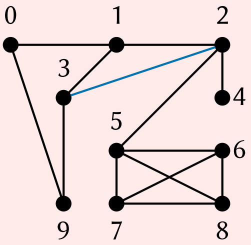


F:	0	0	0	3	4	5	6	7	8	9
S:	3	1	1	1	1	1	1	1	1	1

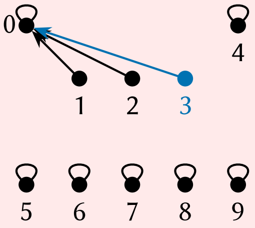
Problem: Can we guarantee *low distances* to roots?

Consider Adding $(0, 1)$, $(1, 2)$, $(2, 3)$, $(2, 4)$, $(3, 1)$, $(5, 6)$, $(5, 7)$, $(6, 7)$,
$(6, 8)$, $(7, 8)$, $(8, 5)$, $(9, 0)$, $(9, 3)$, $(5, 2)$.

Dynamic connectivity: A second attempt



Representation visualization



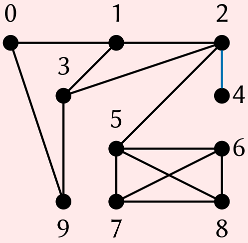
F:	0	0	0	0	4	5	6	7	8	9

S:	4	1	1	1	1	1	1	1	1	1

Problem: Can we guarantee *low distances* to roots?

Consider Adding $(0, 1)$, $(1, 2)$, $(2, 3)$, $(2, 4)$, $(3, 1)$, $(5, 6)$, $(5, 7)$, $(6, 7)$, $(6, 8)$, $(7, 8)$, $(8, 5)$, $(9, 0)$, $(9, 3)$, $(5, 2)$.

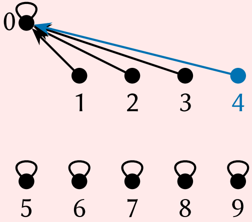
Dynamic connectivity: A second attempt



F:	0	0	0	0	0	5	6	7	8	9

S:	5	1	1	1	1	1	1	1	1	1

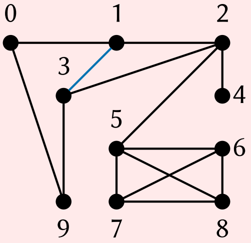
Representation visualization



Problem: Can we guarantee *low distances* to roots?

Consider Adding $(0,1)$, $(1,2)$, $(2,3)$, $(2,4)$, $(3,1)$, $(5,6)$, $(5,7)$, $(6,7)$, $(6,8)$, $(7,8)$, $(8,5)$, $(9,0)$, $(9,3)$, $(5,2)$.

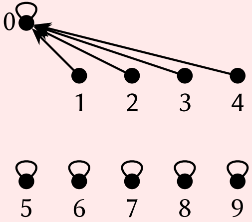
Dynamic connectivity: A second attempt



F:	0	0	0	0	0	5	6	7	8	9

S:	5	1	1	1	1	1	1	1	1	1

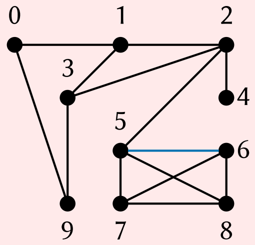
Representation visualization



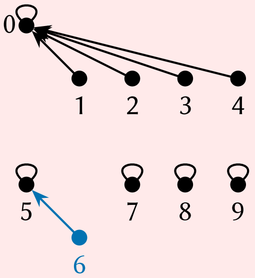
Problem: Can we guarantee *low distances* to roots?

Consider Adding $(0, 1)$, $(1, 2)$, $(2, 3)$, $(2, 4)$, $(3, 1)$, $(5, 6)$, $(5, 7)$, $(6, 7)$, $(6, 8)$, $(7, 8)$, $(8, 5)$, $(9, 0)$, $(9, 3)$, $(5, 2)$.

Dynamic connectivity: A second attempt



Representation visualization

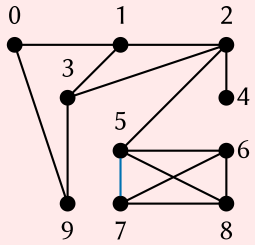


F:	0	0	0	0	0	5	5	7	8	9
S:	5	1	1	1	1	2	1	1	1	1

Problem: Can we guarantee *low distances* to roots?

Consider Adding $(0,1)$, $(1,2)$, $(2,3)$, $(2,4)$, $(3,1)$, $(5,6)$, $(5,7)$, $(6,7)$, $(6,8)$, $(7,8)$, $(8,5)$, $(9,0)$, $(9,3)$, $(5,2)$.

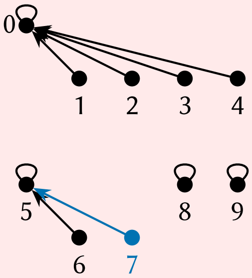
Dynamic connectivity: A second attempt



F:	0	0	0	0	0	5	5	5	8	9

S:	5	1	1	1	1	3	1	1	1	1

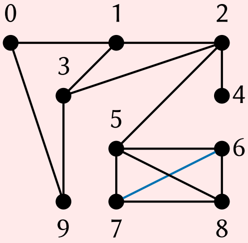
Representation visualization



Problem: Can we guarantee *low distances* to roots?

Consider Adding $(0,1)$, $(1,2)$, $(2,3)$, $(2,4)$, $(3,1)$, $(5,6)$, $(5,7)$, $(6,7)$, $(6,8)$, $(7,8)$, $(8,5)$, $(9,0)$, $(9,3)$, $(5,2)$.

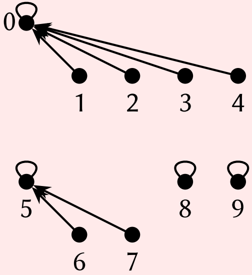
Dynamic connectivity: A second attempt



F:	0	0	0	0	0	5	5	5	8	9

S:	5	1	1	1	1	3	1	1	1	1

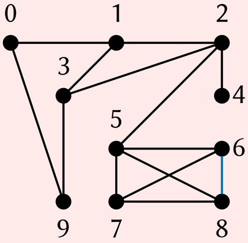
Representation visualization



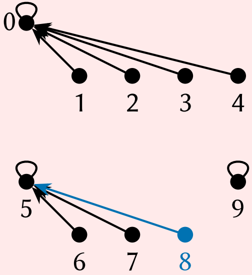
Problem: Can we guarantee *low distances* to roots?

Consider Adding $(0,1)$, $(1,2)$, $(2,3)$, $(2,4)$, $(3,1)$, $(5,6)$, $(5,7)$, $(6,7)$,
$(6,8)$, $(7,8)$, $(8,5)$, $(9,0)$, $(9,3)$, $(5,2)$.

Dynamic connectivity: A second attempt



Representation visualization



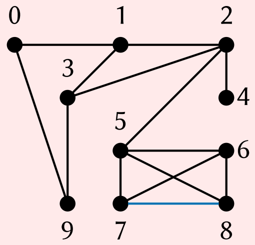
F:	0	0	0	0	0	5	5	5	5	9

S:	5	1	1	1	1	4	1	1	1	1

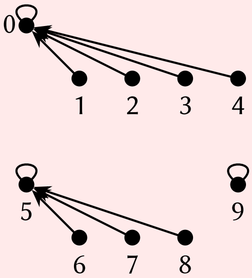
Problem: Can we guarantee *low distances* to roots?

Consider Adding $(0,1)$, $(1,2)$, $(2,3)$, $(2,4)$, $(3,1)$, $(5,6)$, $(5,7)$, $(6,7)$, $(6,8)$, $(7,8)$, $(8,5)$, $(9,0)$, $(9,3)$, $(5,2)$.

Dynamic connectivity: A second attempt



Representation visualization

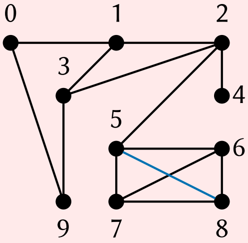


F:	0	0	0	0	0	5	5	5	5	9
S:	5	1	1	1	1	4	1	1	1	1

Problem: Can we guarantee *low distances* to roots?

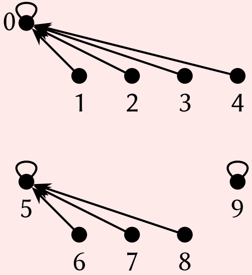
Consider Adding $(0,1)$, $(1,2)$, $(2,3)$, $(2,4)$, $(3,1)$, $(5,6)$, $(5,7)$, $(6,7)$, $(6,8)$, $(7,8)$, $(8,5)$, $(9,0)$, $(9,3)$, $(5,2)$.

Dynamic connectivity: A second attempt



F:	0	0	0	0	0	5	5	5	5	9
S:	5	1	1	1	1	4	1	1	1	1

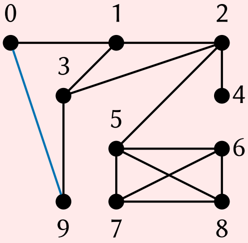
Representation visualization



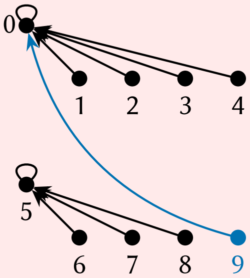
Problem: Can we guarantee *low distances* to roots?

Consider Adding (0, 1), (1, 2), (2, 3), (2, 4), (3, 1), (5, 6), (5, 7), (6, 7), (6, 8), (7, 8), (8, 5), (9, 0), (9, 3), (5, 2).

Dynamic connectivity: A second attempt



Representation visualization



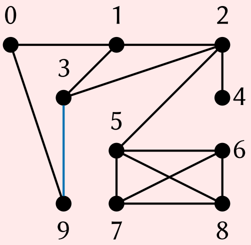
F:	0	0	0	0	0	5	5	5	5	0

S:	6	1	1	1	1	4	1	1	1	1

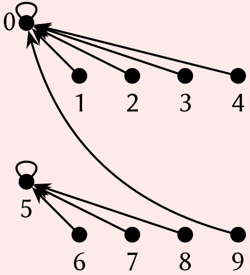
Problem: Can we guarantee *low distances* to roots?

Consider Adding $(0, 1)$, $(1, 2)$, $(2, 3)$, $(2, 4)$, $(3, 1)$, $(5, 6)$, $(5, 7)$, $(6, 7)$, $(6, 8)$, $(7, 8)$, $(8, 5)$, $(9, 0)$, $(9, 3)$, $(5, 2)$.

Dynamic connectivity: A second attempt



Representation visualization

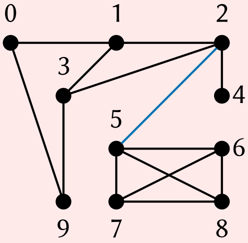


F:	0	0	0	0	0	5	5	5	5	0
S:	6	1	1	1	1	4	1	1	1	1

Problem: Can we guarantee *low distances* to roots?

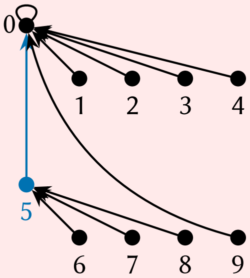
Consider Adding (0, 1), (1, 2), (2, 3), (2, 4), (3, 1), (5, 6), (5, 7), (6, 7), (6, 8), (7, 8), (8, 5), (9, 0), (9, 3), (5, 2).

Dynamic connectivity: A second attempt



F:	0	0	0	0	0	0	5	5	5	0
S:	10	1	1	1	1	4	1	1	1	1

Representation visualization



Problem: Can we guarantee *low distances* to roots?

Consider Adding (0, 1), (1, 2), (2, 3), (2, 4), (3, 1), (5, 6), (5, 7), (6, 7), (6, 8), (7, 8), (8, 5), (9, 0), (9, 3), (5, 2).

Dynamic connectivity: A second attempt

Problem: Can we guarantee *low distances* to roots?

Idea: Keep the *tree height* low when adding a connection (p, q):

- find the roots of the trees holding p and q;
- make the tree root of the *smallest* tree point to the other root.

How big do trees grow via this method?

Dynamic connectivity: A second attempt

Problem: Can we guarantee *low distances* to roots?

Idea: Keep the *tree height* low when adding a connection (p, q):

- find the roots of the trees holding p and q;
- make the tree root of the *smallest* tree point to the other root.

How big do trees grow via this method?

Theorem

The height of a tree T of size $|T|$ built in the above way is $\text{height}(T) \leq \log_2(|T|)$.

Dynamic connectivity: A second attempt

Problem: Can we guarantee *low distances* to roots?

Idea: Keep the *tree height* low when adding a connection (p, q):

- find the roots of the trees holding p and q;
- make the tree root of the *smallest* tree point to the other root.

How big do trees grow via this method?

Theorem

The height of a tree T of size $|T|$ built in the above way is $\mathrm{height}(T) \leq \log_2(|T|)$.

Proof

By induction on the size of tree T.

Dynamic connectivity: A second attempt

Theorem

The height of a tree T of size $|T|$ built in the above way is $\text{height}(T) \leq \log_2(|T|)$.

Proof

Induction Hypothesis: $\text{height}(T) \leq \log_2(|T|)$ for all trees with $|T| < i$.

Dynamic connectivity: A second attempt

Theorem

The height of a tree T of size $|T|$ built in the above way is $\text{height}(T) \leq \log_2(|T|)$.

Proof

Induction Hypothesis: $\text{height}(T) \leq \log_2(|T|)$ for all trees with $|T| < i$.

Base case Trees T of size $|T| = 1$ consists only of a root: $\text{height}(T) = \log_2(1) = 0$.

Dynamic connectivity: A second attempt

Theorem

The height of a tree T of size $|T|$ built in the above way is $\mathrm{height}(T) \leq \log_2(|T|)$.

Proof

Induction Hypothesis: $\mathrm{height}(T) \leq \log_2(|T|)$ for all trees with $|T| < i$.

Base case: Trees T of size $|T| = 1$ consists only of a root: $\mathrm{height}(T) = \log_2(1) = 0$.

Step: Consider combining two trees T_1 and T_2 into tree T of size $|T_1| + |T_2| = i$.

Dynamic connectivity: A second attempt

Theorem

The height of a tree T of size $|T|$ built in the above way is $\text{height}(T) \leq \log_2(|T|)$.

Proof

Induction Hypothesis: $\text{height}(T) \leq \log_2(|T|)$ for all trees with $|T| < i$.

Base case Trees T of size $|T| = 1$ consists only of a root: $\text{height}(T) = \log_2(1) = 0$.

Step Consider combining two trees T_1 and T_2 into tree T of size $|T_1| + |T_2| = i$.

- Assume $|T_1| \leq |T_2|$: add the root of T_1 as a child to the root of T_2.

Dynamic connectivity: A second attempt

Theorem

The height of a tree T of size $|T|$ built in the above way is $\text{height}(T) \leq \log_2(|T|)$.

Proof

Induction Hypothesis: $\text{height}(T) \leq \log_2(|T|)$ for all trees with $|T| < i$.

Base case Trees T of size $|T| = 1$ consists only of a root: $\text{height}(T) = \log_2(1) = 0$.

Step Consider combining two trees T_1 and T_2 into tree T of size $|T_1| + |T_2| = i$.

- Assume $|T_1| \leq |T_2|$: add the root of T_1 as a child to the root of T_2.
- Distance to root *increases* by one for all nodes in T_1.

$$\text{height}(T) = \max(\text{height}(T_1) + 1, \text{height}(T_2)).$$

Dynamic connectivity: A second attempt

Theorem

The height of a tree T of size $|T|$ built in the above way is $\text{height}(T) \le \log_2(|T|)$.

Proof

Induction Hypothesis: $\text{height}(T) \le \log_2(|T|)$ for all trees with $|T| < i$.

Base case Trees T of size $|T| = 1$ consists only of a root: $\text{height}(T) = \log_2(1) = 0$.

Step Consider combining two trees T_1 and T_2 into tree T of size $|T_1| + |T_2| = i$.

- Assume $|T_1| \le |T_2|$: add the root of T_1 as a child to the root of T_2.
- Distance to root *increases* by one for all nodes in T_1.
- By *IH*, $\text{height}(T_1) \le \log_2(|T_1|)$ and $\text{height}(T_2) \le \log_2(|T_2|)$.

$$\text{height}(T) \le \max(\log_2(|T_1|) + 1, \log_2(|T_2|)).$$

Dynamic connectivity: A second attempt

Theorem

The height of a tree T of size $|T|$ built in the above way is $\text{height}(T) \leq \log_2(|T|)$.

Proof

Induction Hypothesis: $\text{height}(T) \leq \log_2(|T|)$ for all trees with $|T| < i$.

Base case Trees T of size $|T| = 1$ consists only of a root: $\text{height}(T) = \log_2(1) = 0$.

Step Consider combining two trees T_1 and T_2 into tree T of size $|T_1| + |T_2| = i$.

- Assume $|T_1| \leq |T_2|$: add the root of T_1 as a child to the root of T_2.
- Distance to root *increases* by one for all nodes in T_1.
- By *IH*, $\text{height}(T_1) \leq \log_2(|T_1|)$ and $\text{height}(T_2) \leq \log_2(|T_2|)$.

$$\text{height}(T) \leq \max(\log_2(|T_1|) + 1, \log_2(|T_2|)).$$

Case $\log_2(|T_1|) + 1 \leq \log_2(|T_2|)$ $\text{height}(T) = \log_2(|T_2|)$.

Dynamic connectivity: A second attempt

Theorem

The height of a tree T of size $|T|$ built in the above way is $\text{height}(T) \leq \log_2(|T|)$.

Proof

Induction Hypothesis: $\text{height}(T) \leq \log_2(|T|)$ for all trees with $|T| < i$.

Base case Trees T of size $|T| = 1$ consists only of a root: $\text{height}(T) = \log_2(1) = 0$.

Step Consider combining two trees T_1 and T_2 into tree T of size $|T_1| + |T_2| = i$.

- Assume $|T_1| \leq |T_2|$: add the root of T_1 as a child to the root of T_2.
- Distance to root *increases* by one for all nodes in T_1.
- By *IH*, $\text{height}(T_1) \leq \log_2(|T_1|)$ and $\text{height}(T_2) \leq \log_2(|T_2|)$.

$$\text{height}(T) \leq \max(\log_2(|T_1|) + 1, \log_2(|T_2|)).$$

Case $\log_2(|T_1|) + 1 \leq \log_2(|T_2|)$ $\text{height}(T) = \log_2(|T_2|) < \log_2(|T|)$.

Dynamic connectivity: A second attempt

Theorem

The height of a tree T of size $|T|$ built in the above way is $\text{height}(T) \le \log_2(|T|)$.

Proof

Induction Hypothesis: $\text{height}(T) \le \log_2(|T|)$ for all trees with $|T| < i$.

Base case Trees T of size $|T| = 1$ consists only of a root: $\text{height}(T) = \log_2(1) = 0$.

Step Consider combining two trees T_1 and T_2 into tree T of size $|T_1| + |T_2| = i$.

- Assume $|T_1| \le |T_2|$: add the root of T_1 as a child to the root of T_2.
- Distance to root *increases* by one for all nodes in T_1.
- By *IH*, $\text{height}(T_1) \le \log_2(|T_1|)$ and $\text{height}(T_2) \le \log_2(|T_2|)$.

$$\text{height}(T) \le \max(\log_2(|T_1|) + 1, \log_2(|T_2|)).$$

Case $\log_2(|T_1|) + 1 > \log_2(|T_2|)$ $\text{height}(T) = \log_2(|T_1|) + 1$.

Dynamic connectivity: A second attempt

Theorem

The height of a tree T of size $|T|$ built in the above way is $\text{height}(T) \leq \log_2(|T|)$.

Proof

Induction Hypothesis: $\text{height}(T) \leq \log_2(|T|)$ for all trees with $|T| < i$.

Base case: Trees T of size $|T| = 1$ consists only of a root: $\text{height}(T) = \log_2(1) = 0$.

Step: Consider combining two trees T_1 and T_2 into tree T of size $|T_1| + |T_2| = i$.

- Assume $|T_1| \leq |T_2|$: add the root of T_1 as a child to the root of T_2.
- Distance to root *increases* by one for all nodes in T_1.
- By *IH*, $\text{height}(T_1) \leq \log_2(|T_1|)$ and $\text{height}(T_2) \leq \log_2(|T_2|)$.

$$\text{height}(T) \leq \max(\log_2(|T_1|) + 1, \log_2(|T_2|)).$$

Case $\log_2(|T_1|) + 1 > \log_2(|T_2|)$: $\text{height}(T) = \log_2(|T_1|) + 1 = \log_2(2|T_1|)$.

Dynamic connectivity: A second attempt

Theorem

The height of a tree T of size $|T|$ built in the above way is $\text{height}(T) \leq \log_2(|T|)$.

Proof

Induction Hypothesis: $\text{height}(T) \leq \log_2(|T|)$ for all trees with $|T| < i$.

Base case Trees T of size $|T| = 1$ consists only of a root: $\text{height}(T) = \log_2(1) = 0$.

Step Consider combining two trees T_1 and T_2 into tree T of size $|T_1| + |T_2| = i$.

- Assume $|T_1| \leq |T_2|$: add the root of T_1 as a child to the root of T_2.
- Distance to root *increases* by one for all nodes in T_1.
- By *IH*, $\text{height}(T_1) \leq \log_2(|T_1|)$ and $\text{height}(T_2) \leq \log_2(|T_2|)$.

$$\text{height}(T) \leq \max(\log_2(|T_1|) + 1, \log_2(|T_2|)).$$

Case $\log_2(|T_1|) + 1 > \log_2(|T_2|)$ $\text{height}(T) = \log_2(2|T_1|) \leq \log_2(|T_1| + |T_2|)$.

Dynamic connectivity: A second attempt

Theorem

The height of a tree T of size $|T|$ built in the above way is $\text{height}(T) \leq \log_2(|T|)$.

Proof

Induction Hypothesis: $\text{height}(T) \leq \log_2(|T|)$ for all trees with $|T| < i$.

Base case Trees T of size $|T| = 1$ consists only of a root: $\text{height}(T) = \log_2(1) = 0$.

Step Consider combining two trees T_1 and T_2 into tree T of size $|T_1| + |T_2| = i$.

- Assume $|T_1| \leq |T_2|$: add the root of T_1 as a child to the root of T_2.
- Distance to root *increases* by one for all nodes in T_1.
- By *IH*, $\text{height}(T_1) \leq \log_2(|T_1|)$ and $\text{height}(T_2) \leq \log_2(|T_2|)$.

$$\text{height}(T) \leq \max(\log_2(|T_1|) + 1, \log_2(|T_2|)).$$

Case $\log_2(|T_1|) + 1 > \log_2(|T_2|)$ $\text{height}(T) = \log_2(2|T_1|) \leq \log_2(|T_1| + |T_2|) = \log_2(|T|)$.

Dynamic connectivity: A second attempt

Theorem

The height of a tree T of size $|T|$ built in the above way is $\text{height}(T) \leq \log_2(|T|)$.

Proof

Induction Hypothesis: $\text{height}(T) \leq \log_2(|T|)$ for all trees with $|T| < i$.

Base case Trees T of size $|T| = 1$ consists only of a root: $\text{height}(T) = \log_2(1) = 0$.

Step Consider combining two trees T_1 and T_2 into tree T of size $|T_1| + |T_2| = i$.

- Assume $|T_1| \leq |T_2|$: add the root of T_1 as a child to the root of T_2.
- Distance to root *increases* by one for all nodes in T_1.
- By *IH*, $\text{height}(T_1) \leq \log_2(|T_1|)$ and $\text{height}(T_2) \leq \log_2(|T_2|)$.

$$\text{height}(T) \leq \max(\log_2(|T_1|) + 1, \log_2(|T_2|)) \leq \log_2(|T|).$$

Dynamic connectivity: A second attempt

Theorem

The height of a tree T of size $|T|$ built in the above way is $\text{height}(T) \leq \log_2(|T|)$.

Proof

Induction Hypothesis: $\text{height}(T) \leq \log_2(|T|)$ for all trees with $|T| < i$.

Base case Trees T of size $|T| = 1$ consists only of a root: $\text{height}(T) = \log_2(1) = 0$.

Step Consider combining two trees T_1 and T_2 into tree T of size $|T_1| + |T_2| = i$.

- Assume $|T_1| \leq |T_2|$: add the root of T_1 as a child to the root of T_2.
- Distance to root *increases* by one for all nodes in T_1.
- By *IH*, $\text{height}(T_1) \leq \log_2(|T_1|)$ and $\text{height}(T_2) \leq \log_2(|T_2|)$.

$$\text{height}(T) \leq \max(\log_2(|T_1|) + 1, \log_2(|T_2|)) \leq \log_2(|T|).$$

- What if $|T_1| > |T_2|$?
  Switch T_1 and T_2 arround in the above.

Dynamic connectivity: A second attempt

We want to optimize for *updating*: no updating of an entire array.

The forest representation

We need to find roots *fast*: costs of checking and updating depends on it!

Finding the root of p: depends on the *distance toward the root* (the depth of p):
Combining trees T by size: $\Theta\left(\log_2(|T|)\right)$.

Theorem

The height of a tree T of size $|T|$ built in the above way is $\text{height}(T) \leq \log_2(|T|)$.

Dynamic connectivity: Conclusion

We briefly looked at three solutions for *dynamic connectivity*.

	Check	*Update*	Minimize (worst case)
Simple representation	$\Theta(1)$	$\Theta(N)$	$\Theta\left(L + N^2\right)$
Forest representation	$\Theta(N)$	$\Theta(N)$	$\Theta\left(L + N^2\right)$
(size-based tree construction)	$\Theta(\log(N))$	$\Theta(\log(N))$	$\Theta(L \log_2(N))$

Suitable combination of data structures and algorithms: efficient dynamic connectivity!

Note: Chapter 1.5 in the book!

Kruskal's Algorithm

Algorithm MST-KRUSKAL($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*):

$E := \emptyset$.
while $|E| \neq |\mathcal{N}| - 1$ **do**
    Find an edge $(m, n) \in \mathcal{E}$ with minimum edge weight such that m and n are not yet connected in $(\mathcal{N}, E)$.
    $E := E \cup \{(m, n)\}$.
return E.

How to find edges $(m, n) \in \mathcal{E}$?

- Sort all edges on increasing edge weight. $\rightarrow \Theta\left(|\mathcal{E}| \log(|\mathcal{E}|)\right)$.
- Maintain a *dynamic connectivity* data structure D that represents the connected components in $(\mathcal{N}, E)$. $\rightarrow \Theta\left(\log(|\mathcal{N}|)\right)$
- For each edge (m, n) in sorted order: check whether they are connected via D. $\rightarrow \Theta\left(\log(|\mathcal{N}|)\right)$ per check.

Complexity. $\Theta\left(|\mathcal{E}| \log(|\mathcal{E}|) + |\mathcal{E}| \log(|\mathcal{N}|)\right)$.

Kruskal's Algorithm

Algorithm MST-KRUSKAL($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*):

1: $E := \emptyset$.
2: **while** $|E| \neq |\mathcal{N}| - 1$ **do**
3: Find an edge $(m, n) \in \mathcal{E}$ with minimum edge weight such that m and n are not yet connected in $(\mathcal{N}, E)$.
4: $E := E \cup \{(m, n)\}$.
5: **return** E.

How to find edges $(m, n) \in \mathcal{E}$?

- Sort all edges on increasing edge weight. $\rightarrow \Theta\left(|\mathcal{E}| \log(|\mathcal{E}|)\right)$.
- Maintain a *dynamic connectivity* data structure D that represents the connected components in $(\mathcal{N}, E)$. $\rightarrow \Theta\left(\log(|\mathcal{N}|)\right)$
- For each edge (m, n) in sorted order: check whether they are connected via D. $\rightarrow \Theta\left(\log(|\mathcal{N}|)\right)$ per check.

Complexity. $\Theta\left(|\mathcal{E}| \log(|\mathcal{E}|) + |\mathcal{E}| \log(|\mathcal{N}|)\right) = \Theta\left(|\mathcal{E}| \log(|\mathcal{E}|)\right)$.

Kruskal's Algorithm

Algorithm MST-KRUSKAL($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*):

1: $E := \emptyset$.
2: **while** $|E| \neq |\mathcal{N}| - 1$ **do**
3: Find an edge $(m, n) \in \mathcal{E}$ with minimum edge weight such that m and n are not yet connected in $(\mathcal{N}, E)$.
4: $E := E \cup \{(m, n)\}$.
5: **return** E.

How to find edges $(m, n) \in \mathcal{E}$?

- Sort all edges on increasing edge weight. $\rightarrow \Theta(|\mathcal{E}| \log(|\mathcal{E}|))$.
- Maintain a *dynamic connectivity* data structure D that represents the connected components in $(\mathcal{N}, E)$. $\rightarrow \Theta(\log(|\mathcal{N}|))$
- For each edge (m, n) in sorted order: check whether they are connected via D. $\rightarrow \Theta(\log(|\mathcal{N}|))$ per check.

Complexity. $\Theta(|\mathcal{E}| \log(|\mathcal{E}|) + |\mathcal{E}| \log(|\mathcal{N}|)) = \Theta(|\mathcal{E}| \log(|\mathcal{N}|))$.

Prim's Algorithm

Algorithm MST-HighLevel($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*):

```
E := ∅.
while (N, E) is not a spanning tree do
    Find an edge (m, n) ∈ ℰ such that E ∪ {(m, n)} is
    a subset of the edges of a minimum spanning tree of 𝒢.
    E := E ∪ {(m, n)}.
return E.
```

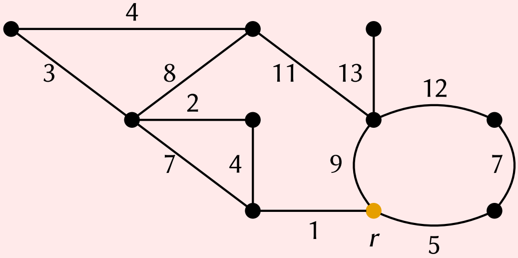
Algorithm MST-Prim($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*):

```
E, M := ∅, r with r a node from 𝒩.
while M ≠ 𝒩 do
    Find the lowest-weight edge (m, n) ∈ ℰ with m ∈ M and n ∉ M.
    E, M := E ∪ {(m, n)}, M ∪ {n}.
return E.
```

Prim's Algorithm

Algorithm MST-PRIM($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*):

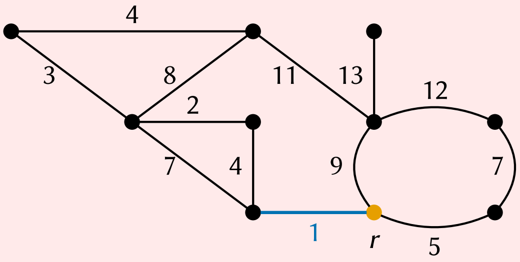
$E, M := \emptyset$, r with r a node from $\mathcal{N}$.
while $M \neq \mathcal{N}$ **do**
    Find the lowest-weight edge $(m, n) \in \mathcal{E}$ with $m \in M$ and $n \notin M$.
    $E, M := E \cup \{(m, n)\}, M \cup \{n\}$.
return E.



Prim's Algorithm

Algorithm MST-PRIM($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*):

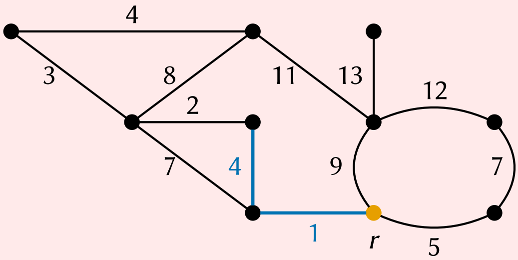
1: $E, M := \emptyset$, r with r a node from $\mathcal{N}$.
2: **while** $M \neq \mathcal{N}$ **do**
3: Find the lowest-weight edge $(m, n) \in \mathcal{E}$ with $m \in M$ and $n \notin M$.
4: $E, M := E \cup \{(m, n)\}, M \cup \{n\}$.
5: **return** E.



Prim's Algorithm

Algorithm MST-PRIM($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*):

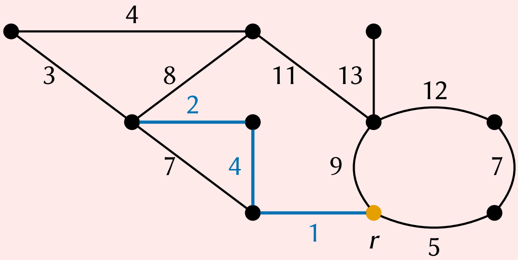
$E, M := \emptyset$, r with r a node from $\mathcal{N}$.
while $M \neq \mathcal{N}$ **do**
    Find the lowest-weight edge $(m, n) \in \mathcal{E}$ with $m \in M$ and $n \notin M$.
    $E, M := E \cup \{(m, n)\}, M \cup \{n\}$.
return E.



Prim's Algorithm

Algorithm MST-Prim($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*)**:**

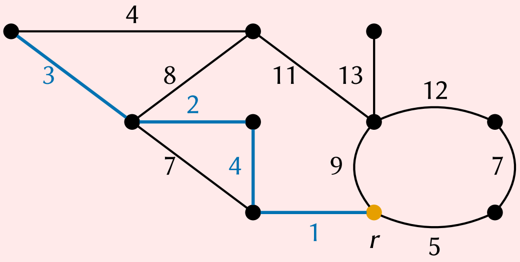
$E, M := \emptyset$, r with r a node from $\mathcal{N}$.
while $M \neq \mathcal{N}$ **do**
    Find the lowest-weight edge $(m, n) \in \mathcal{E}$ with $m \in M$ and $n \notin M$.
    $E, M := E \cup \{(m, n)\}, M \cup \{n\}$.
return E.



Prim's Algorithm

Algorithm MST-PRIM($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*):

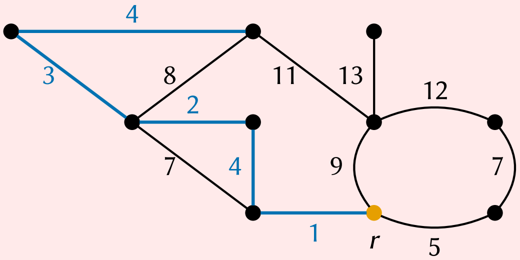
$E, M := \emptyset$, r with r a node from $\mathcal{N}$.
while $M \neq \mathcal{N}$ **do**
    Find the lowest-weight edge $(m, n) \in \mathcal{E}$ with $m \in M$ and $n \notin M$.
    $E, M := E \cup \{(m, n)\}, M \cup \{n\}$.
return E.



Prim's Algorithm

Algorithm MST-PRIM($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*):

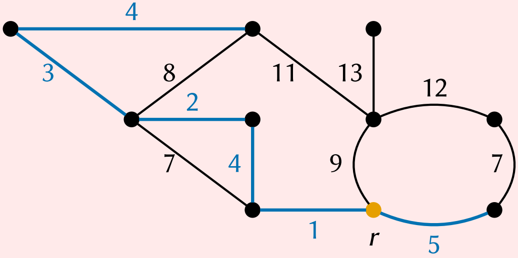
$E, M := \emptyset$, r with r a node from $\mathcal{N}$.
while $M \neq \mathcal{N}$ **do**
    Find the lowest-weight edge $(m, n) \in \mathcal{E}$ with $m \in M$ and $n \notin M$.
    $E, M := E \cup \{(m, n)\}, M \cup \{n\}$.
return E.



Prim's Algorithm

Algorithm MST-PRIM($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*):

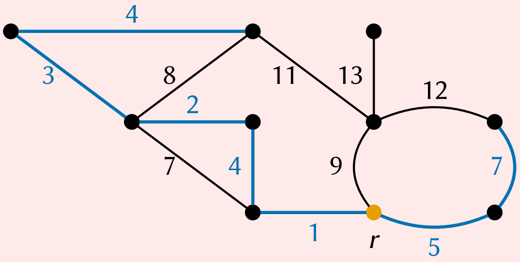
$E, M := \emptyset$, r with r a node from $\mathcal{N}$.
while $M \neq \mathcal{N}$ **do**
  Find the lowest-weight edge $(m, n) \in \mathcal{E}$ with $m \in M$ and $n \notin M$.
  $E, M := E \cup \{(m, n)\}, M \cup \{n\}$.
return E.



Prim's Algorithm

Algorithm MST-PRIM($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*):

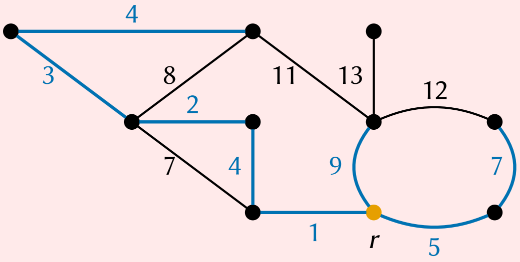
$E, M := \emptyset$, r with r a node from $\mathcal{N}$.
while $M \neq \mathcal{N}$ **do**
    Find the lowest-weight edge $(m, n) \in \mathcal{E}$ with $m \in M$ and $n \notin M$.
    $E, M := E \cup \{(m, n)\}, M \cup \{n\}$.
return E.



Prim's Algorithm

Algorithm MST-PRIM($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*):

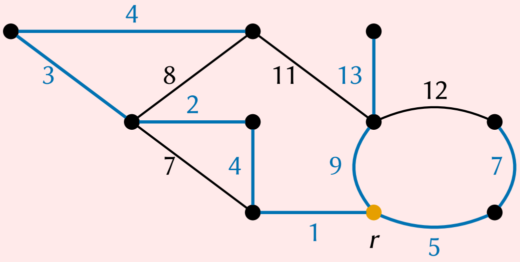
1: $E, M := \emptyset$, r with r a node from $\mathcal{N}$.
2: **while** $M \neq \mathcal{N}$ **do**
3: Find the lowest-weight edge $(m, n) \in \mathcal{E}$ with $m \in M$ and $n \notin M$.
4: $E, M := E \cup \{(m, n)\}, M \cup \{n\}$.
5: **return** E.



Prim's Algorithm

Algorithm MST-PRIM($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*):

1: $E, M := \emptyset$, r with r a node from $\mathcal{N}$.
2: **while** $M \neq \mathcal{N}$ **do**
3: Find the lowest-weight edge $(m, n) \in \mathcal{E}$ with $m \in M$ and $n \notin M$.
4: $E, M := E \cup \{(m, n)\}, M \cup \{n\}$.
5: **return** E.



Prim's Algorithm

Algorithm MST-PRIM($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*):

$E, M := \emptyset$, r with r a node from $\mathcal{N}$.
while $M \neq \mathcal{N}$ **do**
    Find the lowest-weight edge $(m, n) \in \mathcal{E}$ with $m \in M$ and $n \notin M$.
    $E, M := E \cup \{(m, n)\}, M \cup \{n\}$.
return E.

How to find edges $(m, n) \in \mathcal{E}$?

Prim's Algorithm

Algorithm MST-PRIM($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*):

$E, M := \emptyset$, r with r a node from $\mathcal{N}$.
while $M \neq \mathcal{N}$ **do**
    Find the lowest-weight edge $(m, n) \in \mathcal{E}$ with $m \in M$ and $n \notin M$.
    $E, M := E \cup \{(m, n)\}, M \cup \{n\}$.
return E.

How to find edges $(m, n) \in \mathcal{E}$?

- For every node $m \in M$: we can consider all edges $(m, n) \in \mathcal{E}$ with $n \notin N$.

Prim's Algorithm

Algorithm MST-PRIM($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*):

1: $E, M := \emptyset$, r with r a node from $\mathcal{N}$.
2: **while** $M \neq \mathcal{N}$ **do**
3: Find the lowest-weight edge $(m, n) \in \mathcal{E}$ with $m \in M$ and $n \notin M$.
4: $E, M := E \cup \{(m, n)\}, M \cup \{n\}$.
5: **return** E.

How to find edges $(m, n) \in \mathcal{E}$?

- For every node $m \in M$: we can consider all edges $(m, n) \in \mathcal{E}$ with $n \notin N$.
- *Alternatively*: for every node $n \notin M$, consider the edges $(m, n) \in \mathcal{E}$, $m \in M$.

Prim's Algorithm

Algorithm MST-PRIM($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*):

$E, M := \emptyset$, r with r a node from $\mathcal{N}$.
while $M \neq \mathcal{N}$ **do**
    Find the lowest-weight edge $(m, n) \in \mathcal{E}$ with $m \in M$ and $n \notin M$.
    $E, M := E \cup \{(m, n)\}, M \cup \{n\}$.
return E.

How to find edges $(m, n) \in \mathcal{E}$?

- For every node $m \in M$: we can consider all edges $(m, n) \in \mathcal{E}$ with $n \notin N$.
- *Alternatively*: for every node $n \notin M$, consider the edges $(m, n) \in \mathcal{E}$, $m \in M$.
- *Refine*: for every node $n \notin M$, consider the *minimal weight* edge $(m, n) \in \mathcal{E}$, $m \in M$.

Prim's Algorithm

Algorithm MST-PRIM($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*):

1: $E, M := \emptyset$, r with r a node from $\mathcal{N}$.
2: **while** $M \neq \mathcal{N}$ **do**
3: Find the lowest-weight edge $(m, n) \in \mathcal{E}$ with $m \in M$ and $n \notin M$.
4: $E, M := E \cup \{(m, n)\}, M \cup \{n\}$.
5: **return** E.

How to find edges $(m, n) \in \mathcal{E}$?

- For every node $m \in M$: we can consider all edges $(m, n) \in \mathcal{E}$ with $n \notin N$.
- *Alternatively*: for every node $n \notin M$, consider the edges $(m, n) \in \mathcal{E}$, $m \in M$.
- *Refine*: for every node $n \notin M$, consider the *minimal weight* edge $(m, n) \in \mathcal{E}$, $m \in M$.

We need a data structure to quickly find these minimal-weight edges for nodes $m \notin N$!

Prim's Algorithm

Algorithm $\textsc{MST-Prim}(\mathcal{G} = (\mathcal{N}, \mathcal{E}), \mathit{weight})$:

$E, M := \emptyset$, r with r a node from $\mathcal{N}$.
while $M \neq \mathcal{N}$ **do**
  Find the lowest-weight edge $(m, n) \in \mathcal{E}$ with $m \in M$ and $n \notin M$.
  $E, M := E \cup \{(m, n)\}, M \cup \{n\}$.
return E.

How to find edges $(m, n) \in \mathcal{E}$?

We need a data structure to quickly find these minimal-weight edges for nodes $m \notin N$!

Idea: a minimum-priority queue Q that holds nodes v with priority

$$p(Q, v) = \min\{\mathit{weight}((m, v)) \mid (m, v) \in \mathcal{E}, m \in M\}.$$

Store the minimal-weight edge (m, v) as additional information with each node v.

Prim's Algorithm

Algorithm MST-PRIM($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*):

$E, M := \emptyset, r$ with r a node from $\mathcal{N}$.
while $M \neq \mathcal{N}$ **do**
    Find the lowest-weight edge $(m, n) \in \mathcal{E}$ with $m \in M$ and $n \notin M$.
    $E, M := E \cup \{(m, n)\}, M \cup \{n\}$.
return E.

How to find edges $(m, n) \in \mathcal{E}$?

Idea: a minimum-priority queue Q that holds nodes v with priority

$$p(Q, v) = \min\{\mathit{weight}((m, v)) \mid (m, v) \in \mathcal{E}, m \in M\}.$$

Consider adding a node n to M. We need to update Q for every edge $(n, w) \in \mathcal{E}$.

Prim's Algorithm

Algorithm MST-PRIM($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*):

1: $E, M := \emptyset$, r with r a node from $\mathcal{N}$.
2: **while** $M \neq \mathcal{N}$ **do**
3: Find the lowest-weight edge $(m, n) \in \mathcal{E}$ with $m \in M$ and $n \notin M$.
4: $E, M := E \cup \{(m, n)\}, M \cup \{n\}$.
5: **return** E.

How to find edges $(m, n) \in \mathcal{E}$?

Idea: a minimum-priority queue Q that holds nodes v with priority

$$p(Q, v) = \min\{\mathit{weight}((m, v)) \mid (m, v) \in \mathcal{E}, m \in M\}.$$

Consider adding a node n to M. We need to update Q for every edge $(n, w) \in \mathcal{E}$.

- If $w \notin Q$: add w with weight $\mathit{weight}((n, w))$.

Prim's Algorithm

Algorithm MST-PRIM($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*):

$E, M := \emptyset$, r with r a node from $\mathcal{N}$.
while $M \neq \mathcal{N}$ **do**
  Find the lowest-weight edge $(m, n) \in \mathcal{E}$ with $m \in M$ and $n \notin M$.
  $E, M := E \cup \{(m, n)\}, M \cup \{n\}$.
return E.

How to find edges $(m, n) \in \mathcal{E}$?

Idea: a minimum-priority queue Q that holds nodes v with priority

$$p(Q, v) = \min\{\mathit{weight}((m, v)) \mid (m, v) \in \mathcal{E}, m \in M\}.$$

Consider adding a node n to M. We need to update Q for every edge $(n, w) \in \mathcal{E}$.

- If $w \notin Q$: add w with weight $\mathit{weight}((n, w))$. $\rightarrow \Theta(\log(|\mathcal{N}|))$ per edge.

Prim's Algorithm

Algorithm MST-PRIM($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*):

$E, M := \emptyset$, r with r a node from $\mathcal{N}$.
while $M \neq \mathcal{N}$ **do**
    Find the lowest-weight edge $(m, n) \in \mathcal{E}$ with $m \in M$ and $n \notin M$.
    $E, M := E \cup \{(m, n)\}, M \cup \{n\}$.
return E.

How to find edges $(m, n) \in \mathcal{E}$?

Idea: a minimum-priority queue Q that holds nodes v with priority

$$p(Q, v) = \min\{\mathit{weight}((m, v)) \mid (m, v) \in \mathcal{E}, m \in M\}.$$

Consider adding a node n to M. We need to update Q for every edge $(n, w) \in \mathcal{E}$.

- If $w \in Q$ *and* $\mathit{weight}((n, w)) < p(Q, w)$: lower $p(Q, w)$ to $\mathit{weight}((n, w))$.

Prim's Algorithm

Algorithm MST-PRIM($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*):

$E, M := \emptyset$, r with r a node from $\mathcal{N}$.
while $M \neq \mathcal{N}$ **do**
    Find the lowest-weight edge $(m, n) \in \mathcal{E}$ with $m \in M$ and $n \notin M$.
    $E, M := E \cup \{(m, n)\}, M \cup \{n\}$.
return E.

How to find edges $(m, n) \in \mathcal{E}$?

Idea: a minimum-priority queue Q that holds nodes v with priority

$$p(Q, v) = \min\{\mathit{weight}((m, v)) \mid (m, v) \in \mathcal{E}, m \in M\}.$$

Consider adding a node n to M. We need to update Q for every edge $(n, w) \in \mathcal{E}$.

- If *$w \in Q$ and $\mathit{weight}((n, w)) < p(Q, w)$*: lower $p(Q, w)$ to $\mathit{weight}((n, w))$.
  If Q is a min-heap: lowering $p(Q, w)$ is a swim operation *after* we find $w \in Q$.

Prim's Algorithm

Algorithm MST-PRIM($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*):

1: $E, M := \emptyset$, r with r a node from $\mathcal{N}$.
2: **while** $M \neq \mathcal{N}$ **do**
3: Find the lowest-weight edge $(m, n) \in \mathcal{E}$ with $m \in M$ and $n \notin M$.
4: $E, M := E \cup \{(m, n)\}, M \cup \{n\}$.
5: **return** E.

How to find edges $(m, n) \in \mathcal{E}$?

Idea: a minimum-priority queue Q that holds nodes v with priority

$$p(Q, v) = \min\{\mathit{weight}((m, v)) \mid (m, v) \in \mathcal{E}, m \in M\}.$$

Consider adding a node n to M. We need to update Q for every edge $(n, w) \in \mathcal{E}$.

- If $w \in Q$ *and* $\mathit{weight}((n, w)) < p(Q, w)$: lower $p(Q, w)$ to $\mathit{weight}((n, w))$.
  If Q is a min-heap: lowering $p(Q, w)$ is a swim operation *after* we find $w \in Q$.
  To find $w \in Q$: keep track of the position of every node in Q via an array.

Prim's Algorithm

Algorithm MST-PRIM($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*):

1: $E, M := \emptyset$, r with r a node from $\mathcal{N}$.
2: **while** $M \neq \mathcal{N}$ **do**
3: Find the lowest-weight edge $(m, n) \in \mathcal{E}$ with $m \in M$ and $n \notin M$.
4: $E, M := E \cup \{(m, n)\}, M \cup \{n\}$.
5: **return** E.

How to find edges $(m, n) \in \mathcal{E}$?

Idea: a minimum-priority queue Q that holds nodes v with priority

$$p(Q, v) = \min\{\mathit{weight}((m, v)) \mid (m, v) \in \mathcal{E}, m \in M\}.$$

Consider adding a node n to M. We need to update Q for every edge $(n, w) \in \mathcal{E}$.

- If $w \in Q$ *and* $\mathit{weight}((n, w)) < p(Q, w)$: lower $p(Q, w)$ to $\mathit{weight}((n, w))$.
  $\longrightarrow \Theta(\log(|\mathcal{N}|))$ per edge.

Prim's Algorithm

Algorithm MST-PRIM($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*):

$E, M := \emptyset$, r with r a node from $\mathcal{N}$.
while $M \neq \mathcal{N}$ **do**
    Find the lowest-weight edge $(m, n) \in \mathcal{E}$ with $m \in M$ and $n \notin M$.
    $E, M := E \cup \{(m, n)\}, M \cup \{n\}$.
return E.

How to find edges $(m, n) \in \mathcal{E}$?

Idea: a minimum-priority queue Q that holds nodes v with priority

$$p(Q, v) = \min\{\mathit{weight}((m, v)) \mid (m, v) \in \mathcal{E}, m \in M\}.$$

Consider adding a node n to M. We need to update Q for every edge $(n, w) \in \mathcal{E}$.

$\rightarrow \Theta(\log(|\mathcal{N}|))$ per edge.

Prim's Algorithm

Algorithm MST-PRIM($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*):

$E, M := \emptyset$, r with r a node from $\mathcal{N}$.
while $M \neq \mathcal{N}$ **do**
  Find the lowest-weight edge $(m, n) \in \mathcal{E}$ with $m \in M$ and $n \notin M$.
  $E, M := E \cup \{(m, n)\}, M \cup \{n\}$.
return E.

How to find edges $(m, n) \in \mathcal{E}$?

Idea: a minimum-priority queue Q that holds nodes v with priority

$$p(Q, v) = \min\{weight((m, v)) \mid (m, v) \in \mathcal{E}, m \in M\}.$$

Consider adding a node n to M. We need to update Q for every edge $(n, w) \in \mathcal{E}$.
$\longrightarrow \Theta(\log(|\mathcal{N}|))$ per edge.

To find the next edge to add: remove nodes n from Q until $n \notin M$.

Prim's Algorithm

Algorithm MST-PRIM($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*):

1: $E, M := \emptyset$, r with r a node from $\mathcal{N}$.
2: **while** $M \neq \mathcal{N}$ **do**
3: Find the lowest-weight edge $(m, n) \in \mathcal{E}$ with $m \in M$ and $n \notin M$.
4: $E, M := E \cup \{(m, n)\}, M \cup \{n\}$.
5: **return** E.

How to find edges $(m, n) \in \mathcal{E}$?

Idea: a minimum-priority queue Q that holds nodes v with priority

$$p(Q, v) = \min\{\mathit{weight}((m, v)) \mid (m, v) \in \mathcal{E}, m \in M\}.$$

Consider adding a node n to M. We need to update Q for every edge $(n, w) \in \mathcal{E}$.
$\longrightarrow \Theta(\log(|\mathcal{N}|))$ per edge.

To find the next edge to add: remove nodes n from Q until $n \notin M$.

Complexity.

Prim's Algorithm

Algorithm $\textsc{MST-Prim}(\mathcal{G} = (\mathcal{N}, \mathcal{E}), \mathit{weight})$:

$E, M := \emptyset$, r with r a node from $\mathcal{N}$.
while $M \neq \mathcal{N}$ **do**
    Find the lowest-weight edge $(m, n) \in \mathcal{E}$ with $m \in M$ and $n \notin M$.
    $E, M := E \cup \{(m, n)\}, M \cup \{n\}$.
return E.

How to find edges $(m, n) \in \mathcal{E}$?

Idea: a minimum-priority queue Q that holds nodes v with priority

$$p(Q, v) = \min\{\mathit{weight}((m, v)) \mid (m, v) \in \mathcal{E}, m \in M\}.$$

Consider adding a node n to M. We need to update Q for every edge $(n, w) \in \mathcal{E}$.
$\longrightarrow \Theta\left(\log(|\mathcal{N}|)\right)$ per edge.

To find the next edge to add: remove nodes n from Q until $n \notin M$.

Complexity. $\Theta\left(|\mathcal{E}| \log(|\mathcal{N}|)\right)$.

Prim's Algorithm

Algorithm MST-PRIM($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*):

$E, M := \emptyset$, r with r a node from $\mathcal{N}$.
while $M \neq \mathcal{N}$ **do**
  Find the lowest-weight edge $(m, n) \in \mathcal{E}$ with $m \in M$ and $n \notin M$.
  $E, M := E \cup \{(m, n)\}, M \cup \{n\}$.
return E.

How to find edges $(m, n) \in \mathcal{E}$?

Idea: a minimum-priority queue Q that holds nodes v with priority

$$p(Q, v) = \min\{weight((m, v)) \mid (m, v) \in \mathcal{E}, m \in M\}.$$

Consider adding a node n to M. We need to update Q for every edge $(n, w) \in \mathcal{E}$.Fibonacci Heap: $\Theta(\log(|\mathcal{N}|))$ to add or remove a value, amortized $\Theta(1)$ to lower a weight.

To find the next edge to add: remove nodes n from Q until $n \notin M$.

Complexity. $\Theta(|\mathcal{N}| \log(|\mathcal{N}|) + |\mathcal{E}|)$.

Problem: Directions

Problem

Consider a weighted directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ in which

- the nodes $\mathcal{N}$ represent road crossings;
- the edges $\mathcal{E}$ are the roads connecting these crossings; and
- the weights $\mathit{weight}((m, n))$ represent the cost to travel along the road (m, n) (e.g., length of the road, duration of traveling along the road, fuel costs, ...).

Provide the *shortest* route from crossing A to crossing B.

Problem: Directions

Problem

Consider a weighted directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ in which

- the nodes $\mathcal{N}$ represent road crossings;
- the edges $\mathcal{E}$ are the roads connecting these crossings; and
- the weights $\mathit{weight}((m, n))$ represent the cost to travel along the road (m, n) (e.g., length of the road, duration of traveling along the road, fuel costs, $\dots$).

Provide the *shortest* route from crossing A to crossing B.

Shortest in terms of the provided weight:
No other path from A and B should have a *lower* sum of edge weights.

Problem: Directions

Problem

Consider a weighted directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ in which

- the nodes $\mathcal{N}$ represent road crossings;
- the edges $\mathcal{E}$ are the roads connecting these crossings; and
- the weights $\mathit{weight}((m, n))$ represent the cost to travel along the road (m, n) (e.g., length of the road, duration of traveling along the road, fuel costs, …).

Provide the *shortest* route from crossing A to crossing B.

Definition

Let $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ be a directed graph with edge weights *weight*.

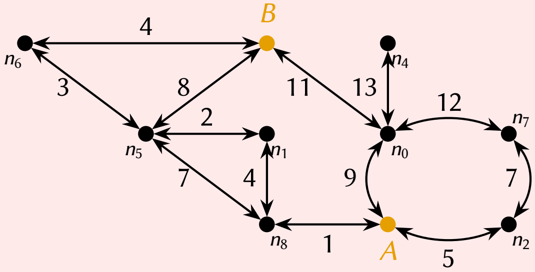
A *shortest path* from $A \in \mathcal{N}$ to $B \in \mathcal{N}$ is a directed path from A to B in which the sum of edge weights is minimal (no other path from A to B has a lower sum of edge weights).

Problem: Directions

Definition

Let $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ be a directed graph with edge weights *weight*.

A *shortest path* from $A \in \mathcal{N}$ to $B \in \mathcal{N}$ is a directed path from A to B in which the sum of edge weights is minimal (no other path from A to B has a lower sum of edge weights).

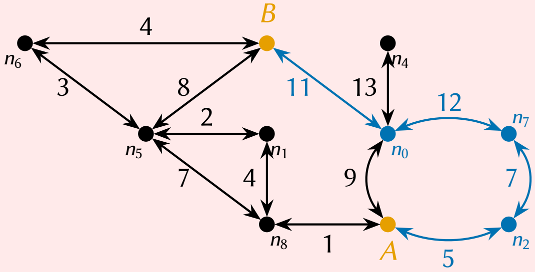


Problem: Directions

Definition

Let $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ be a directed graph with edge weights *weight*.

A *shortest path* from $A \in \mathcal{N}$ to $B \in \mathcal{N}$ is a directed path from A to B in which the sum of edge weights is minimal (no other path from A to B has a lower sum of edge weights).



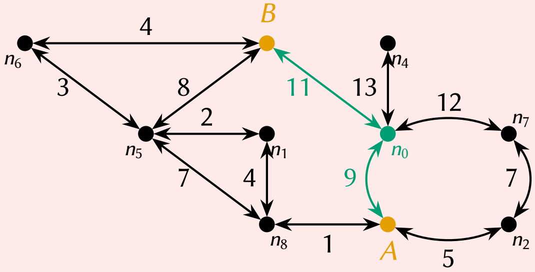
$An_2 n_7 n_0 B \longrightarrow 35$

Problem: Directions

Definition

Let $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ be a directed graph with edge weights *weight*.

A *shortest path* from $A \in \mathcal{N}$ to $B \in \mathcal{N}$ is a directed path from A to B in which the sum of edge weights is minimal (no other path from A to B has a lower sum of edge weights).



$An_2n_7n_0B \longrightarrow 35$

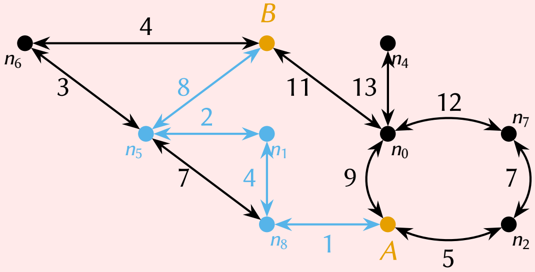
$An_0B \longrightarrow 20$

Problem: Directions

Definition

Let $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ be a directed graph with edge weights *weight*.

A *shortest path* from $A \in \mathcal{N}$ to $B \in \mathcal{N}$ is a directed path from A to B in which the sum of edge weights is minimal (no other path from A to B has a lower sum of edge weights).



$A n_2 n_7 n_0 B \longrightarrow 35$

$A n_0 B \longrightarrow 20$

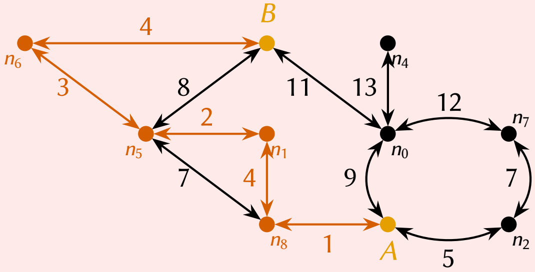
$A n_8 n_1 n_5 B \longrightarrow 15$

Problem: Directions

Definition

Let $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ be a directed graph with edge weights *weight*.

A *shortest path* from $A \in \mathcal{N}$ to $B \in \mathcal{N}$ is a directed path from A to B in which the sum of edge weights is minimal (no other path from A to B has a lower sum of edge weights).



$An_2n_7n_0B \longrightarrow 35$
$An_0B \longrightarrow 20$
$An_8n_1n_5B \longrightarrow 15$
$An_8n_1n_5n_6B \longrightarrow 14$

Problem: Directions

Definition

Let $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ be a directed graph with edge weights *weight*.

A *shortest path* from $A \in \mathcal{N}$ to $B \in \mathcal{N}$ is a directed path from A to B in which the sum of edge weights is minimal (no other path from A to B has a lower sum of edge weights).

The *single-source shortest path* problem

Given a directed edge-weighted graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ and source node $s \in \mathcal{N}$, find *a* shortest path (if any) from s to every target node $t \in \mathcal{N}$.

Problem: Directions

Definition

Let $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ be a directed graph with edge weights *weight*.

A *shortest path* from $A \in \mathcal{N}$ to $B \in \mathcal{N}$ is a directed path from A to B in which the sum of edge weights is minimal (no other path from A to B has a lower sum of edge weights).

The *single-source shortest path* problem

Given a directed edge-weighted graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ and source node $s \in \mathcal{N}$,
find *a* shortest path (if any) from s to every target node $t \in \mathcal{N}$.

The *all-pairs shortest path* problem

Given a directed edge-weighted graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ and source node $s \in \mathcal{N}$,
find *a* shortest path (if any) between every pair of source and target nodes $(s, t) \in \mathcal{N} \times \mathcal{N}$.

Problem: Directions

Definition

Let $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ be a directed graph with edge weights *weight*.

A *shortest path* from $A \in \mathcal{N}$ to $B \in \mathcal{N}$ is a directed path from A to B in which the sum of edge weights is minimal (no other path from A to B has a lower sum of edge weights).

The *single-source shortest path* problem

Given a directed edge-weighted graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ and source node $s \in \mathcal{N}$,
find *a* shortest path (if any) from s to every target node $t \in \mathcal{N}$.

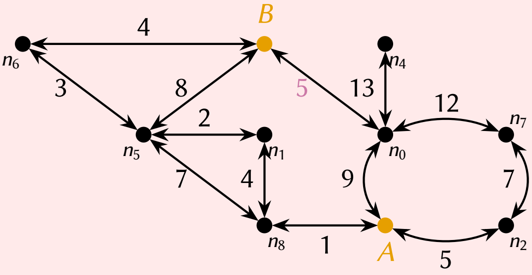
The *all-pairs shortest path* problem

Given a directed edge-weighted graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ and source node $s \in \mathcal{N}$,
find *a* shortest path (if any) between every pair of source and target nodes $(s, t) \in \mathcal{N} \times \mathcal{N}$.

Reminder breadth-first search answers the
single-source shortest path problem on *unweighted graphs*.

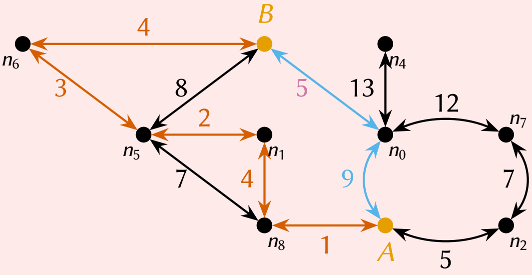
Shortest paths

What is *the* shortest path from A to B in this graph?



Shortest paths

What is *the* shortest path from A to B in this graph?

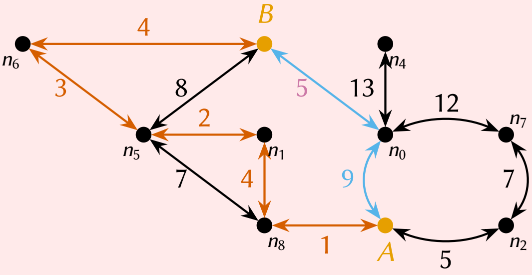


There are two options with total weight 14

$An_8 n_1 n_5 n_6 B$ and $An_0 B$.

Shortest paths

What is *the* shortest path from A to B in this graph?



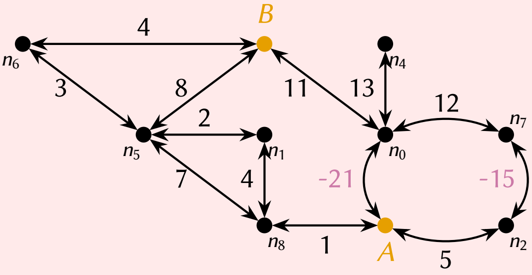
There are two options with total weight 14

$An_8 n_1 n_5 n_6 B$ and $An_0 B$.

Shortest path algorithms internally choose one of these options.

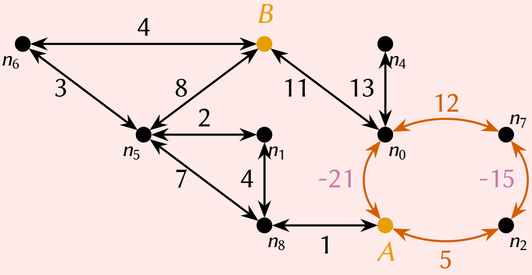
Shortest paths and negative weights

What is the shortest path from A to B in this graph?



Shortest paths and negative weights

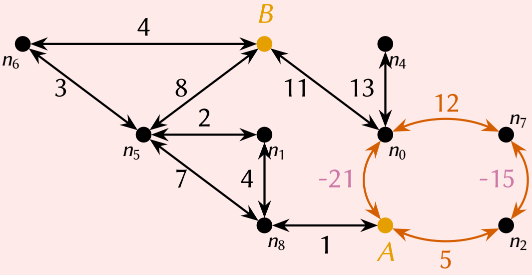
What is the shortest path from A to B in this graph?



- For any path: adding steps $An_2n_7n_0A$ *reduces* the weight of that path by 19!

Shortest paths and negative weights

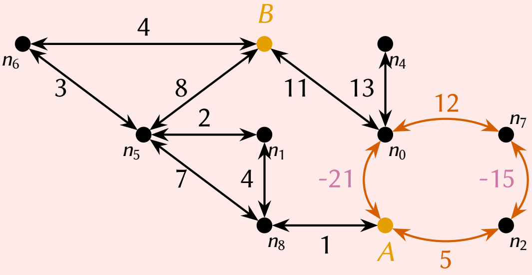
What is the shortest path from A to B in this graph?



- For any path: adding steps $An_2n_7n_0A$ *reduces* the weight of that path by 19!
- The "shortest path" has infinite length and infinitely negative weight?

Shortest paths and negative weights

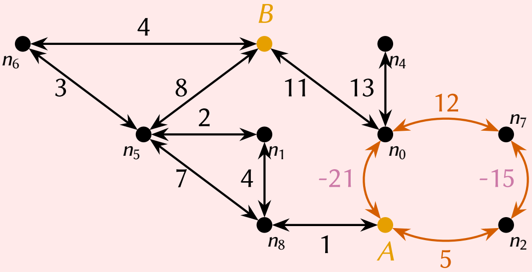
What is the shortest path from A to B in this graph?



- For any path: adding steps $An_2n_7n_0A$ *reduces* the weight of that path by 19!
- The "shortest path" has infinite length and infinitely negative weight?
- *Solution*? Require a *simple path* (without repeating nodes).

Shortest paths and negative weights

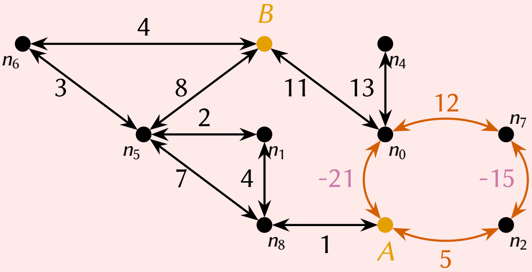
What is the shortest path from A to B in this graph?



- For any path: adding steps *$An_2n_7n_0A$ reduces* the weight of that path by 19!
- The “shortest path” has infinite length and infinitely negative weight?
- *Solution*? Require a *simple path* (without repeating nodes).
  Determining whether a solution of cost k exists is an *NP-complete problem*: no practical algorithms known.

Shortest paths and negative weights

What is the shortest path from A to B in this graph?

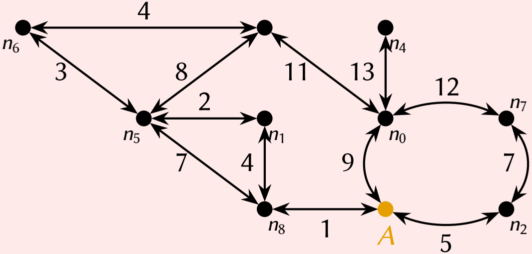


- For any path: adding steps $A n_2 n_7 n_0 A$ *reduces* the weight of that path by 19!
- The "shortest path" has infinite length and infinitely negative weight?
- *Solution*? Require a *simple path* (without repeating nodes).
  Determining whether a solution of cost k exists is an *NP-complete problem*: no practical algorithms known.

Solution. Either disallow *negative edge weights* or *negative-weight cycles*.

Representing the shortest paths from a source node

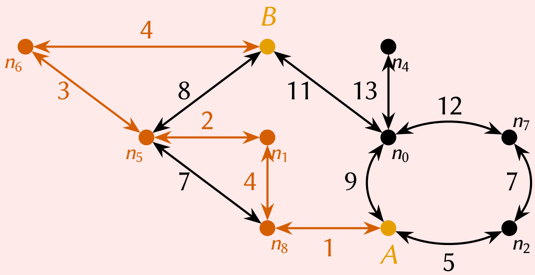
Consider the single-source shortest paths from A



Representing the shortest paths from a source node

Consider the single-source shortest paths from A

- We have already seen that $An_8n_1n_5n_6B$ is the shortest path from A to B.

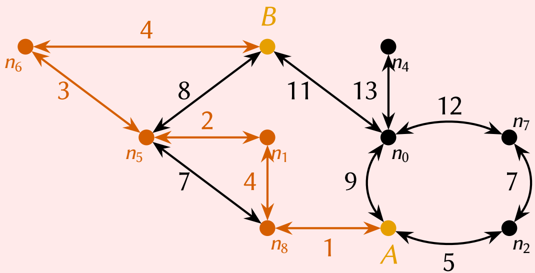


Representing the shortest paths from a source node

Consider the single-source shortest paths from A

- We have already seen that $An_8n_1n_5n_6B$ is the shortest path from A to B.
- The paths $An_8n_1n_5n_6$, $An_8n_1n_5$, An_8n_1, and An_8 are also shortest paths!

Can we represent *all* shortest paths from A without enumerating all of them?



Representing the shortest paths from a source node

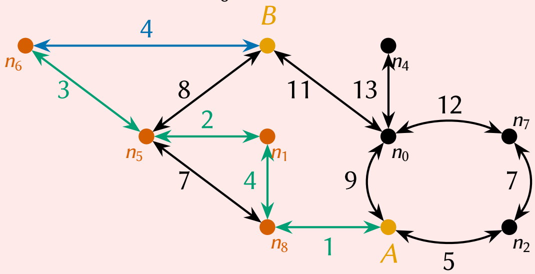
Consider the single-source shortest paths from A

- We have already seen that $An_8n_1n_5n_6B$ is the shortest path from A to B.
- The paths $An_8n_1n_5n_6$, $An_8n_1n_5$, An_8n_1, and An_8 are also shortest paths!

Can we represent *all* shortest paths from A without enumerating all of them?

Observation: The shortest path to B consists of two parts:

- the last edge (from node n_6 to node B); and
- the shortest path from A to node n_6.



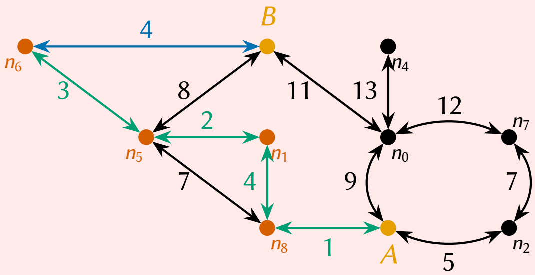
Representing the shortest paths from a source node

Consider the single-source shortest paths from A

- We have already seen that $An_8n_1n_5n_6B$ is the shortest path from A to B.
- The paths $An_8n_1n_5n_6$, $An_8n_1n_5$, An_8n_1, and An_8 are also shortest paths!

Can we represent *all* shortest paths from A without enumerating all of them?

Idea: Store, for each node, the *previous node* on the path.



Representing the shortest paths from a source node

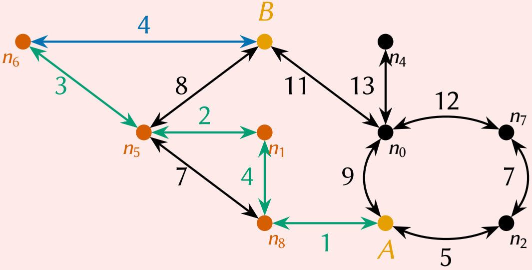
Consider the single-source shortest paths from A

- We have already seen that $An_8n_1n_5n_6B$ is the shortest path from A to B.
- The paths $An_8n_1n_5n_6$, $An_8n_1n_5$, An_8n_1, and An_8 are also shortest paths!

Can we represent *all* shortest paths from A without enumerating all of them?

Idea: Store, for each node, the *previous node* on the path.

- We can use an array to store this information per node.



n_0:	
n_1:	n_8
n_2:	
A:	A
n_4:	
n_5:	n_1
n_6:	n_5
n_7:	
n_8:	A
B:	n_6

Representing the shortest paths from a source node

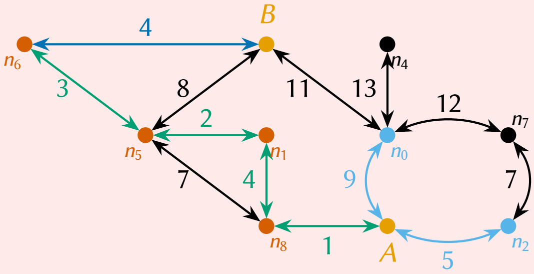
Consider the single-source shortest paths from A

- We have already seen that $An_8n_1n_5n_6B$ is the shortest path from A to B.
- The paths $An_8n_1n_5n_6$, $An_8n_1n_5$, An_8n_1, and An_8 are also shortest paths!

Can we represent *all* shortest paths from A without enumerating all of them?

Idea: Store, for each node, the *previous node* on the path.

- We can use an array to store this information per node.



n_0:	A
n_1:	n_8
n_2:	A
A:	A
n_4:	
n_5:	n_1
n_6:	n_5
n_7:	
n_8:	A
B:	n_6

Representing the shortest paths from a source node

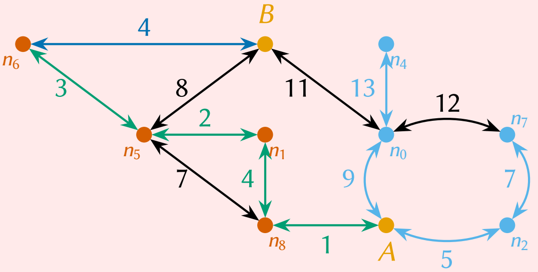
Consider the single-source shortest paths from A

- We have already seen that $An_8n_1n_5n_6B$ is the shortest path from A to B.
- The paths $An_8n_1n_5n_6$, $An_8n_1n_5$, An_8n_1, and An_8 are also shortest paths!

Can we represent *all* shortest paths from A without enumerating all of them?

Idea: Store, for each node, the *previous node* on the path.

- We can use an array to store this information per node.



n_0:	A
n_1:	n_8
n_2:	A
A:	A
n_4:	n_0
n_5:	n_1
n_6:	n_5
n_7:	n_2
n_8:	A
B:	n_6

Representing the shortest paths from a source node

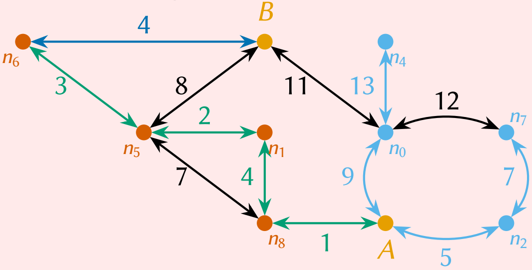
Consider the single-source shortest paths from A

- We have already seen that $An_8n_1n_5n_6B$ is the shortest path from A to B.
- The paths $An_8n_1n_5n_6$, $An_8n_1n_5$, An_8n_1, and An_8 are also shortest paths!

Can we represent *all* shortest paths from A without enumerating all of them?

Idea: Store, for each node, the *previous node* on the path.

- We can use an array to store this information per node.
- We can also store the total weight (cost) to reach each node.



n_0:	A	
n_1:	n_8	
n_2:	A	
A:	A	0
n_4:	n_0	
n_5:	n_1	
n_6:	n_5	
n_7:	n_2	
n_8:	A	
B:	n_6	

Representing the shortest paths from a source node

Consider the single-source shortest paths from A

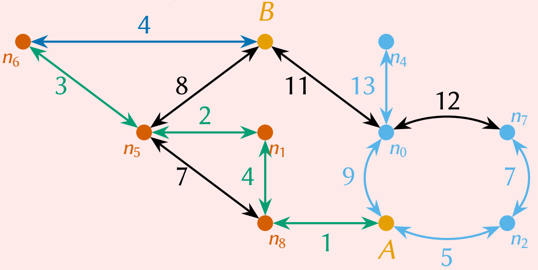
- We have already seen that $An_8n_1n_5n_6B$ is the shortest path from A to B.
- The paths $An_8n_1n_5n_6$, $An_8n_1n_5$, An_8n_1, and An_8 are also shortest paths!

Can we represent *all* shortest paths from A without enumerating all of them?

Idea: Store, for each node, the *previous node* on the path.

- We can use an array to store this information per node.
- We can also store the total weight (cost) to reach each node.

n_0:	A	9
n_1:	n_8	
n_2:	A	5
A:	A	0
n_4:	n_0	
n_5:	n_1	
n_6:	n_5	
n_7:	n_2	
n_8:	A	1
B:	n_6	



Representing the shortest paths from a source node

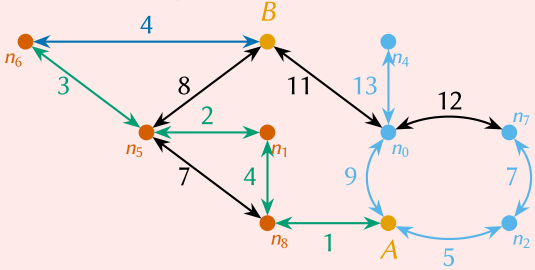
Consider the single-source shortest paths from A

- We have already seen that $An_8n_1n_5n_6B$ is the shortest path from A to B.
- The paths $An_8n_1n_5n_6$, $An_8n_1n_5$, An_8n_1, and An_8 are also shortest paths!

Can we represent *all* shortest paths from A without enumerating all of them?

Idea: Store, for each node, the *previous node* on the path.

- We can use an array to store this information per node.
- We can also store the total weight (cost) to reach each node.



n_0:	A	9
n_1:	n_8	5
n_2:	A	5
A:	A	0
n_4:	n_0	22
n_5:	n_1	
n_6:	n_5	
n_7:	n_2	12
n_8:	A	1
B:	n_6	

Representing the shortest paths from a source node

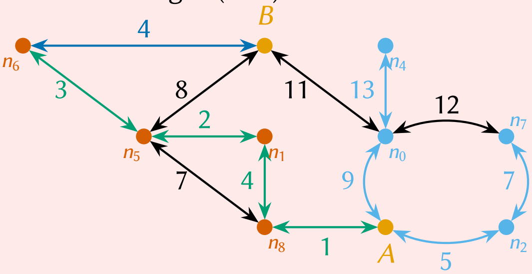
Consider the single-source shortest paths from A

- We have already seen that $An_8n_1n_5n_6B$ is the shortest path from A to B.
- The paths $An_8n_1n_5n_6$, $An_8n_1n_5$, An_8n_1, and An_8 are also shortest paths!

Can we represent *all* shortest paths from A without enumerating all of them?

Idea: Store, for each node, the *previous node* on the path.

- We can use an array to store this information per node.
- We can also store the total weight (cost) to reach each node.



n_0:	A	9
n_1:	n_8	5
n_2:	A	5
A:	A	0
n_4:	n_0	22
n_5:	n_1	7
n_6:	n_5	
n_7:	n_2	12
n_8:	A	1
B:	n_6	

Representing the shortest paths from a source node

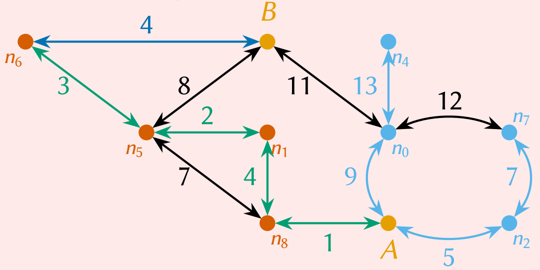
Consider the single-source shortest paths from A

- We have already seen that $An_8n_1n_5n_6B$ is the shortest path from A to B.
- The paths $An_8n_1n_5n_6$, $An_8n_1n_5$, An_8n_1, and An_8 are also shortest paths!

Can we represent *all* shortest paths from A without enumerating all of them?

Idea: Store, for each node, the *previous node* on the path.

- We can use an array to store this information per node.
- We can also store the total weight (cost) to reach each node.



n_0:	A	9
n_1:	n_8	5
n_2:	A	5
A:	A	0
n_4:	n_0	22
n_5:	n_1	7
n_6:	n_5	10
n_7:	n_2	12
n_8:	A	1
B:	n_6	

Representing the shortest paths from a source node

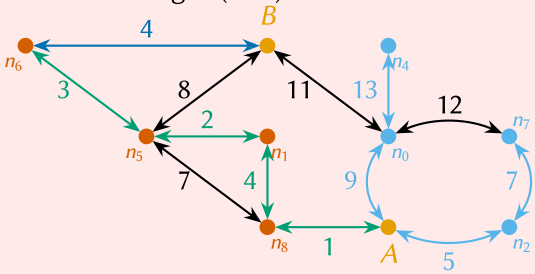
Consider the single-source shortest paths from A

- We have already seen that $An_8n_1n_5n_6B$ is the shortest path from A to B.
- The paths $An_8n_1n_5n_6$, $An_8n_1n_5$, An_8n_1, and An_8 are also shortest paths!

Can we represent *all* shortest paths from A without enumerating all of them?

Idea: Store, for each node, the *previous node* on the path.

- We can use an array to store this information per node.
- We can also store the total weight (cost) to reach each node.



n_0:	A	9
n_1:	n_8	5
n_2:	A	5
A:	A	0
n_4:	n_0	22
n_5:	n_1	7
n_6:	n_5	10
n_7:	n_2	12
n_8:	A	1
B:	n_6	14

Representing the shortest paths from a source node

Consider the single-source shortest paths from A

- We have already seen that $An_8n_1n_5n_6B$ is the shortest path from A to B.
- The paths $An_8n_1n_5n_6$, $An_8n_1n_5$, An_8n_1, and An_8 are also shortest paths!

Can we represent *all* shortest paths from A without enumerating all of them?

Idea: Store, for each node, the *previous node* on the path.

- We can use an array to store this information per node.
- We can also store the total weight (cost) to reach each node.

Generalization to the all-pairs shortest path problem

A $|\mathcal{N}| \times |\mathcal{N}|$ matrix representing one such array *per* node.

A single-source shortest path algorithm

Algorithm SSSP-HIGHLEVEL($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*, $s \in \mathcal{N}$)**:**

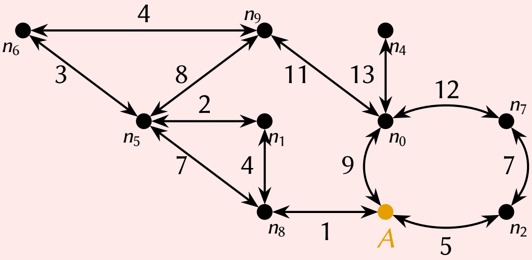
$path, cost := [n \mapsto ?\ |\ n \in \mathcal{N}], [n \mapsto \infty\ |\ n \in \mathcal{N}]$.

$path[s], cost[s] := s, 0$.

while $(m, n) \in \mathcal{E}$ with $cost[m] + weight((m, n)) < cost[n]$ **do**

    $path[n], cost[n] := m, weight((m, n)) + cost[m]$.

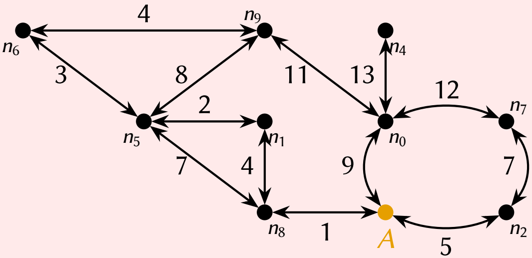
return $path, cost$.



A single-source shortest path algorithm

Algorithm SSSP-HighLevel($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $weight$, $s \in \mathcal{N}$)**:**

$path, cost := [n \mapsto ?\ |\ n \in \mathcal{N}], [n \mapsto \infty\ |\ n \in \mathcal{N}]$.
$path[s], cost[s] := s, 0$.
while $(m, n) \in \mathcal{E}$ with $cost[m] + weight((m, n)) < cost[n]$ **do**
    $path[n], cost[n] := m, weight((m, n)) + cost[m]$.
return $path, cost$.

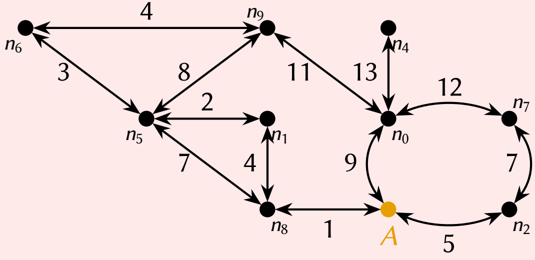


n_0:	?	∞
n_1:	?	∞
n_2:	?	∞
A:	?	∞
n_4:	?	∞
n_5:	?	∞
n_6:	?	∞
n_7:	?	∞
n_8:	?	∞
n_9:	?	∞

A single-source shortest path algorithm

Algorithm SSSP-HighLevel($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*, $s \in \mathcal{N}$):

$path, cost := [n \mapsto ? \mid n \in \mathcal{N}], [n \mapsto \infty \mid n \in \mathcal{N}]$.
$path[s], cost[s] := s, 0$.
while $(m, n) \in \mathcal{E}$ with $cost[m] + weight((m, n)) < cost[n]$ **do**
    $path[n], cost[n] := m, weight((m, n)) + cost[m]$.
return $path, cost$.



n_0:	?	∞
n_1:	?	∞
n_2:	?	∞
A:	A	0
n_4:	?	∞
n_5:	?	∞
n_6:	?	∞
n_7:	?	∞
n_8:	?	∞
n_9:	?	∞

A single-source shortest path algorithm

Algorithm $\textsc{SSSP-HighLevel}(\mathcal{G} = (\mathcal{N}, \mathcal{E}), \mathit{weight}, s \in \mathcal{N})$:

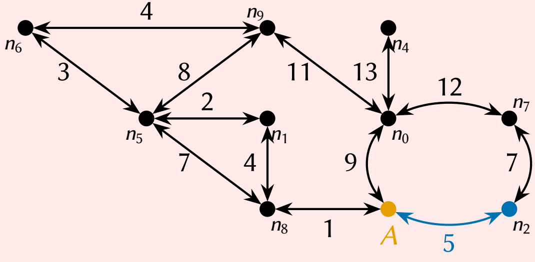
$\mathit{path}, \mathit{cost} := [n \mapsto ? \mid n \in \mathcal{N}], [n \mapsto \infty \mid n \in \mathcal{N}]$.

$\mathit{path}[s], \mathit{cost}[s] := s, 0$.

while $(m, n) \in \mathcal{E}$ with $\mathit{cost}[m] + \mathit{weight}((m, n)) < \mathit{cost}[n]$ **do**

  $\mathit{path}[n], \mathit{cost}[n] := m, \mathit{weight}((m, n)) + \mathit{cost}[m]$.

return $\mathit{path}, \mathit{cost}$.



n_0:	?	∞
n_1:	?	∞
n_2:	?	∞
A:	A	0
n_4:	?	∞
n_5:	?	∞
n_6:	?	∞
n_7:	?	∞
n_8:	?	∞
n_9:	?	∞

A single-source shortest path algorithm

Algorithm SSSP-HighLevel($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*, $s \in \mathcal{N}$):

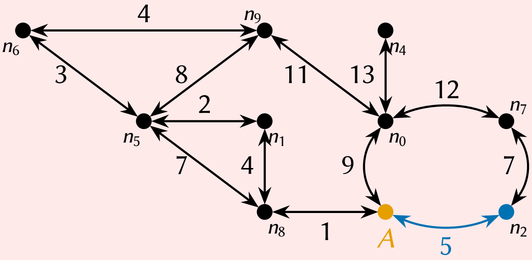
$path, cost := [n \mapsto ? \mid n \in \mathcal{N}], [n \mapsto \infty \mid n \in \mathcal{N}]$.

$path[s], cost[s] := s, 0$.

while $(m, n) \in \mathcal{E}$ with $cost[m] + weight((m, n)) < cost[n]$ **do**

  $path[n], cost[n] := m, weight((m, n)) + cost[m]$.

return $path, cost$.



n_0:	?	∞
n_1:	?	∞
n_2:	A	5
A:	A	0
n_4:	?	∞
n_5:	?	∞
n_6:	?	∞
n_7:	?	∞
n_8:	?	∞
n_9:	?	∞

A single-source shortest path algorithm

Algorithm SSSP-HighLevel($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*, $s \in \mathcal{N}$):

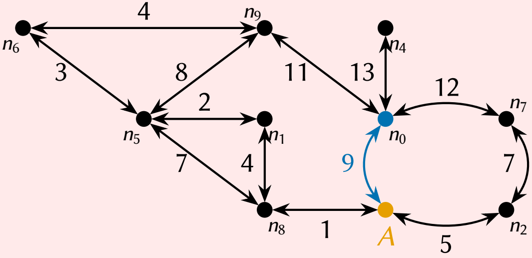
$path, cost := [n \mapsto ? \mid n \in \mathcal{N}], [n \mapsto \infty \mid n \in \mathcal{N}]$.

$path[s], cost[s] := s, 0$.

while $(m, n) \in \mathcal{E}$ with $cost[m] + weight((m, n)) < cost[n]$ **do**

  $path[n], cost[n] := m, weight((m, n)) + cost[m]$.

return $path, cost$.

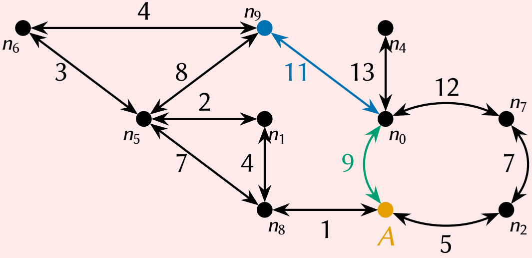


n_0:	A	9
n_1:	?	∞
n_2:	A	5
A:	A	0
n_4:	?	∞
n_5:	?	∞
n_6:	?	∞
n_7:	?	∞
n_8:	?	∞
n_9:	?	∞

A single-source shortest path algorithm

Algorithm SSSP-HighLevel($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $weight$, $s \in \mathcal{N}$)**:**

$path, cost := [n \mapsto ? \mid n \in \mathcal{N}], [n \mapsto \infty \mid n \in \mathcal{N}]$.
$path[s], cost[s] := s, 0$.
while $(m, n) \in \mathcal{E}$ with $cost[m] + weight((m, n)) < cost[n]$ **do**
  $path[n], cost[n] := m, weight((m, n)) + cost[m]$.
return $path, cost$.



n_0:	A	9
n_1:	?	∞
n_2:	A	5
A:	A	0
n_4:	?	∞
n_5:	?	∞
n_6:	?	∞
n_7:	?	∞
n_8:	?	∞
n_9:	n_0	20

A single-source shortest path algorithm

Algorithm $\textsc{SSSP-HighLevel}(\mathcal{G} = (\mathcal{N}, \mathcal{E}), \mathit{weight}, s \in \mathcal{N})$:

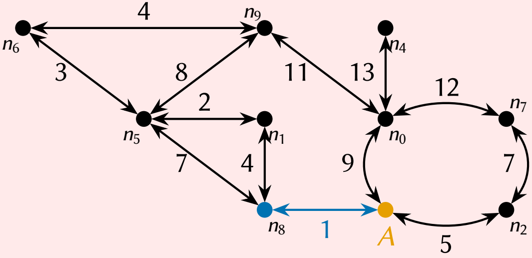
$\mathit{path}, \mathit{cost} := [n \mapsto ?\mid n \in \mathcal{N}], [n \mapsto \infty \mid n \in \mathcal{N}]$.

$\mathit{path}[s], \mathit{cost}[s] := s, 0$.

while $(m, n) \in \mathcal{E}$ with $\mathit{cost}[m] + \mathit{weight}((m, n)) < \mathit{cost}[n]$ **do**

  $\mathit{path}[n], \mathit{cost}[n] := m, \mathit{weight}((m, n)) + \mathit{cost}[m]$.

return $\mathit{path}, \mathit{cost}$.

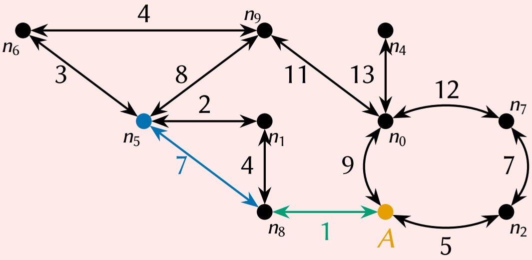


n_0:	A	9
n_1:	?	∞
n_2:	A	5
A:	A	0
n_4:	?	∞
n_5:	?	∞
n_6:	?	∞
n_7:	?	∞
n_8:	A	1
n_9:	n_0	20

A single-source shortest path algorithm

Algorithm SSSP-HighLevel($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $weight$, $s \in \mathcal{N}$):

$path, cost := [n \mapsto ? \mid n \in \mathcal{N}], [n \mapsto \infty \mid n \in \mathcal{N}]$.
$path[s], cost[s] := s, 0$.
while $(m, n) \in \mathcal{E}$ with $cost[m] + weight((m, n)) < cost[n]$ **do**
    $path[n], cost[n] := m, weight((m, n)) + cost[m]$.
return $path, cost$.



n_0:	A	9
n_1:	?	∞
n_2:	A	5
A:	A	0
n_4:	?	∞
n_5:	n_8	8
n_6:	?	∞
n_7:	?	∞
n_8:	A	1
n_9:	n_0	20

A single-source shortest path algorithm

Algorithm SSSP-HIGHLEVEL($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*, $s \in \mathcal{N}$):

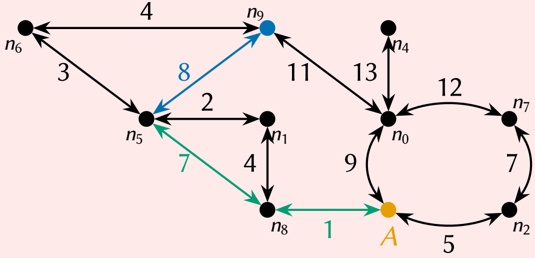
$path, cost := [n \mapsto ? \mid n \in \mathcal{N}], [n \mapsto \infty \mid n \in \mathcal{N}]$.

$path[s], cost[s] := s, 0$.

while $(m, n) \in \mathcal{E}$ with $cost[m] + weight((m, n)) < cost[n]$ **do**

  $path[n], cost[n] := m, weight((m, n)) + cost[m]$.

return $path, cost$.

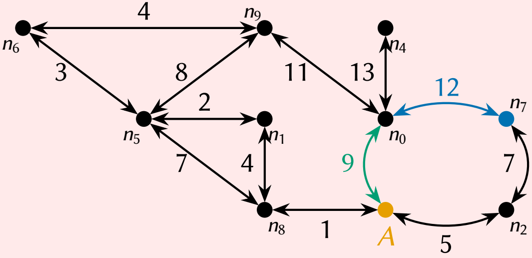


n_0:	A	9
n_1:	?	∞
n_2:	A	5
A:	A	0
n_4:	?	∞
n_5:	n_8	8
n_6:	?	∞
n_7:	?	∞
n_8:	A	1
n_9:	n_5	16

A single-source shortest path algorithm

Algorithm $\textsc{SSSP-HighLevel}(\mathcal{G} = (\mathcal{N}, \mathcal{E}), \mathit{weight}, s \in \mathcal{N})$**:**

```
path, cost := [n ↦ ? | n ∈ 𝒩], [n ↦ ∞ | n ∈ 𝒩].
path[s], cost[s] := s, 0.
while (m, n) ∈ ℰ with cost[m] + weight((m, n)) < cost[n] do
    path[n], cost[n] := m, weight((m, n)) + cost[m].
return path, cost.
```

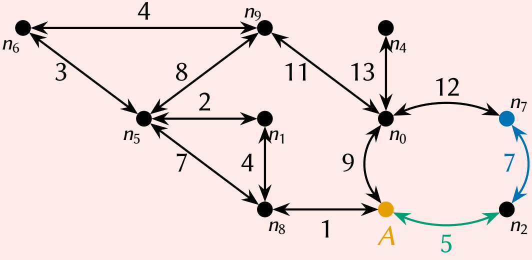


n_0:	A	9
n_1:	?	∞
n_2:	A	5
A:	A	0
n_4:	?	∞
n_5:	n_8	8
n_6:	?	∞
n_7:	n_0	21
n_8:	A	1
n_9:	n_5	16

A single-source shortest path algorithm

Algorithm SSSP-HighLevel($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $weight$, $s \in \mathcal{N}$):

$path, cost := [n \mapsto ? \mid n \in \mathcal{N}], [n \mapsto \infty \mid n \in \mathcal{N}]$.
$path[s], cost[s] := s, 0$.
while $(m, n) \in \mathcal{E}$ with $cost[m] + weight((m, n)) < cost[n]$ **do**
    $path[n], cost[n] := m, weight((m, n)) + cost[m]$.
return $path, cost$.



n_0:	A	9
n_1:	?	∞
n_2:	A	5
A:	A	0
n_4:	?	∞
n_5:	n_8	8
n_6:	?	∞
n_7:	n_2	12
n_8:	A	1
n_9:	n_5	16

A single-source shortest path algorithm

Algorithm SSSP-HighLevel($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*, $s \in \mathcal{N}$):

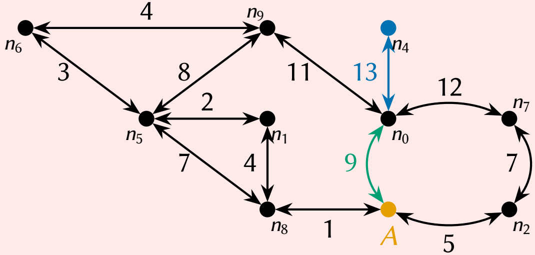
$path, cost := [n \mapsto ? \mid n \in \mathcal{N}], [n \mapsto \infty \mid n \in \mathcal{N}]$.

$path[s], cost[s] := s, 0$.

while $(m, n) \in \mathcal{E}$ with $cost[m] + weight((m, n)) < cost[n]$ **do**

$\quad path[n], cost[n] := m, weight((m, n)) + cost[m]$.

return $path, cost$.

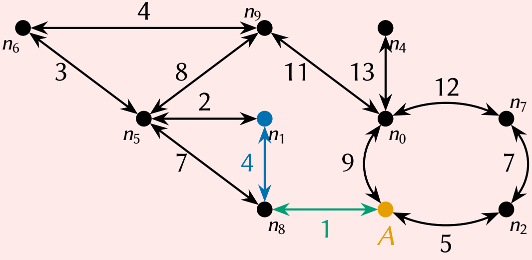


n_0:	A	9
n_1:	?	∞
n_2:	A	5
A:	A	0
n_4:	n_0	22
n_5:	n_8	8
n_6:	?	∞
n_7:	n_2	12
n_8:	A	1
n_9:	n_5	16

A single-source shortest path algorithm

Algorithm SSSP-HighLevel$(\mathcal{G} = (\mathcal{N}, \mathcal{E}), weight, s \in \mathcal{N})$:

$path, cost := [n \mapsto ? \mid n \in \mathcal{N}], [n \mapsto \infty \mid n \in \mathcal{N}]$.
$path[s], cost[s] := s, 0$.
while $(m, n) \in \mathcal{E}$ with $cost[m] + weight((m, n)) < cost[n]$ **do**
    $path[n], cost[n] := m, weight((m, n)) + cost[m]$.
return $path, cost$.

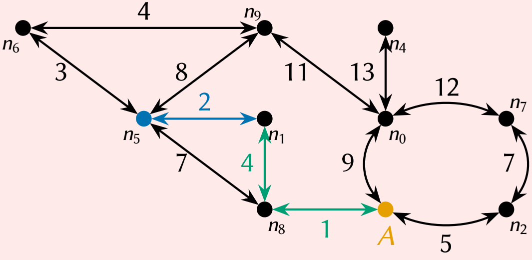


n_0:	A	9
n_1:	n_8	5
n_2:	A	5
A:	A	0
n_4:	n_0	22
n_5:	n_8	8
n_6:	?	∞
n_7:	n_2	12
n_8:	A	1
n_9:	n_5	16

A single-source shortest path algorithm

Algorithm SSSP-HighLevel($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*, $s \in \mathcal{N}$):

$path, cost := [n \mapsto ? \mid n \in \mathcal{N}], [n \mapsto \infty \mid n \in \mathcal{N}]$.
$path[s], cost[s] := s, 0$.
while $(m, n) \in \mathcal{E}$ with $cost[m] + weight((m, n)) < cost[n]$ **do**
  $path[n], cost[n] := m, weight((m, n)) + cost[m]$.
return $path, cost$.

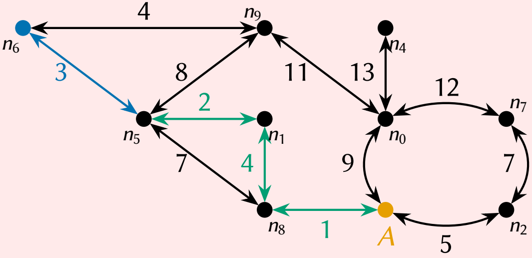


n_0:	A	9
n_1:	n_8	5
n_2:	A	5
A:	A	0
n_4:	n_0	22
n_5:	n_1	7
n_6:	?	∞
n_7:	n_2	12
n_8:	A	1
n_9:	n_5	16

A single-source shortest path algorithm

Algorithm SSSP-HighLevel($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*, $s \in \mathcal{N}$):

$path, cost := [n \mapsto ? \mid n \in \mathcal{N}], [n \mapsto \infty \mid n \in \mathcal{N}]$.
$path[s], cost[s] := s, 0$.
while $(m, n) \in \mathcal{E}$ with $cost[m] + weight((m, n)) < cost[n]$ **do**
  $path[n], cost[n] := m, weight((m, n)) + cost[m]$.
return $path, cost$.

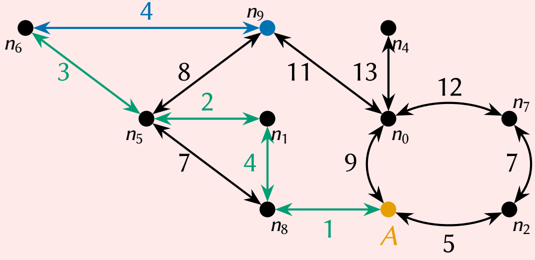


n_0:	A	9
n_1:	n_8	5
n_2:	A	5
A:	A	0
n_4:	n_0	22
n_5:	n_1	7
n_6:	n_5	10
n_7:	n_2	12
n_8:	A	1
n_9:	n_5	16

A single-source shortest path algorithm

Algorithm SSSP-HighLevel($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*, $s \in \mathcal{N}$)**:**

$\quad$ $path, cost := [n \mapsto ? \mid n \in \mathcal{N}], [n \mapsto \infty \mid n \in \mathcal{N}]$.
$\quad$ $path[s], cost[s] := s, 0$.
$\quad$ **while** $(m, n) \in \mathcal{E}$ with $cost[m] + weight((m, n)) < cost[n]$ **do**
$\quad\quad$ $path[n], cost[n] := m, weight((m, n)) + cost[m]$.
$\quad$ **return** $path, cost$.



n_0:	A	9
n_1:	n_8	5
n_2:	A	5
A:	A	0
n_4:	n_0	22
n_5:	n_1	7
n_6:	n_5	10
n_7:	n_2	12
n_8:	A	1
n_9:	n_6	14

A single-source shortest path algorithm

Algorithm $\textsc{SSSP-HighLevel}(\mathcal{G} = (\mathcal{N}, \mathcal{E}), \mathit{weight}, s \in \mathcal{N})$**:**

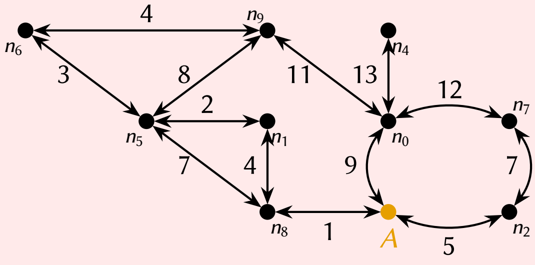
$\mathit{path}, \mathit{cost} := [n \mapsto ? \mid n \in \mathcal{N}], [n \mapsto \infty \mid n \in \mathcal{N}]$.

$\mathit{path}[s], \mathit{cost}[s] := s, 0$.

while $(m, n) \in \mathcal{E}$ with $\mathit{cost}[m] + \mathit{weight}((m, n)) < \mathit{cost}[n]$ **do**

  $\mathit{path}[n], \mathit{cost}[n] := m, \mathit{weight}((m, n)) + \mathit{cost}[m]$.

return $\mathit{path}, \mathit{cost}$.



n_0:	A	9
n_1:	n_8	5
n_2:	A	5
A:	A	0
n_4:	n_0	22
n_5:	n_1	7
n_6:	n_5	10
n_7:	n_2	12
n_8:	A	1
n_9:	n_5	16

A single-source shortest path algorithm

Algorithm SSSP-HighLevel($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*, $s \in \mathcal{N}$)**:**

$\mathit{path}, \mathit{cost} := [n \mapsto ?\ |\ n \in \mathcal{N}], [n \mapsto \infty\ |\ n \in \mathcal{N}]$.
$\mathit{path}[s], \mathit{cost}[s] := s, 0$.
while $(m, n) \in \mathcal{E}$ with $\mathit{cost}[m] + \mathit{weight}((m, n)) < \mathit{cost}[n]$ **do**
    $\mathit{path}[n], \mathit{cost}[n] := m, \mathit{weight}((m, n)) + \mathit{cost}[m]$.
return $\mathit{path}, \mathit{cost}$.

Theorem

Algorithm SSSP-HighLevel is correct.

A single-source shortest path algorithm

Algorithm SSSP-HighLevel($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*, $s \in \mathcal{N}$)**:**

$path, cost := [n \mapsto ? \mid n \in \mathcal{N}], [n \mapsto \infty \mid n \in \mathcal{N}]$.
$path[s], cost[s] := s, 0$.
while $(m, n) \in \mathcal{E}$ with $cost[m] + weight((m, n)) < cost[n]$ **do**
    $path[n], cost[n] := m, weight((m, n)) + cost[m]$.
return $path, cost$.

Theorem

Let cost be the cost of the paths from s represented by path.
We say that $(m, n) \in \mathcal{E}$ is eligible if $cost[m] + weight((m, n)) < cost[n]$.
The values in cost are the costs of the shortest paths from s if no edges are eligible.

A single-source shortest path algorithm

Theorem

Let cost be the cost of the paths from s represented by path.
We say that $(m, n) \in \mathcal{E}$ is eligible if $cost[m] + weight((m, n)) < cost[n]$.
The values in cost are the costs of the shortest paths from s if no edges are eligible.

Observation

If there is an *eligible edge* (m, n), then we have *certainly not* found all shortest paths!

A single-source shortest path algorithm

Theorem

Let cost be the cost of the paths from s represented by path.
We say that $(m, n) \in \mathcal{E}$ is eligible if $cost[m] + weight((m, n)) < cost[n]$.
The values in cost are the costs of the shortest paths from s if no edges are eligible.

Proof

Assume no edges are eligible and that there is a shortest path $sn_1 \ldots n_i t$.

We have to prove $cost[t] = weight((s, n_1)) + weight((n_1, n_2)) + \cdots + weight((n_i, t))$.

A single-source shortest path algorithm

Theorem

Let cost be the cost of the paths from s represented by path.
We say that $(m, n) \in \mathcal{E}$ is eligible if $cost[m] + weight((m, n)) < cost[n]$.
The values in cost are the costs of the shortest paths from s if no edges are eligible.

Proof

Assume no edges are eligible and that there is a shortest path $sn_1 \ldots n_i t$.

We have to prove $cost[t] = weight((s, n_1)) + weight((n_1, n_2)) + \cdots + weight((n_i, t))$.
We have

$$cost[t] \leq cost[n_i] + weight((n_i, t))$$

A single-source shortest path algorithm

Theorem

Let cost be the cost of the paths from s represented by path.
We say that $(m, n) \in \mathcal{E}$ *is eligible if* $cost[m] + weight((m, n)) < cost[n]$.
The values in cost are the costs of the shortest paths from s if no edges are eligible.

Proof

Assume no edges are eligible and that there is a shortest path $sn_1 \ldots n_i t$.

We have to prove $cost[t] = weight((s, n_1)) + weight((n_1, n_2)) + \cdots + weight((n_i, t))$.
We have

$$cost[t] \leq cost[n_i] + weight((n_i, t))$$

A single-source shortest path algorithm

Theorem

Let cost be the cost of the paths from s represented by path.
We say that $(m, n) \in \mathcal{E}$ is eligible if $cost[m] + weight((m, n)) < cost[n]$.
The values in cost are the costs of the shortest paths from s if no edges are eligible.

Proof

Assume no edges are eligible and that there is a shortest path $sn_1 \dots n_i t$.

We have to prove $cost[t] = weight((s, n_1)) + weight((n_1, n_2)) + \cdots + weight((n_i, t))$.
We have

$$\begin{aligned} cost[t] &\le cost[n_i] + weight((n_i, t)) \\ &\le cost[n_{i-1}] + weight((n_{i-1}, n_i)) + weight((n_i, t)) \end{aligned}$$

A single-source shortest path algorithm

Theorem

Let cost be the cost of the paths from s represented by path.
We say that $(m, n) \in \mathcal{E}$ is eligible if $cost[m] + weight((m, n)) < cost[n]$.
The values in cost are the costs of the shortest paths from s if no edges are eligible.

Proof

Assume no edges are eligible and that there is a shortest path $sn_1 \dots n_i t$.

We have to prove $cost[t] = weight((s, n_1)) + weight((n_1, n_2)) + \cdots + weight((n_i, t))$.
We have

$$\begin{aligned} cost[t] &\leq cost[n_i] + weight((n_i, t)) \\ &\leq cost[n_{i-1}] + weight((n_{i-1}, n_i)) + weight((n_i, t)) \\ &\leq weight((s, n_1)) + \cdots + weight((n_{i-1}, n_i)) + weight((n_i, t)). \end{aligned}$$

A single-source shortest path algorithm

Theorem

Let cost be the cost of the paths from s represented by path.
We say that $(m, n) \in \mathcal{E}$ is eligible if $cost[m] + weight((m, n)) < cost[n]$.
The values in cost are the costs of the shortest paths from s if no edges are eligible.

Proof

Assume no edges are eligible and that there is a shortest path $sn_1 \ldots n_i t$.

We have to prove $cost[t] = weight((s, n_1)) + weight((n_1, n_2)) + \cdots + weight((n_i, t))$.
We have

$$cost[t] \leq weight((s, n_1)) + \cdots + weight((n_{i-1}, n_i)) + weight((n_i, t)).$$

As $cost[t]$ is the cost of a path from s to t and $sn_1 \ldots n_i t$ is the shortest path, we also have:

$$weight((s, n_1)) + \cdots + weight((n_{i-1}, n_i)) + weight((n_i, t)) \leq cost[t].$$

A single-source shortest path algorithm

Algorithm SSSP-HighLevel($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*, $s \in \mathcal{N}$)**:**

1: $\mathit{path}, \mathit{cost} := [n \mapsto ? \mid n \in \mathcal{N}], [n \mapsto \infty \mid n \in \mathcal{N}]$.
2: $\mathit{path}[s], \mathit{cost}[s] := s, 0$.
3: **while** $(m, n) \in \mathcal{E}$ with $\mathit{cost}[m] + \mathit{weight}((m, n)) < \mathit{cost}[n]$ **do**
4: $\quad \mathit{path}[n], \mathit{cost}[n] := m, \mathit{weight}((m, n)) + \mathit{cost}[m]$.
5: **return** $\mathit{path}, \mathit{cost}$.

How to find eligible edges $(m, n) \in \mathcal{E}$?

Efficient shortest path algorithms depend on a good method to explorer eligible edges: e.g., we want to prevent revisiting the same edge multiple times.

Dijkstra's shortest-path algorithm

Algorithm SSSP-HighLevel($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*, $s \in \mathcal{N}$):

$\mathit{path}, \mathit{cost} := [n \mapsto ? \mid n \in \mathcal{N}], [n \mapsto \infty \mid n \in \mathcal{N}]$.
$\mathit{path}[s], \mathit{cost}[s] := s, 0$.
while $(m, n) \in \mathcal{E}$ with $\mathit{cost}[m] + \mathit{weight}((m, n)) < \mathit{cost}[n]$ **do**
    $\mathit{path}[n], \mathit{cost}[n] := m, \mathit{weight}((m, n)) + \mathit{cost}[m]$.
return $\mathit{path}, \mathit{cost}$.

Algorithm SSSP-Dijkstra($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*, $s \in \mathcal{N}$):

$\mathit{path}, \mathit{cost} := [n \mapsto ? \mid n \in \mathcal{N}], [n \mapsto \infty \mid n \in \mathcal{N}]$.
$\mathit{path}[s], \mathit{cost}[s] := s, 0$.
for all nodes m on increasing shortest-path distance to s **do**
    for all edges $(m, n) \in \mathcal{E}$ **do**
        if $\mathit{cost}[m] + \mathit{weight}((m, n)) < \mathit{cost}[n]$ **then**
            $\mathit{path}[n], \mathit{cost}[n] := m, \mathit{weight}((m, n)) + \mathit{cost}[m]$.
return $\mathit{path}, \mathit{cost}$.

Dijkstra's shortest-path algorithm

We *disallow* negative edge weights.

Idea

Use a priority queue to process nodes in increasing best-known distance to s.

Algorithm SSSP-DIJKSTRA($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*, $s \in \mathcal{N}$)**:**

$\mathit{path}, \mathit{cost} := [n \mapsto ? \mid n \in \mathcal{N}], [n \mapsto \infty \mid n \in \mathcal{N}]$.
$\mathit{path}[s], \mathit{cost}[s] := s, 0$.
for all nodes m on increasing shortest-path distance to s **do**
  for all edges $(m, n) \in \mathcal{E}$ **do**
    if $\mathit{cost}[m] + \mathit{weight}((m, n)) < \mathit{cost}[n]$ **then**
      $\mathit{path}[n], \mathit{cost}[n] := m, \mathit{weight}((m, n)) + \mathit{cost}[m]$.
return $\mathit{path}, \mathit{cost}$.

Dijkstra's shortest-path algorithm

We *disallow* negative edge weights.

Idea

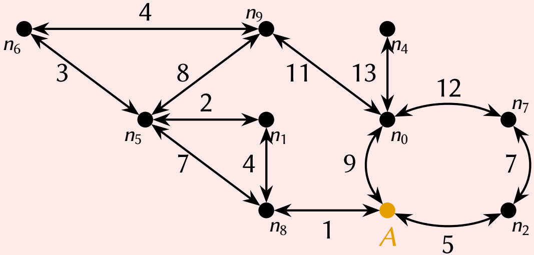
Use a priority queue to process nodes in increasing best-known distance to s.

Algorithm $\textsc{SSSP-Dijkstra}(\mathcal{G} = (\mathcal{N}, \mathcal{E}), \mathit{weight}, s \in \mathcal{N})$**:**

$\mathit{path}, \mathit{cost} := [n \mapsto ? \mid n \in \mathcal{N}], [n \mapsto \infty \mid n \in \mathcal{N}]$.
$\mathit{path}[s], \mathit{cost}[s] := s, 0$.
Q := a minimum-priority queue that holds node s with priority 0.
while $Q \neq \emptyset$ **do**
    Remove node m with lowest priority from Q.
    for all edges $(m, n) \in \mathcal{E}$ **do**
        if $\mathit{cost}[m] + \mathit{weight}((m, n)) < \mathit{cost}[n]$ **then**
            $\mathit{path}[n], \mathit{cost}[n] := m, \mathit{weight}((m, n)) + \mathit{cost}[m]$.
            Update n in Q such that n has priority $\mathit{cost}[n]$ in Q.
return $\mathit{path}, \mathit{cost}$.

Dijkstra's shortest-path algorithm

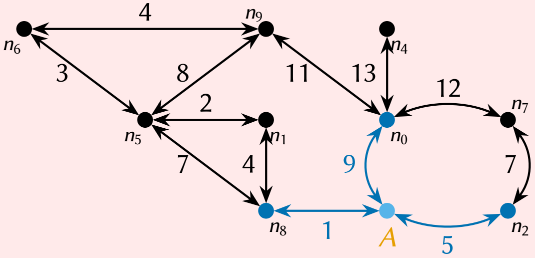
$Q := \{(A : 0)\}$.



n_0:	?	∞
n_1:	?	∞
n_2:	?	∞
A:	A	0
n_4:	?	∞
n_5:	?	∞
n_6:	?	∞
n_7:	?	∞
n_8:	?	∞
n_9:	?	∞

Dijkstra's shortest-path algorithm

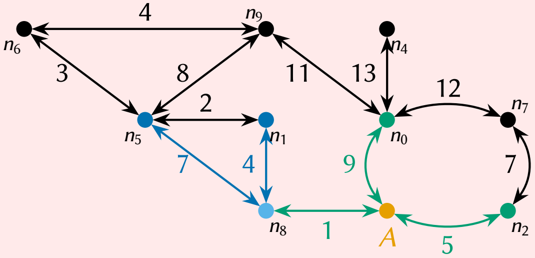
$Q := \{(n_8 : 1), (n_2 : 5), (n_0 : 9)\}.$



n_0:	A	9
n_1:	?	∞
n_2:	A	5
A:	A	0
n_4:	?	∞
n_5:	?	∞
n_6:	?	∞
n_7:	?	∞
n_8:	A	1
n_9:	?	∞

Dijkstra's shortest-path algorithm

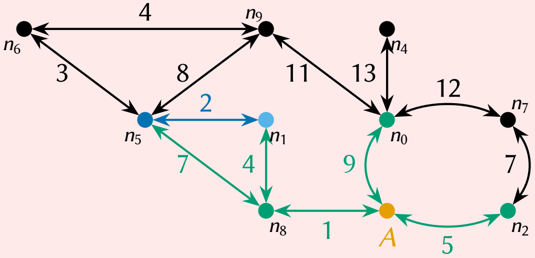
$$Q := \{(n_1 : 5), (n_2 : 5), (n_5 : 8), (n_0 : 9)\}.$$



n_0:	A	9
n_1:	n_8	5
n_2:	A	5
A:	A	0
n_4:	?	∞
n_5:	n_8	8
n_6:	?	∞
n_7:	?	∞
n_8:	A	1
n_9:	?	∞

Dijkstra's shortest-path algorithm

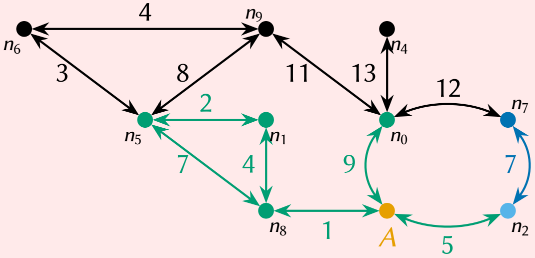
$Q := \{(n_2 : 5), (n_5 : 7), (n_0 : 9)\}.$



n_0:	A	9
n_1:	n_8	5
n_2:	A	5
A:	A	0
n_4:	?	∞
n_5:	n_1	7
n_6:	?	∞
n_7:	?	∞
n_8:	A	1
n_9:	?	∞

Dijkstra's shortest-path algorithm

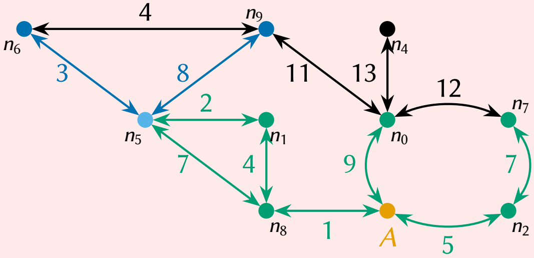
$Q := \{(n_5 : 7), (n_0 : 9), (n_7 : 12)\}.$



n_0:	A	9
n_1:	n_8	5
n_2:	A	5
A:	A	0
n_4:	?	∞
n_5:	n_1	7
n_6:	?	∞
n_7:	n_2	12
n_8:	A	1
n_9:	?	∞

Dijkstra's shortest-path algorithm

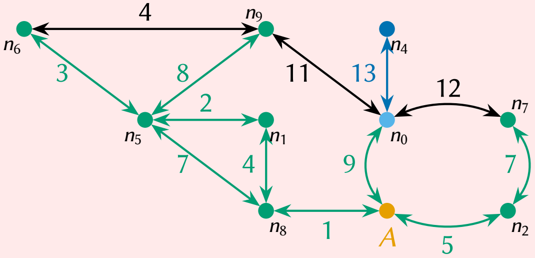
$$Q := \{(n_0 : 9), (n_6 : 10), (n_7 : 12), (n_9 : 15)\}.$$



n_0:	A	9
n_1:	n_8	5
n_2:	A	5
A:	A	0
n_4:	?	∞
n_5:	n_1	7
n_6:	n_5	10
n_7:	n_2	12
n_8:	A	1
n_9:	n_5	15

Dijkstra's shortest-path algorithm

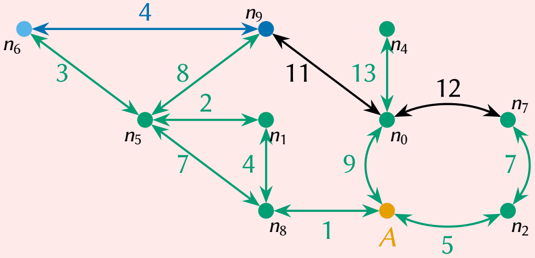
$Q := \{(n_6 : 10), (n_7 : 12), (n_9 : 15), (n_4 : 22)\}.$



n_0:	A	9
n_1:	n_8	5
n_2:	A	5
A:	A	0
n_4:	n_0	22
n_5:	n_1	7
n_6:	n_5	10
n_7:	n_2	12
n_8:	A	1
n_9:	n_5	15

Dijkstra's shortest-path algorithm

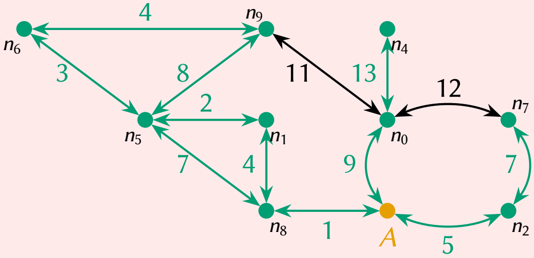
$Q := \{(n_7 : 12), (n_9 : 14), (n_4 : 22)\}.$



n_0:	A	9
n_1:	n_8	5
n_2:	A	5
A:	A	0
n_4:	n_0	22
n_5:	n_1	7
n_6:	n_5	10
n_7:	n_2	12
n_8:	A	1
n_9:	n_6	14

Dijkstra's shortest-path algorithm

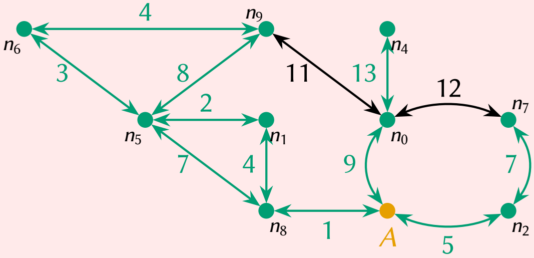
$Q := \{(n_9 : 15), (n_4 : 22)\}.$



n_0:	A	9
n_1:	n_8	5
n_2:	A	5
A:	A	0
n_4:	n_0	22
n_5:	n_1	7
n_6:	n_5	10
n_7:	n_2	12
n_8:	A	1
n_9:	n_6	14

Dijkstra's shortest-path algorithm

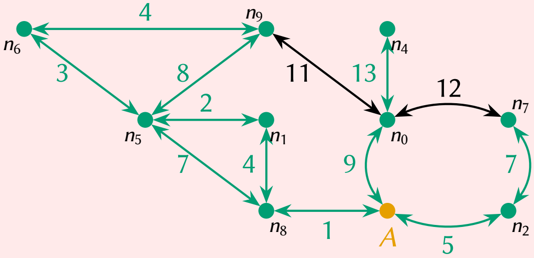
$Q := \{(n_4 : 22)\}.$



n_0:	A	9
n_1:	n_8	5
n_2:	A	5
A:	A	0
n_4:	n_0	22
n_5:	n_1	7
n_6:	n_5	10
n_7:	n_2	12
n_8:	A	1
n_9:	n_6	14

Dijkstra's shortest-path algorithm

$Q := \{\}$.



n_0:	A	9
n_1:	n_8	5
n_2:	A	5
A:	A	0
n_4:	n_0	22
n_5:	n_1	7
n_6:	n_5	10
n_7:	n_2	12
n_8:	A	1
n_9:	n_6	14

Dijkstra's shortest-path algorithm

Algorithm SSSP-DIJKSTRA($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $weight$, $s \in \mathcal{N}$):

$path, cost := [n \mapsto ? \mid n \in \mathcal{N}], [n \mapsto \infty \mid n \in \mathcal{N}]$.
$path[s], cost[s] := s, 0$.
Q := a minimum-priority queue that holds node s with priority 0.
while $Q \neq \emptyset$ **do**
    Remove node m with lowest priority from Q.
    for all edges $(m, n) \in \mathcal{E}$ **do**
        if $cost[m] + weight((m, n)) < cost[n]$ **then**
            $path[n], cost[n] := m, weight((m, n)) + cost[m]$.
            Update n in Q such that n has priority $cost[n]$ in Q.
return $path, cost$.

Complexity: Manage Q in the same way as in Prim's Algorithm.

Dijkstra's shortest-path algorithm

Algorithm SSSP-Dijkstra($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*, $s \in \mathcal{N}$):

$\mathit{path}, \mathit{cost} := [n \mapsto ?\ |\ n \in \mathcal{N}], [n \mapsto \infty\ |\ n \in \mathcal{N}]$.
$\mathit{path}[s], \mathit{cost}[s] := s, 0$.
Q := a minimum-priority queue that holds node s with priority 0.
while $Q \neq \emptyset$ **do**
  Remove node m with lowest priority from Q.
  for all edges $(m, n) \in \mathcal{E}$ **do**
    if $\mathit{cost}[m] + \mathit{weight}((m, n)) < \mathit{cost}[n]$ **then**
      $\mathit{path}[n], \mathit{cost}[n] := m, \mathit{weight}((m, n)) + \mathit{cost}[m]$.
      Update n in Q such that n has priority $\mathit{cost}[n]$ in Q.
return $\mathit{path}, \mathit{cost}$.

Complexity: Manage Q in the same way as in Prim's Algorithm.

- $\Theta(|\mathcal{E}| \log(|\mathcal{N}|))$.

Dijkstra's shortest-path algorithm

Algorithm SSSP-DIJKSTRA($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $weight$, $s \in \mathcal{N}$):

$path, cost := [n \mapsto ?\ |\ n \in \mathcal{N}], [n \mapsto \infty\ |\ n \in \mathcal{N}]$.
$path[s], cost[s] := s, 0$.
Q := a minimum-priority queue that holds node s with priority 0.
while $Q \neq \emptyset$ **do**
    Remove node m with lowest priority from Q.
    for all edges $(m, n) \in \mathcal{E}$ **do**
        if $cost[m] + weight((m, n)) < cost[n]$ **then**
            $path[n], cost[n] := m, weight((m, n)) + cost[m]$.
            Update n in Q such that n has priority $cost[n]$ in Q.
return $path, cost$.

Complexity: Manage Q in the same way as in Prim's Algorithm.

- $\Theta(|\mathcal{E}| \log(|\mathcal{N}|))$.
- $\Theta(|\mathcal{N}| \log(|\mathcal{N}|) + |\mathcal{E}|)$ with a Fibonacci Heap.

Dijkstra's shortest-path algorithm

Algorithm SSSP-DIJKSTRA($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*, $s \in \mathcal{N}$)**:**

$\mathit{path}, \mathit{cost} := [n \mapsto ?\ |\ n \in \mathcal{N}], [n \mapsto \infty\ |\ n \in \mathcal{N}]$.
$\mathit{path}[s], \mathit{cost}[s] := s, 0$.
Q := a minimum-priority queue that holds node s with priority 0.
while $Q \neq \emptyset$ **do**
  Remove node m with lowest priority from Q.
  for all edges $(m, n) \in \mathcal{E}$ **do**
    if $\mathit{cost}[m] + \mathit{weight}((m, n)) < \mathit{cost}[n]$ **then**
      $\mathit{path}[n], \mathit{cost}[n] := m, \mathit{weight}((m, n)) + \mathit{cost}[m]$.
      Update n in Q such that n has priority $\mathit{cost}[n]$ in Q.
return $\mathit{path}, \mathit{cost}$.

Correctness

Dijkstra's shortest-path algorithm

Algorithm SSSP-DIJKSTRA($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, weight, $s \in \mathcal{N}$):

$\mathit{path}, \mathit{cost} := [n \mapsto ? \mid n \in \mathcal{N}], [n \mapsto \infty \mid n \in \mathcal{N}]$.
$\mathit{path}[s], \mathit{cost}[s] := s, 0$.
$Q :=$ a minimum-priority queue that holds node s with priority 0.
while $Q \neq \emptyset$ **do**
  Remove node m with lowest priority from Q.
  for all edges $(m, n) \in \mathcal{E}$ **do**
    if $\mathit{cost}[m] + \mathit{weight}((m, n)) < \mathit{cost}[n]$ **then**
      $\mathit{path}[n], \mathit{cost}[n] := m, \mathit{weight}((m, n)) + \mathit{cost}[m]$.
      Update n in Q such that n has priority $\mathit{cost}[n]$ in Q.
return $\mathit{path}, \mathit{cost}$.

Correctness: for every node t

- *path* represents a shortest path from s to t of cost $\mathit{cost}[t]$; or
- there exists a node $u \in Q$ with priority $\mathit{cost}[u]$ such that a shortest path from s to t goes through u and the shortest path from s to u has cost $\mathit{cost}[u]$.

Variants of the shortest-path problem

- *The all-pairs shortest path problem*:
  Give a directed edge-weighted graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$,
  find *a* shortest path (if any) between every pair of nodes $(s, t) \in \mathcal{N} \times \mathcal{N}$.

- *The single-sink shortest path problem*:
  Give a directed edge-weighted graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ and sink $t \in \mathcal{N}$
  find *a* shortest path (if any) from any node $s \in \mathcal{N}$ to t.

- Shortest paths on *undirected weighted graphs*:

Variants of the shortest-path problem

- *The all-pairs shortest path problem*:
  Give a directed edge-weighted graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$,
  find *a* shortest path (if any) between every pair of nodes $(s, t) \in \mathcal{N} \times \mathcal{N}$.
  Solution: Run SSSP-Dijkstra for each node.

- *The single-sink shortest path problem*:
  Give a directed edge-weighted graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ and sink $t \in \mathcal{N}$
  find *a* shortest path (if any) from any node $s \in \mathcal{N}$ to t.
  Solution: Reverse edges in $\mathcal{G}$ and run SSSP-Dijkstra with source t.

- Shortest paths on *undirected weighted graphs*:
  Solution: interpret each undirected edge as *two* directed edges.

Variants of the shortest-path problem

Problem: The source-sink shortest path problem

Give a directed edge-weighted graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ and source and target nodes $s, t \in \mathcal{N}$, find *a* shortest path (if any) from s to t.

Variants of the shortest-path problem

Problem: The source-sink shortest path problem

Give a directed edge-weighted graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ and source and target nodes $s, t \in \mathcal{N}$, find *a* shortest path (if any) from s to t.

- We can run SSSP-Dijkstra to find the path from s to t, and stop as soon as we remove t from the queue Q.

Variants of the shortest-path problem

Problem: The source-sink shortest path problem

Give a directed edge-weighted graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ and source and target nodes $s, t \in \mathcal{N}$, find *a* shortest path (if any) from s to t.

- We can run SSSP-Dijkstra to find the path from s to t, and stop as soon as we remove t from the queue Q.
- *Downside*: before removing t from Q, SSSP-Dijkstra might consider many edges that go into the "wrong direction" (away from t).

Variants of the shortest-path problem

Problem: The source-sink shortest path problem

Give a directed edge-weighted graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ and source and target nodes $s, t \in \mathcal{N}$, find *a* shortest path (if any) from s to t.

- We can run SSSP-Dijkstra to find the path from s to t, and stop as soon as we remove t from the queue Q.
- *Downside*: before removing t from Q, SSSP-Dijkstra might consider many edges that go into the "wrong direction" (away from t).
- Sometimes we can *estimate* the minimum cost of a shortest path between two nodes. For example: straight-line distance between two points on a map.

Variants of the shortest-path problem

Problem: The source-sink shortest path problem

Give a directed edge-weighted graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ and source and target nodes $s, t \in \mathcal{N}$, find *a* shortest path (if any) from s to t.

- We can run SSSP-Dijkstra to find the path from s to t, and stop as soon as we remove t from the queue Q.
- *Downside*: before removing t from Q, SSSP-Dijkstra might consider many edges that go into the "wrong direction" (away from t).
- Sometimes we can *estimate* the minimum cost of a shortest path between two nodes. For example: straight-line distance between two points on a map.
- *Optimization*: use these estimates to guide the choice of eligible edges.

Variants of the shortest-path problem

Problem: The source-sink shortest path problem

Give a directed edge-weighted graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ and source and target nodes $s, t \in \mathcal{N}$, find *a* shortest path (if any) from s to t.

- We can run SSSP-Dijkstra to find the path from s to t, and stop as soon as we remove t from the queue Q.
- *Downside*: before removing t from Q, SSSP-Dijkstra might consider many edges that go into the "wrong direction" (away from t).
- Sometimes we can *estimate* the minimum cost of a shortest path between two nodes. For example: straight-line distance between two points on a map.
- *Optimization*: use these estimates to guide the choice of eligible edges.

Such a strategy leads to the *A^* search algorithm*.

Shortest and longest paths in directed acyclic graphs

Algorithm SSSP-HighLevel($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $weight$, $s \in \mathcal{N}$)**:**

$path, cost := [n \mapsto ? \mid n \in \mathcal{N}], [n \mapsto \infty \mid n \in \mathcal{N}]$.
$path[s], cost[s] := s, 0$.
while $(m, n) \in \mathcal{E}$ with $cost[m] + weight((m, n)) < cost[n]$ **do**
  $path[n], cost[n] := m, weight((m, n)) + cost[m]$.
return $path, cost$.

Shortest and longest paths in directed acyclic graphs

Algorithm SSSP-HighLevel($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*, $s \in \mathcal{N}$)**:**

1: $path, cost := [n \mapsto ? \mid n \in \mathcal{N}], [n \mapsto \infty \mid n \in \mathcal{N}]$.
2: $path[s], cost[s] := s, 0$.
3: **while** $(m, n) \in \mathcal{E}$ with $cost[m] + weight((m, n)) < cost[n]$ **do**
4: $\quad path[n], cost[n] := m, weight((m, n)) + cost[m]$.
5: **return** $path, cost$.

Idea: process nodes in *topological order*

Shortest and longest paths in directed acyclic graphs

Idea: process nodes in *topological order*

We process node n *after* determining the shortest path from s to m for all $(m, n) \in \mathcal{E}$.

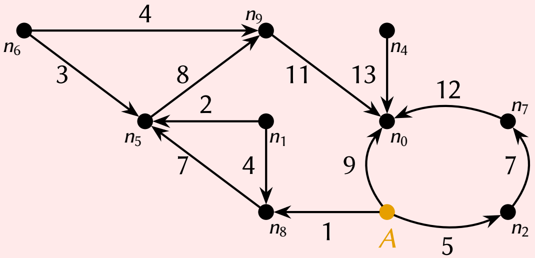
Algorithm SSSP-DAG($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*, $s \in \mathcal{N}$):

1: $path, cost := [n \mapsto ? \mid n \in \mathcal{N}], [n \mapsto \infty \mid n \in \mathcal{N}]$.
2: $path[s], cost[s] := s, 0$.
3: **for all** $n \in \mathcal{N}$ in topological order (and that follow s) **do**
4:     **for all** $(m, n) \in \mathcal{E}$ with $cost[m] + weight((m, n)) < cost[n]$ **do**
5:         $path[n], cost[n] := m, weight((m, n)) + cost[m]$.
6: **return** $path, cost$.

Shortest and longest paths in directed acyclic graphs

Algorithm SSSP-DAG($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*, $s \in \mathcal{N}$):

$path, cost := [n \mapsto ? \mid n \in \mathcal{N}], [n \mapsto \infty \mid n \in \mathcal{N}]$.
$path[s], cost[s] := s, 0$.
for all $n \in \mathcal{N}$ in topological order (and that follow s) **do**
  for all $(m, n) \in \mathcal{E}$ with $cost[m] + weight((m, n)) < cost[n]$ **do**
    $path[n], cost[n] := m, weight((m, n)) + cost[m]$.
return $path, cost$.



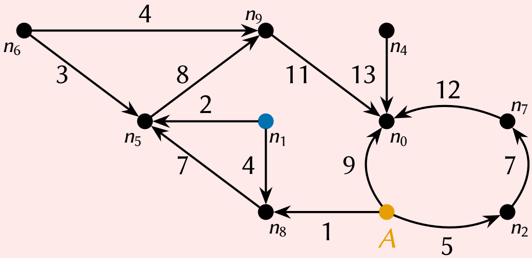
n_0	?	∞
n_1	?	∞
n_2	?	∞
A	A	0
n_4	?	∞
n_5	?	∞
n_6	?	∞
n_7	?	∞
n_8	?	∞
n_9	?	∞

Topological order: $n_4, n_6, A, n_1, n_8, n_2, n_7, n_5, n_9, n_0$.

Shortest and longest paths in directed acyclic graphs

Algorithm SSSP-DAG($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*, $s \in \mathcal{N}$):

$path, cost := [n \mapsto ? \mid n \in \mathcal{N}], [n \mapsto \infty \mid n \in \mathcal{N}]$.
$path[s], cost[s] := s, 0$.
for all $n \in \mathcal{N}$ in topological order (and that follow s) **do**
  for all $(m, n) \in \mathcal{E}$ with $cost[m] + weight((m, n)) < cost[n]$ **do**
    $path[n], cost[n] := m, weight((m, n)) + cost[m]$.
return $path, cost$.



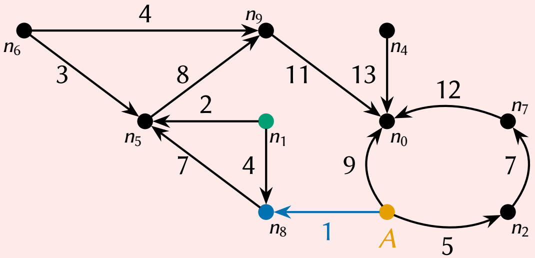
n_0	?	∞
n_1	?	∞
n_2	?	∞
A	A	0
n_4	?	∞
n_5	?	∞
n_6	?	∞
n_7	?	∞
n_8	?	∞
n_9	?	∞

Topological order: $n_4, n_6, A, n_1, n_8, n_2, n_7, n_5, n_9, n_0$.

Shortest and longest paths in directed acyclic graphs

Algorithm SSSP-DAG($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*, $s \in \mathcal{N}$):

$\mathit{path}, \mathit{cost} := [n \mapsto ? \mid n \in \mathcal{N}], [n \mapsto \infty \mid n \in \mathcal{N}]$.
$\mathit{path}[s], \mathit{cost}[s] := s, 0$.
for all $n \in \mathcal{N}$ in topological order (and that follow s) **do**
  for all $(m, n) \in \mathcal{E}$ with $\mathit{cost}[m] + \mathit{weight}((m, n)) < \mathit{cost}[n]$ **do**
    $\mathit{path}[n], \mathit{cost}[n] := m, \mathit{weight}((m, n)) + \mathit{cost}[m]$.
return $\mathit{path}, \mathit{cost}$.



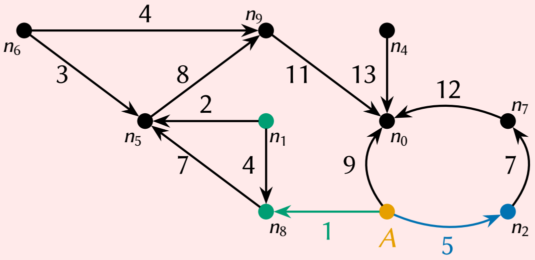
n_0	?	∞
n_1	?	∞
n_2	?	∞
A	A	0
n_4	?	∞
n_5	?	∞
n_6	?	∞
n_7	?	∞
n_8	A	1
n_9	?	∞

Topological order: $n_4, n_6, A, n_1, n_8, n_2, n_7, n_5, n_9, n_0$.

Shortest and longest paths in directed acyclic graphs

Algorithm SSSP-DAG($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*, $s \in \mathcal{N}$):

$path, cost := [n \mapsto ? \mid n \in \mathcal{N}], [n \mapsto \infty \mid n \in \mathcal{N}]$.
$path[s], cost[s] := s, 0$.
for all $n \in \mathcal{N}$ in topological order (and that follow s) **do**
  for all $(m, n) \in \mathcal{E}$ with $cost[m] + weight((m, n)) < cost[n]$ **do**
    $path[n], cost[n] := m, weight((m, n)) + cost[m]$.
return $path, cost$.



n_0	?	∞
n_1	?	∞
n_2	A	5
A	A	0
n_4	?	∞
n_5	?	∞
n_6	?	∞
n_7	?	∞
n_8	A	1
n_9	?	∞

Topological order: $n_4, n_6, A, n_1, n_8, n_2, n_7, n_5, n_9, n_0$.

Shortest and longest paths in directed acyclic graphs

Algorithm SSSP-DAG($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*, $s \in \mathcal{N}$):

$\mathit{path}, \mathit{cost} := [n \mapsto ? \mid n \in \mathcal{N}], [n \mapsto \infty \mid n \in \mathcal{N}]$.

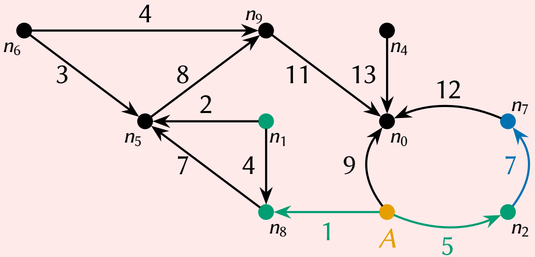
$\mathit{path}[s], \mathit{cost}[s] := s, 0$.

for all $n \in \mathcal{N}$ in topological order (and that follow s) **do**

    for all $(m, n) \in \mathcal{E}$ with $\mathit{cost}[m] + \mathit{weight}((m, n)) < \mathit{cost}[n]$ **do**

        $\mathit{path}[n], \mathit{cost}[n] := m, \mathit{weight}((m, n)) + \mathit{cost}[m]$.

return $\mathit{path}, \mathit{cost}$.



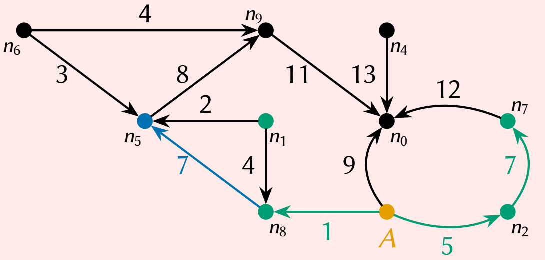
n_0	?	∞
n_1	?	∞
n_2	A	5
A	A	0
n_4	?	∞
n_5	?	∞
n_6	?	∞
n_7	n_2	12
n_8	A	1
n_9	?	∞

Topological order: $n_4, n_6, A, n_1, n_8, n_2, n_7, n_5, n_9, n_0$.

Shortest and longest paths in directed acyclic graphs

Algorithm SSSP-DAG($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $weight$, $s \in \mathcal{N}$):

$path, cost := [n \mapsto ?\mid n \in \mathcal{N}], [n \mapsto \infty \mid n \in \mathcal{N}]$.
$path[s], cost[s] := s, 0$.
for all $n \in \mathcal{N}$ in topological order (and that follow s) **do**
  for all $(m, n) \in \mathcal{E}$ with $cost[m] + weight((m, n)) < cost[n]$ **do**
    $path[n], cost[n] := m, weight((m, n)) + cost[m]$.
return $path, cost$.



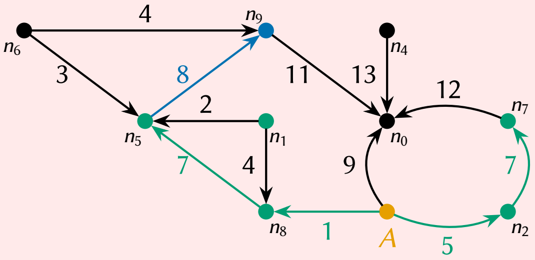
n_0	?	∞
n_1	?	∞
n_2	A	5
A	A	0
n_4	?	∞
n_5	n_8	8
n_6	?	∞
n_7	n_2	12
n_8	A	1
n_9	?	∞

Topological order: $n_4, n_6, A, n_1, n_8, n_2, n_7, n_5, n_9, n_0$.

Shortest and longest paths in directed acyclic graphs

Algorithm SSSP-DAG($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*, $s \in \mathcal{N}$):

$path, cost := [n \mapsto ? \mid n \in \mathcal{N}], [n \mapsto \infty \mid n \in \mathcal{N}]$.
$path[s], cost[s] := s, 0$.
for all $n \in \mathcal{N}$ in topological order (and that follow s) **do**
  for all $(m, n) \in \mathcal{E}$ with $cost[m] + weight((m, n)) < cost[n]$ **do**
    $path[n], cost[n] := m, weight((m, n)) + cost[m]$.
return $path, cost$.



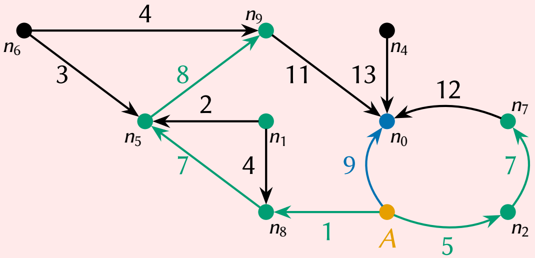
n_0	?	∞
n_1	?	∞
n_2	A	5
A	A	0
n_4	?	∞
n_5	n_8	8
n_6	?	∞
n_7	n_2	12
n_8	A	1
n_9	n_5	16

Topological order: $n_4, n_6, A, n_1, n_8, n_2, n_7, n_5, n_9, n_0$.

Shortest and longest paths in directed acyclic graphs

Algorithm SSSP-DAG($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $weight$, $s \in \mathcal{N}$):

$path, cost := [n \mapsto ? \mid n \in \mathcal{N}], [n \mapsto \infty \mid n \in \mathcal{N}]$.
$path[s], cost[s] := s, 0$.
for all $n \in \mathcal{N}$ in topological order (and that follow s) **do**
  for all $(m, n) \in \mathcal{E}$ with $cost[m] + weight((m, n)) < cost[n]$ **do**
    $path[n], cost[n] := m, weight((m, n)) + cost[m]$.
return $path, cost$.



n_0	A	9
n_1	?	∞
n_2	A	5
A	A	0
n_4	?	∞
n_5	n_8	8
n_6	?	∞
n_7	n_2	12
n_8	A	1
n_9	n_5	16

Topological order: $n_4, n_6, A, n_1, n_8, n_2, n_7, n_5, n_9, n_0$.

Shortest and longest paths in directed acyclic graphs

Algorithm SSSP-DAG($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*, $s \in \mathcal{N}$):

$\mathit{path}, \mathit{cost} := [n \mapsto ? \mid n \in \mathcal{N}], [n \mapsto \infty \mid n \in \mathcal{N}]$.
$\mathit{path}[s], \mathit{cost}[s] := s, 0$.
for all $n \in \mathcal{N}$ in topological order (and that follow s) **do**
  for all $(m, n) \in \mathcal{E}$ with $\mathit{cost}[m] + \mathit{weight}((m, n)) < \mathit{cost}[n]$ **do**
    $\mathit{path}[n], \mathit{cost}[n] := m, \mathit{weight}((m, n)) + \mathit{cost}[m]$.
return $\mathit{path}, \mathit{cost}$.

Complexity

- $\Theta(|\mathcal{N}|)$ to initialize *path* and *cost*.
- $\Theta(|\mathcal{N}| + |\mathcal{E}|)$ for topological sort.
- $\Theta(|\mathcal{N}| + |\mathcal{E}|)$ for visiting each node and edge in topological order.

Shortest and longest paths in directed acyclic graphs

Algorithm SSSP-DAG($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*, $s \in \mathcal{N}$):

1: $path, cost := [n \mapsto ? \mid n \in \mathcal{N}], [n \mapsto \infty \mid n \in \mathcal{N}]$.
2: $path[s], cost[s] := s, 0$.
3: **for all** $n \in \mathcal{N}$ in topological order (and that follow s) **do**
4:     **for all** $(m, n) \in \mathcal{E}$ with $cost[m] + weight((m, n)) < cost[n]$ **do**
5:         $path[n], cost[n] := m, weight((m, n)) + cost[m]$.
6: **return** $path, cost$.

Complexity

- $\Theta(|\mathcal{N}|)$ to initialize *path* and *cost*.
- $\Theta(|\mathcal{N}| + |\mathcal{E}|)$ for topological sort.
- $\Theta(|\mathcal{N}| + |\mathcal{E}|)$ for visiting each node and edge in topological order.

Total: $\Theta(|\mathcal{N}| + |\mathcal{E}|)$.

Shortest and longest paths in directed acyclic graphs

Algorithm SSSP-DAG($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*, $s \in \mathcal{N}$):

$\mathit{path}, \mathit{cost} := [n \mapsto ? \mid n \in \mathcal{N}], [n \mapsto \infty \mid n \in \mathcal{N}]$.

$\mathit{path}[s], \mathit{cost}[s] := s, 0$.

for all $n \in \mathcal{N}$ in topological order (and that follow s) **do**

    for all $(m, n) \in \mathcal{E}$ with $\mathit{cost}[m] + \mathit{weight}((m, n)) < \mathit{cost}[n]$ **do**

        $\mathit{path}[n], \mathit{cost}[n] := m, \mathit{weight}((m, n)) + \mathit{cost}[m]$.

return $\mathit{path}, \mathit{cost}$.

We observe that this method has no issues with *negative weights*.

Shortest and longest paths in directed acyclic graphs

Algorithm SSSP-DAG($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*, $s \in \mathcal{N}$):

$\mathit{path}, \mathit{cost} := [n \mapsto ? \mid n \in \mathcal{N}], [n \mapsto \infty \mid n \in \mathcal{N}]$.
$\mathit{path}[s], \mathit{cost}[s] := s, 0$.
for all $n \in \mathcal{N}$ in topological order (and that follow s) **do**
  for all $(m, n) \in \mathcal{E}$ with $\mathit{cost}[m] + \mathit{weight}((m, n)) < \mathit{cost}[n]$ **do**
    $\mathit{path}[n], \mathit{cost}[n] := m, \mathit{weight}((m, n)) + \mathit{cost}[m]$.
return $\mathit{path}, \mathit{cost}$.

We observe that this method has no issues with *negative weights*.

Longest paths in a directed acyclic graph?

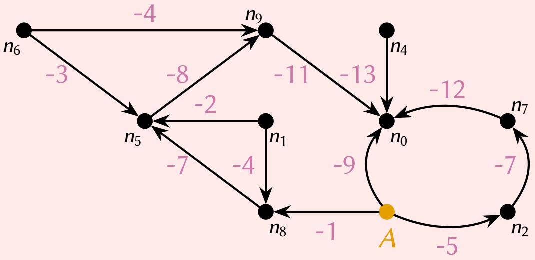
For every edge $(m, n) \in \mathcal{E}$, replace weights $\mathit{weight}((m, n))$ by $-\mathit{weight}((m, n))$ and then compute the shortest paths with respect to these *negated* weights.

$\longrightarrow$ The longest path becomes the path with the *most negative* cost.

Shortest and longest paths in directed acyclic graphs

Algorithm SSSP-DAG($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*, $s \in \mathcal{N}$):

$\mathit{path}, \mathit{cost} := [n \mapsto ? \mid n \in \mathcal{N}], [n \mapsto \infty \mid n \in \mathcal{N}]$.
$\mathit{path}[s], \mathit{cost}[s] := s, 0$.
for all $n \in \mathcal{N}$ in topological order (and that follow s) **do**
  for all $(m, n) \in \mathcal{E}$ with $\mathit{cost}[m] + \mathit{weight}((m, n)) < \mathit{cost}[n]$ **do**
    $\mathit{path}[n], \mathit{cost}[n] := m, \mathit{weight}((m, n)) + \mathit{cost}[m]$.
return $\mathit{path}, \mathit{cost}$.



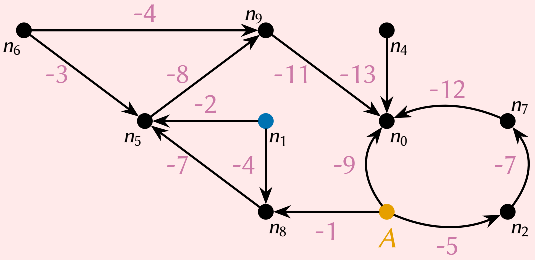
n_0	?	∞
n_1	?	∞
n_2	?	∞
A	A	0
n_4	?	∞
n_5	?	∞
n_6	?	∞
n_7	?	∞
n_8	?	∞
n_9	?	∞

Topological order: $n_4, n_6, A, n_1, n_8, n_2, n_7, n_5, n_9, n_0$.

Shortest and longest paths in directed acyclic graphs

Algorithm SSSP-DAG($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*, $s \in \mathcal{N}$):

$path, cost := [n \mapsto ? \mid n \in \mathcal{N}], [n \mapsto \infty \mid n \in \mathcal{N}]$.
$path[s], cost[s] := s, 0$.
for all $n \in \mathcal{N}$ in topological order (and that follow s) **do**
  for all $(m, n) \in \mathcal{E}$ with $cost[m] + weight((m, n)) < cost[n]$ **do**
    $path[n], cost[n] := m, weight((m, n)) + cost[m]$.
return $path, cost$.



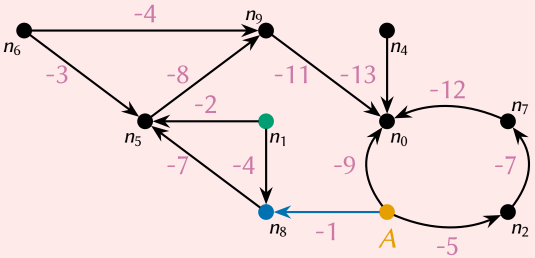
n_0	?	∞
n_1	?	∞
n_2	?	∞
A	A	0
n_4	?	∞
n_5	?	∞
n_6	?	∞
n_7	?	∞
n_8	?	∞
n_9	?	∞

Topological order: $n_4, n_6, A, n_1, n_8, n_2, n_7, n_5, n_9, n_0$.

Shortest and longest paths in directed acyclic graphs

Algorithm SSSP-DAG($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*, $s \in \mathcal{N}$):

$path, cost := [n \mapsto ? \mid n \in \mathcal{N}], [n \mapsto \infty \mid n \in \mathcal{N}]$.
$path[s], cost[s] := s, 0$.
for all $n \in \mathcal{N}$ in topological order (and that follow s) **do**
  for all $(m, n) \in \mathcal{E}$ with $cost[m] + weight((m, n)) < cost[n]$ **do**
    $path[n], cost[n] := m, weight((m, n)) + cost[m]$.
return $path, cost$.



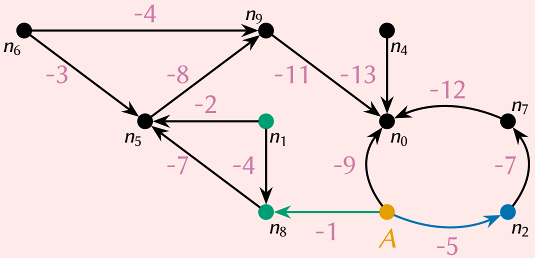
n_0	?	∞
n_1	?	∞
n_2	?	∞
A	A	0
n_4	?	∞
n_5	?	∞
n_6	?	∞
n_7	?	∞
n_8	A	-1
n_9	?	∞

Topological order: $n_4, n_6, A, n_1, n_8, n_2, n_7, n_5, n_9, n_0$.

Shortest and longest paths in directed acyclic graphs

Algorithm SSSP-DAG($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, $weight$, $s \in \mathcal{N}$):

$path, cost := [n \mapsto ?\mid n \in \mathcal{N}], [n \mapsto \infty \mid n \in \mathcal{N}]$.
$path[s], cost[s] := s, 0$.
for all $n \in \mathcal{N}$ in topological order (and that follow s) **do**
  for all $(m, n) \in \mathcal{E}$ with $cost[m] + weight((m, n)) < cost[n]$ **do**
    $path[n], cost[n] := m, weight((m, n)) + cost[m]$.
return $path, cost$.



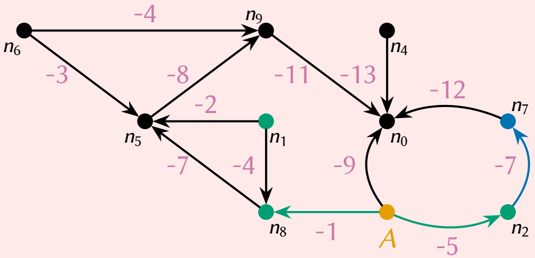
n_0	?	∞
n_1	?	∞
n_2	A	-5
A	A	0
n_4	?	∞
n_5	?	∞
n_6	?	∞
n_7	?	∞
n_8	A	-1
n_9	?	∞

Topological order: $n_4, n_6, A, n_1, n_8, n_2, n_7, n_5, n_9, n_0$.

Shortest and longest paths in directed acyclic graphs

Algorithm SSSP-DAG($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*, $s \in \mathcal{N}$):

$path, cost := [n \mapsto ? \mid n \in \mathcal{N}], [n \mapsto \infty \mid n \in \mathcal{N}]$.
$path[s], cost[s] := s, 0$.
for all $n \in \mathcal{N}$ in topological order (and that follow s) **do**
  for all $(m, n) \in \mathcal{E}$ with $cost[m] + weight((m, n)) < cost[n]$ **do**
    $path[n], cost[n] := m, weight((m, n)) + cost[m]$.
return $path, cost$.



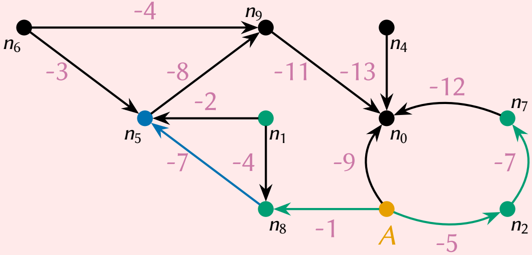
n_0	?	∞
n_1	?	∞
n_2	A	-5
A	A	0
n_4	?	∞
n_5	?	∞
n_6	?	∞
n_7	n_2	-12
n_8	A	-1
n_9	?	∞

Topological order: $n_4, n_6, A, n_1, n_8, n_2, n_7, n_5, n_9, n_0$.

Shortest and longest paths in directed acyclic graphs

Algorithm SSSP-DAG($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*, $s \in \mathcal{N}$):

$path, cost := [n \mapsto ? \mid n \in \mathcal{N}], [n \mapsto \infty \mid n \in \mathcal{N}]$.
$path[s], cost[s] := s, 0$.
for all $n \in \mathcal{N}$ in topological order (and that follow s) **do**
  for all $(m, n) \in \mathcal{E}$ with $cost[m] + weight((m, n)) < cost[n]$ **do**
    $path[n], cost[n] := m, weight((m, n)) + cost[m]$.
return $path, cost$.



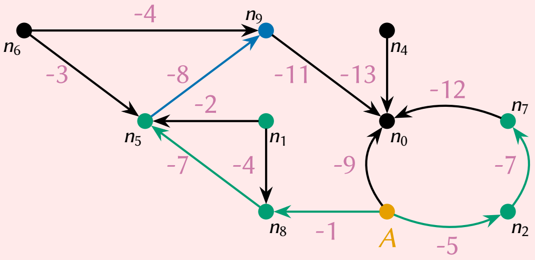
n_0	?	∞
n_1	?	∞
n_2	A	-5
A	A	0
n_4	?	∞
n_5	n_8	-8
n_6	?	∞
n_7	n_2	-12
n_8	A	-1
n_9	?	∞

Topological order: $n_4, n_6, A, n_1, n_8, n_2, n_7, n_5, n_9, n_0$.

Shortest and longest paths in directed acyclic graphs

Algorithm SSSP-DAG($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*, $s \in \mathcal{N}$):

$path, cost := [n \mapsto ? \mid n \in \mathcal{N}], [n \mapsto \infty \mid n \in \mathcal{N}]$.
$path[s], cost[s] := s, 0$.
for all $n \in \mathcal{N}$ in topological order (and that follow s) **do**
  for all $(m, n) \in \mathcal{E}$ with $cost[m] + weight((m, n)) < cost[n]$ **do**
    $path[n], cost[n] := m, weight((m, n)) + cost[m]$.
return $path, cost$.



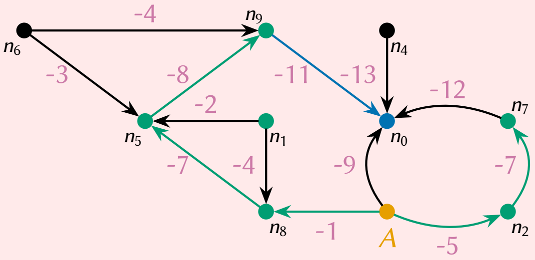
n_0	?	∞
n_1	?	∞
n_2	A	-5
A	A	0
n_4	?	∞
n_5	n_8	-8
n_6	?	∞
n_7	n_2	-12
n_8	A	-1
n_9	n_5	-16

Topological order: $n_4, n_6, A, n_1, n_8, n_2, n_7, n_5, n_9, n_0$.

Shortest and longest paths in directed acyclic graphs

Algorithm SSSP-DAG($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*, $s \in \mathcal{N}$):

$path, cost := [n \mapsto ? \mid n \in \mathcal{N}], [n \mapsto \infty \mid n \in \mathcal{N}]$.
$path[s], cost[s] := s, 0$.
for all $n \in \mathcal{N}$ in topological order (and that follow s) **do**
  for all $(m, n) \in \mathcal{E}$ with $cost[m] + weight((m, n)) < cost[n]$ **do**
    $path[n], cost[n] := m, weight((m, n)) + cost[m]$.
return $path, cost$.



n_0	n_9	-27
n_1	?	∞
n_2	A	-5
A	A	0
n_4	?	∞
n_5	n_8	-8
n_6	?	∞
n_7	n_2	-12
n_8	A	-1
n_9	n_5	-16

Topological order: $n_4, n_6, A, n_1, n_8, n_2, n_7, n_5, n_9, n_0$.

Shortest and longest paths in directed acyclic graphs

Algorithm SSSP-DAG($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*, $s \in \mathcal{N}$)**:**

$\mathit{path}, \mathit{cost} := [n \mapsto ? \mid n \in \mathcal{N}], [n \mapsto \infty \mid n \in \mathcal{N}]$.
$\mathit{path}[s], \mathit{cost}[s] := s, 0$.
for all $n \in \mathcal{N}$ in topological order (and that follow s) **do**
  for all $(m, n) \in \mathcal{E}$ with $\mathit{cost}[m] + \mathit{weight}((m, n)) < \mathit{cost}[n]$ **do**
    $\mathit{path}[n], \mathit{cost}[n] := m, \mathit{weight}((m, n)) + \mathit{cost}[m]$.
return $\mathit{path}, \mathit{cost}$.

We observe that this method has no issues with *negative weights*.

Longest paths in a directed acyclic graph?

The *longest path* in $\Theta(|\mathcal{N}| + |\mathcal{E}|)$.

Shortest and longest paths in directed acyclic graphs

Algorithm SSSP-DAG($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*, $s \in \mathcal{N}$):

$path, cost := [n \mapsto ? \mid n \in \mathcal{N}], [n \mapsto \infty \mid n \in \mathcal{N}]$.
$path[s], cost[s] := s, 0$.
for all $n \in \mathcal{N}$ in topological order (and that follow s) **do**
  for all $(m, n) \in \mathcal{E}$ with $cost[m] + weight((m, n)) < cost[n]$ **do**
    $path[n], cost[n] := m, weight((m, n)) + cost[m]$.
return $path, cost$.

We observe that this method has no issues with *negative weights*.

Longest paths in a directed acyclic graph?

The *longest path* in $\Theta(|\mathcal{N}| + |\mathcal{E}|)$.

This *only* works for directed acyclic graphs:
Determining whether a longest path without node repetition of cost k exists in a graph is an *NP-complete problem*: no practical algorithms known to solve this problem!

Shortest paths and negative weights

- SSSP-Dijkstra requires non-negative weights.
- SSSP-DAG requires a directed acyclic graph.

Shortest paths and negative weights

- SSSP-Dijkstra requires non-negative weights.
- SSSP-DAG requires a directed acyclic graph.

Problem

Given a directed edge-weighted graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ and source node $s \in \mathcal{N}$.
For every target node $t \in \mathcal{N}$,

- either detect that there is a negative-cost cycle on *a* path from s to t; or
- find *a* shortest path (if any) from s to t.

Shortest paths and negative weights

- SSSP-Dijkstra requires non-negative weights.
- SSSP-DAG requires a directed acyclic graph.

Problem

Given a directed edge-weighted graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ and source node $s \in \mathcal{N}$.
For every target node $t \in \mathcal{N}$,

- either detect that there is a negative-cost cycle on *a* path from s to t; or
- find *a* shortest path (if any) from s to t.

- We cannot "simply" eliminate negative weights by adding a sufficiently-positive number: this will distort path lengths of paths consisting of many edges.

Shortest paths and negative weights

Algorithm SSSP-HighLevel($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*, $s \in \mathcal{N}$)**:**

$\mathit{path}, \mathit{cost} := [n \mapsto ? \mid n \in \mathcal{N}], [n \mapsto \infty \mid n \in \mathcal{N}]$.
$\mathit{path}[s], \mathit{cost}[s] := s, 0$.
while $(m, n) \in \mathcal{E}$ with $\mathit{cost}[m] + \mathit{weight}((m, n)) < \mathit{cost}[n]$ **do**
    $\mathit{path}[n], \mathit{cost}[n] := m, \mathit{weight}((m, n)) + \mathit{cost}[m]$.
return $\mathit{path}, \mathit{cost}$.

Algorithm SSSP-Bellman-Ford($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*, $s \in \mathcal{N}$)**:**

$\mathit{path}, \mathit{cost} := [n \mapsto ? \mid n \in \mathcal{N}], [n \mapsto \infty \mid n \in \mathcal{N}]$.
$\mathit{path}[s], \mathit{cost}[s] := s, 0$.
for $i := 1$ upto $|\mathcal{N}| - 1$ **do**
    for all $(m, n) \in \mathcal{E}$ with $\mathit{cost}[m] + \mathit{weight}((m, n)) < \mathit{cost}[n]$ **do**
        $\mathit{path}[n], \mathit{cost}[n] := m, \mathit{weight}((m, n)) + \mathit{cost}[m]$.
return $\mathit{path}, \mathit{cost}$.

Shortest paths and negative weights

Algorithm SSSP-BELLMAN-FORD($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*, $s \in \mathcal{N}$):

$path, cost := [n \mapsto ? \mid n \in \mathcal{N}], [n \mapsto \infty \mid n \in \mathcal{N}]$.

$path[s], cost[s] := s, 0$.

for $i := 1$ upto $|\mathcal{N}| - 1$ **do**

    for all $(m, n) \in \mathcal{E}$ with $cost[m] + weight((m, n)) < cost[n]$ **do**

        $path[n], cost[n] := m, weight((m, n)) + cost[m]$.

if there is an eligible edge **then**

    Found a negative-cost cycle.

return $path, cost$.

Shortest paths and negative weights

Algorithm SSSP-BELLMAN-FORD($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*, $s \in \mathcal{N}$):

$path, cost := [n \mapsto ? \mid n \in \mathcal{N}], [n \mapsto \infty \mid n \in \mathcal{N}]$.
$path[s], cost[s] := s, 0$.
for $i := 1$ upto $|\mathcal{N}| - 1$ **do**
  for all $(m, n) \in \mathcal{E}$ with $cost[m] + weight((m, n)) < cost[n]$ **do**
    $path[n], cost[n] := m, weight((m, n)) + cost[m]$.
if there is an eligible edge **then**
  Found a negative-cost cycle.
return $path, cost$.

Theorem

Algorithm SSSP-BELLMAN-FORD is correct.

Shortest paths and negative weights

Algorithm SSSP-BELLMAN-FORD($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*, $s \in \mathcal{N}$):

$path, cost := [n \mapsto ? \mid n \in \mathcal{N}], [n \mapsto \infty \mid n \in \mathcal{N}]$.
$path[s], cost[s] := s, 0$.
for $i := 1$ upto $|\mathcal{N}| - 1$ **do**
    for all $(m, n) \in \mathcal{E}$ with $cost[m] + weight((m, n)) < cost[n]$ **do**
        $path[n], cost[n] := m, weight((m, n)) + cost[m]$.
if there is an eligible edge **then**
    Found a negative-cost cycle.
return $path, cost$.

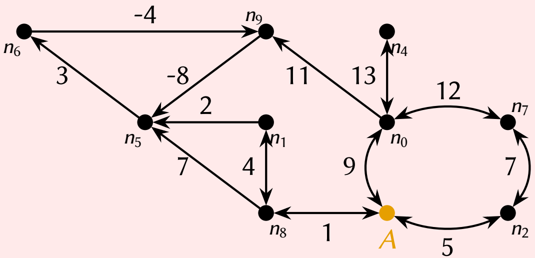
Theorem

Algorithm SSSP-BELLMAN-FORD is correct.

Proof

Invariant: If there is a shortest path $sn_1 \dots n_{i-1}$ of $i - 1$ edges, then *cost* and *path* represent a shortest path from s to n_{i-1}.

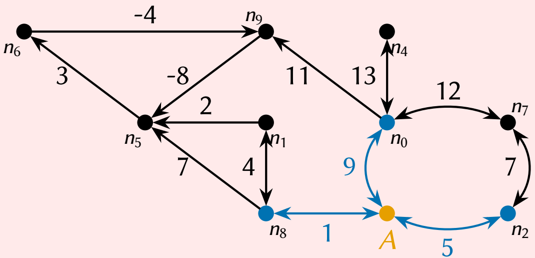
Shortest paths and negative weights



n_0:	?	∞
n_1:	?	∞
n_2:	?	∞
A:	A	0
n_4:	?	∞
n_5:	?	∞
n_6:	?	∞
n_7:	?	∞
n_8:	?	∞
n_9:	?	∞

Shortest paths and negative weights

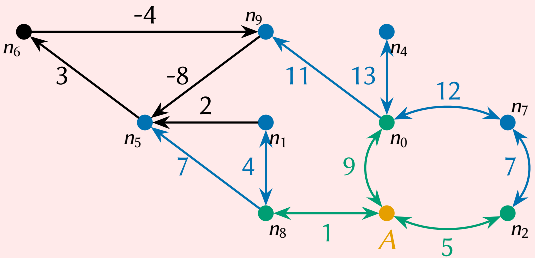
$i := 1$.



n_0:	A	9
n_1:	?	∞
n_2:	A	5
A:	A	0
n_4:	?	∞
n_5:	?	∞
n_6:	?	∞
n_7:	?	∞
n_8:	A	1
n_9:	?	∞

Shortest paths and negative weights

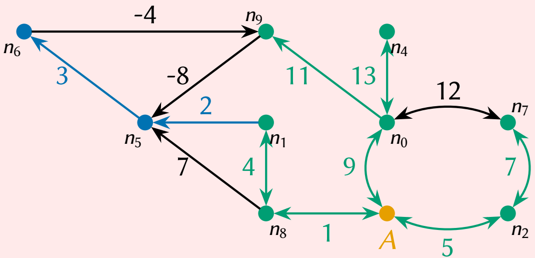
$i := 2.$



n_0:	A	9
n_1:	n_8	5
n_2:	A	5
A:	A	0
n_4:	n_0	22
n_5:	n_8	8
n_6:	?	∞
n_7:	n_2	12
n_8:	A	1
n_9:	n_0	20

Shortest paths and negative weights

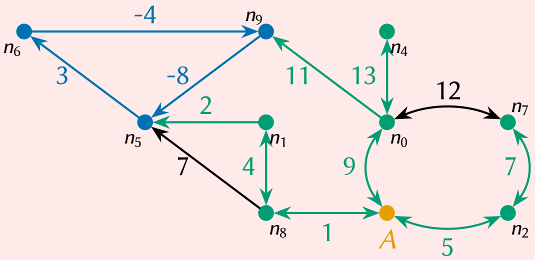
$i := 3.$



n_0:	A	9
n_1:	n_8	5
n_2:	A	5
A:	A	0
n_4:	n_0	22
n_5:	n_1	7
n_6:	n_5	10
n_7:	n_2	12
n_8:	A	1
n_9:	n_0	20

Shortest paths and negative weights

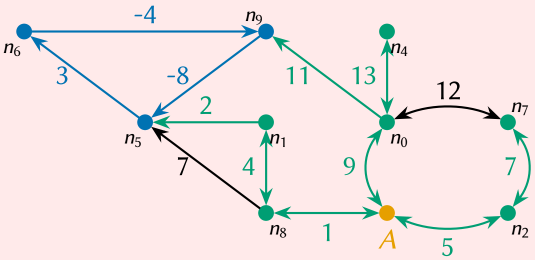
$i := 4$.



n_0:	A	9
n_1:	n_8	5
n_2:	A	5
A:	A	0
n_4:	n_0	22
n_5:	n_9	-2
n_6:	n_5	1
n_7:	n_2	12
n_8:	A	1
n_9:	n_6	6

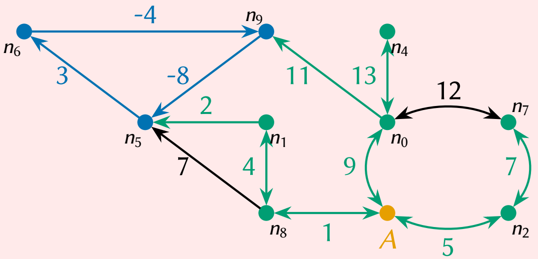
Shortest paths and negative weights

$i := 5, \dots, 9.$



n_0:	A	9
n_1:	n_8	5
n_2:	A	5
A:	A	0
n_4:	n_0	22
n_5:	n_9	$\dots$
n_6:	n_5	$\dots$
n_7:	n_2	12
n_8:	A	1
n_9:	n_6	$\dots$

Shortest paths and negative weights



n_0:	A	9
n_1:	n_8	5
n_2:	A	5
A:	A	0
n_4:	n_0	22
n_5:	n_9	$\dots$
n_6:	n_5	$\dots$
n_7:	n_2	12
n_8:	A	1
n_9:	n_6	$\dots$

As there are eligible edges: there exists a negative-cost cycle.

Shortest paths and negative weights

Algorithm SSSP-BELLMAN-FORD($\mathcal{G} = (\mathcal{N}, \mathcal{E})$, *weight*, $s \in \mathcal{N}$):

$path, cost := [n \mapsto ? \mid n \in \mathcal{N}], [n \mapsto \infty \mid n \in \mathcal{N}]$.
$path[s], cost[s] := s, 0$.
for $i := 1$ upto $|\mathcal{N}| - 1$ **do**
    for all $(m, n) \in \mathcal{E}$ with $cost[m] + weight((m, n)) < cost[n]$ **do**
        $path[n], cost[n] := m, weight((m, n)) + cost[m]$.
if there is an eligible edge **then**
    Found a negative-cost cycle.
return $path, cost$.

Complexity: $\Theta\left(|\mathcal{N}||\mathcal{E}|\right)$.

Example: Arbitrage

Consider a *currency exchange* where one can exchange *some* currencies X for currency Y at exchange rate $r(X, Y)$. For example $r(\mathrm{CAD}, \mathrm{EUR}) = 0.68$.

$$
\begin{tikzcd}
\mathrm{EUR} \arrow[d, "0.99"'] \arrow[dr, "1.08"] & \mathrm{CAD} \arrow[l, "0.68"'] \\
\mathrm{CHF} \arrow[r, "1.11"'] & \mathrm{USD} \arrow[u, "1.38"']
\end{tikzcd}
$$

Example: Arbitrage

Consider a *currency exchange* where one can exchange *some* currencies X for currency Y at exchange rate $r(X, Y)$.
For example $r(\mathrm{CAD}, \mathrm{EUR}) = 0.68$.

$$
\begin{tikzcd}
\mathrm{EUR} \arrow[d, "0.99"'] \arrow[dr, "1.08"] & \mathrm{CAD} \arrow[l, "0.68"'] \\
\mathrm{CHF} \arrow[r, "1.11"'] & \mathrm{USD} \arrow[u, "1.38"']
\end{tikzcd}
$$

The arbitrage problem

Is there a sequence of currencies $C_1, \ldots, C_n, C_1$ such that exchanging X units of C_1 for C_2, exchanging C_2 for C_3, ..., exchanging C_{n-1} for C_n, and exchanging C_n back to Y units of C_1 yields *a profit* $(Y > X)$.

Example: Arbitrage

Consider a *currency exchange* where one can exchange *some* currencies X for currency Y at exchange rate $r(X, Y)$.
For example $r(\mathrm{CAD}, \mathrm{EUR}) = 0.68$.

100EUR

$$\begin{tikzcd}
\mathrm{EUR} \arrow[d, "0.99"'] \arrow[dr, "1.08"] & \mathrm{CAD} \arrow[l, "0.68"'] \\
\mathrm{CHF} \arrow[r, "1.11"'] & \mathrm{USD} \arrow[u, "1.38"']
\end{tikzcd}$$

The arbitrage problem

Is there a sequence of currencies $C_1, \ldots, C_n, C_1$ such that exchanging X units of C_1 for C_2, exchanging C_2 for $C_3, \ldots,$ exchanging C_{n-1} for C_n, and exchanging C_n back to Y units of C_1 yields *a profit* ($Y > X$).

Example: Arbitrage

Consider a *currency exchange* where one can exchange *some* currencies X for currency Y at exchange rate $r(X, Y)$.
For example $r(\text{CAD}, \text{EUR}) = 0.68$.

$100\text{EUR} \longrightarrow 99\text{CHF}$

$$
\begin{tikzcd}
\text{EUR} \arrow[d, "0.99"'] \arrow[dr, "1.08"] & \text{CAD} \arrow[l, "0.68"'] \\
\text{CHF} \arrow[r, "1.11"'] & \text{USD} \arrow[u, "1.38"']
\end{tikzcd}
$$

The arbitrage problem

Is there a sequence of currencies $C_1, \ldots, C_n, C_1$ such that exchanging X units of C_1 for C_2, exchanging C_2 for $C_3, \ldots$, exchanging C_{n-1} for C_n, and exchanging C_n back to Y units of C_1 yields *a profit* ($Y > X$).

Example: Arbitrage

Consider a *currency exchange* where one can exchange *some* currencies X for currency Y at exchange rate $r(X, Y)$.
For example $r(\mathrm{CAD}, \mathrm{EUR}) = 0.68$.

$$100\mathrm{EUR} \longrightarrow 99\mathrm{CHF}$$
$$\longrightarrow 109.89\mathrm{USD}$$

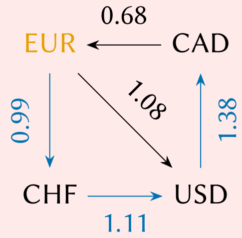
$$\begin{tikzcd}
\mathrm{EUR} \arrow[d, "0.99"'] \arrow[dr, "1.08"] & \mathrm{CAD} \arrow[l, "0.68"'] \\
\mathrm{CHF} \arrow[r, "1.11"'] & \mathrm{USD} \arrow[u, "1.38"']
\end{tikzcd}$$

The arbitrage problem

Is there a sequence of currencies $C_1, \ldots, C_n, C_1$ such that exchanging X units of C_1 for C_2, exchanging C_2 for C_3, …, exchanging C_{n-1} for C_n, and exchanging C_n back to Y units of C_1 yields *a profit* ($Y > X$).

Example: Arbitrage

Consider a *currency exchange* where one can exchange *some* currencies X for currency Y at exchange rate $r(X, Y)$.
For example $r(\mathrm{CAD}, \mathrm{EUR}) = 0.68$.



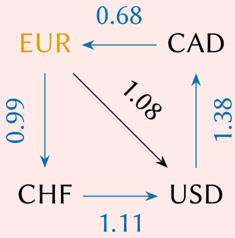
$$\begin{aligned} 100\mathrm{EUR} &\longrightarrow 99\mathrm{CHF} \\ &\longrightarrow 109.89\mathrm{USD} \\ &\longrightarrow 151.64\mathrm{CAD} \end{aligned}$$

The arbitrage problem

Is there a sequence of currencies $C_1, \ldots, C_n, C_1$ such that exchanging X units of C_1 for C_2, exchanging C_2 for C_3, ..., exchanging C_{n-1} for C_n, and exchanging C_n back to Y units of C_1 yields *a profit* ($Y > X$).

Example: Arbitrage

Consider a *currency exchange* where one can exchange *some* currencies X for currency Y at exchange rate $r(X, Y)$.
For example $r(\text{CAD}, \text{EUR}) = 0.68$.



$$\begin{aligned}100\text{EUR} &\longrightarrow 99\text{CHF}\\ &\longrightarrow 109.89\text{USD}\\ &\longrightarrow 151.64\text{CAD}\\ &\longrightarrow 103.12\text{EUR}\end{aligned}$$

The arbitrage problem

Is there a sequence of currencies $C_1, \ldots, C_n, C_1$ such that exchanging X units of C_1 for C_2, exchanging C_2 for C_3, ..., exchanging C_{n-1} for C_n, and exchanging C_n back to Y units of C_1 yields *a profit* ($Y > X$).

Example: Arbitrage

The arbitrage problem

Is there a sequence of currencies $C_1, \ldots, C_n, C_1$ such that exchanging X units of C_1 for C_2, exchanging C_2 for C_3, …, exchanging C_{n-1} for C_n, and exchanging C_n back to Y units of C_1 yields *a profit* ($Y > X$).

Find a cycle $C_1, \ldots, C_n, C_1$ such that

$$r(C_1, C_2) \times r(C_2, C_3) \times \cdots \times r(C_n, C_1) > 1 \text{ and is as large as possible.}$$

Example: Arbitrage

The arbitrage problem

Is there a sequence of currencies $C_1, \ldots, C_n, C_1$ such that exchanging X units of C_1 for C_2, exchanging C_2 for C_3, …, exchanging C_{n-1} for C_n, and exchanging C_n back to Y units of C_1 yields *a profit* ($Y > X$).

Find a cycle $C_1, \ldots, C_n, C_1$ such that

$$\log(r(C_1, C_2)) + \log(r(C_2, C_3)) + \cdots + \log(r(C_n, C_1)) > \log(1)$$

and is as large as possible.

Example: Arbitrage

The arbitrage problem

Is there a sequence of currencies $C_1, \dots, C_n, C_1$ such that exchanging X units of C_1 for C_2, exchanging C_2 for C_3, ..., exchanging C_{n-1} for C_n, and exchanging C_n back to Y units of C_1 yields *a profit* ($Y > X$).

Find a cycle $C_1, \dots, C_n, C_1$ such that

$$\log(r(C_1, C_2)) + \log(r(C_2, C_3)) + \cdots + \log(r(C_n, C_1)) > 0$$

and is as large as possible.

Example: Arbitrage

The arbitrage problem

Is there a sequence of currencies $C_1, \ldots, C_n, C_1$ such that exchanging X units of C_1 for C_2, exchanging C_2 for C_3, ..., exchanging C_{n-1} for C_n, and exchanging C_n back to Y units of C_1 yields *a profit* ($Y > X$).

Find a cycle $C_1, \ldots, C_n, C_1$ such that

$$(-\log(r(C_1, C_2))) + (-\log(r(C_2, C_3))) + \cdots + (-\log(r(C_n, C_1))) < 0$$

and is as *small* as possible.

Example: Arbitrage

The arbitrage problem

Is there a sequence of currencies $C_1, \ldots, C_n, C_1$ such that exchanging X units of C_1 for C_2, exchanging C_2 for C_3, …, exchanging C_{n-1} for C_n, and exchanging C_n back to Y units of C_1 yields *a profit* $(Y > X)$.

Find a cycle $C_1, \ldots, C_n, C_1$ such that

$$(-\log(r(C_1, C_2))) + (-\log(r(C_2, C_3))) + \cdots + (-\log(r(C_n, C_1))) < 0$$

and is as *small* as possible.

Solution

Use SSSP-Bellman-Ford with $\mathit{weight}((m, n)) = -\log(r(m, n))$ and find a negative cycle!